

Software Design

A thick, horizontal yellow brushstroke with a textured, painterly appearance, extending across the width of the slide and positioned directly beneath the title text.

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

Ñ Introduction to software engineering

Ñ Life cycle models

Ñ Requirements Analysis and Specification:

- y Requirements gathering and analysis

- y 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

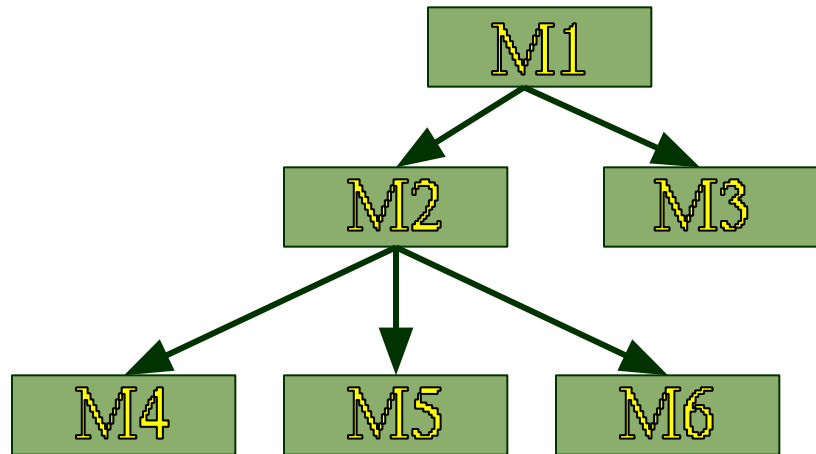
- Ñ Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language.
- Ñ A good software design is seldom arrived by using a single step procedure but rather through several iterations through a series of steps.
- Ñ Design activities can be broadly classified into two important parts:
 - y Preliminary (or high-level) design and
 - y Detailed design

Items Designed During Design Phase

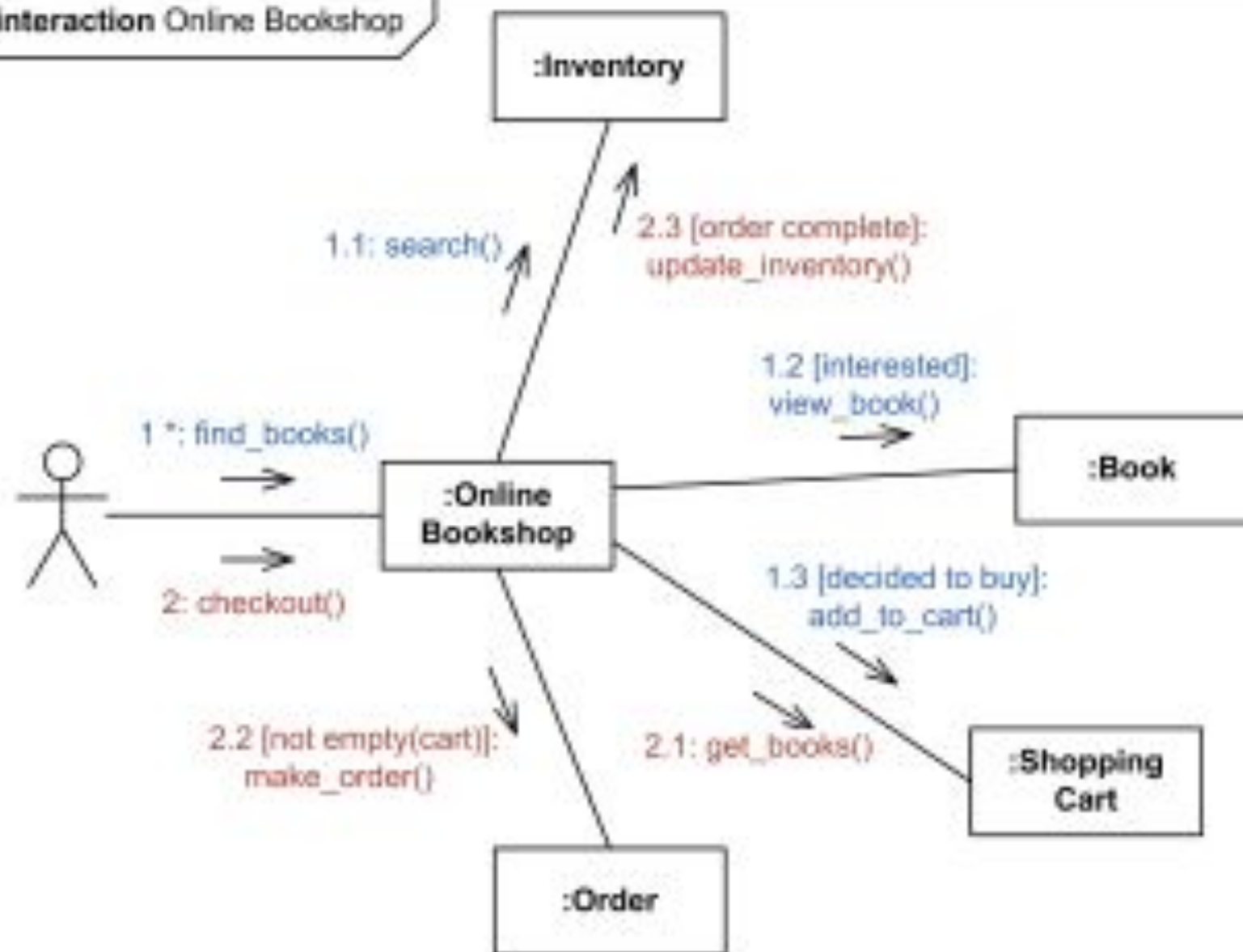


- Ñ module structure,
- Ñ control relationship among the modules
 - y call relationship or invocation relationship
- Ñ interface among different modules,
 - y data items exchanged among different modules,
- Ñ data structures of individual modules,
- Ñ algorithms for individual modules.

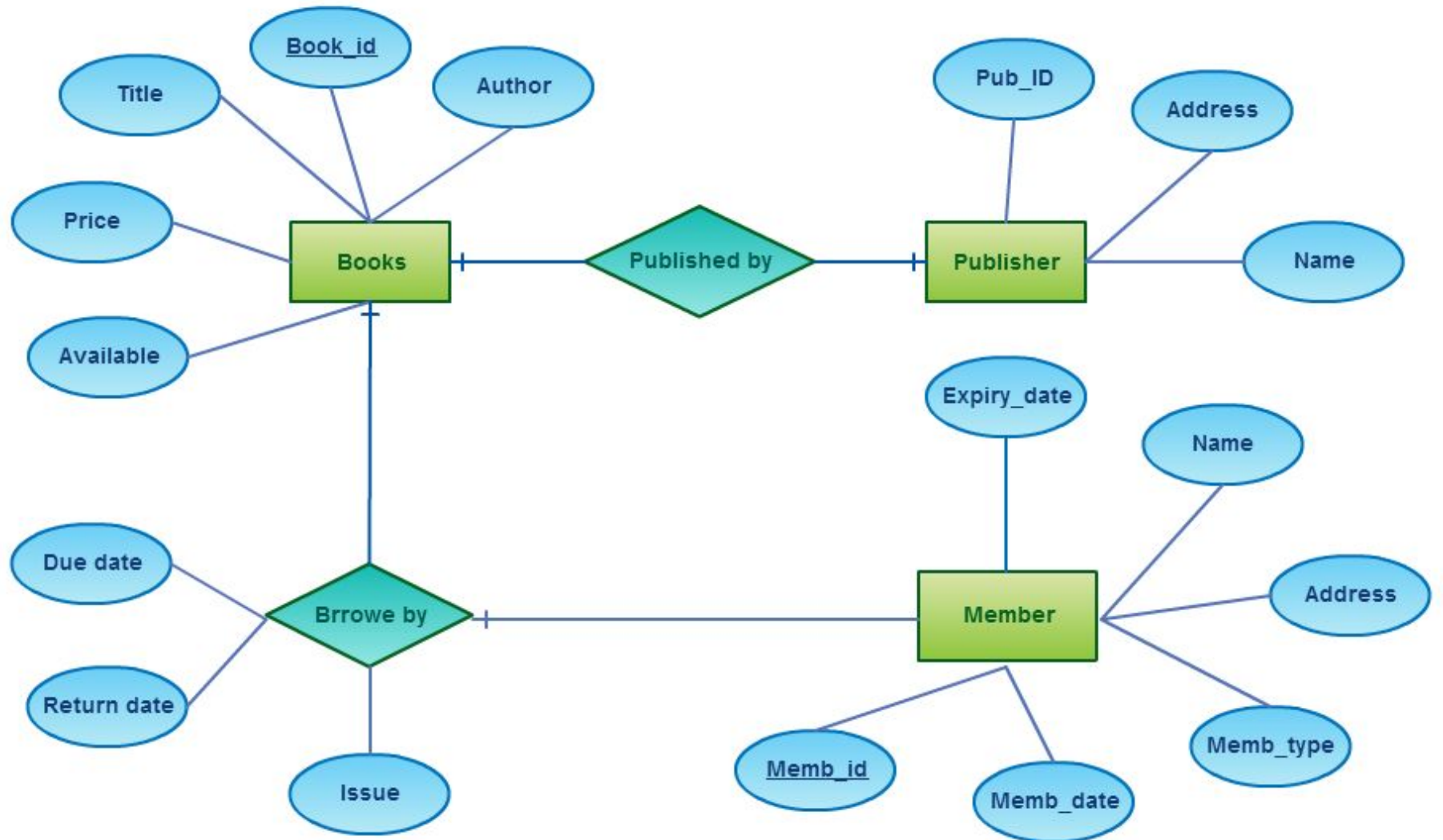
Module Structure



interaction Online Bookshop

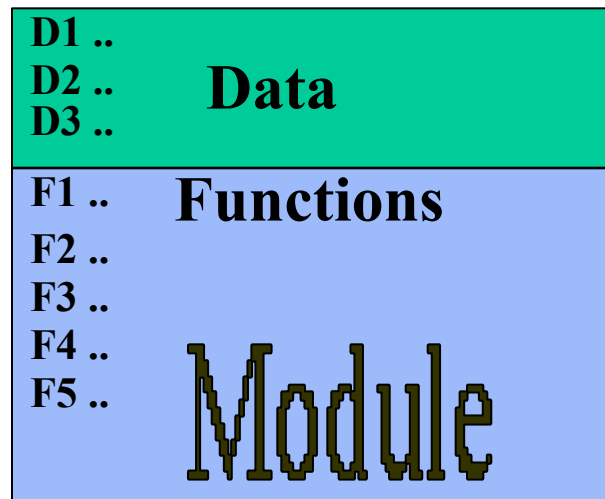


E-R Diagram of Library Management System



Introduction

Ñ A module consists of:
y several functions
y 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?

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

Understandability

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

Modularity

Ñ Modularity is a fundamental attributes of any good design.

- y Decomposition of a problem cleanly into modules:
- y Modules are almost independent of each other
- y divide and conquer principle.

Modularity

Ñ If modules are independent:

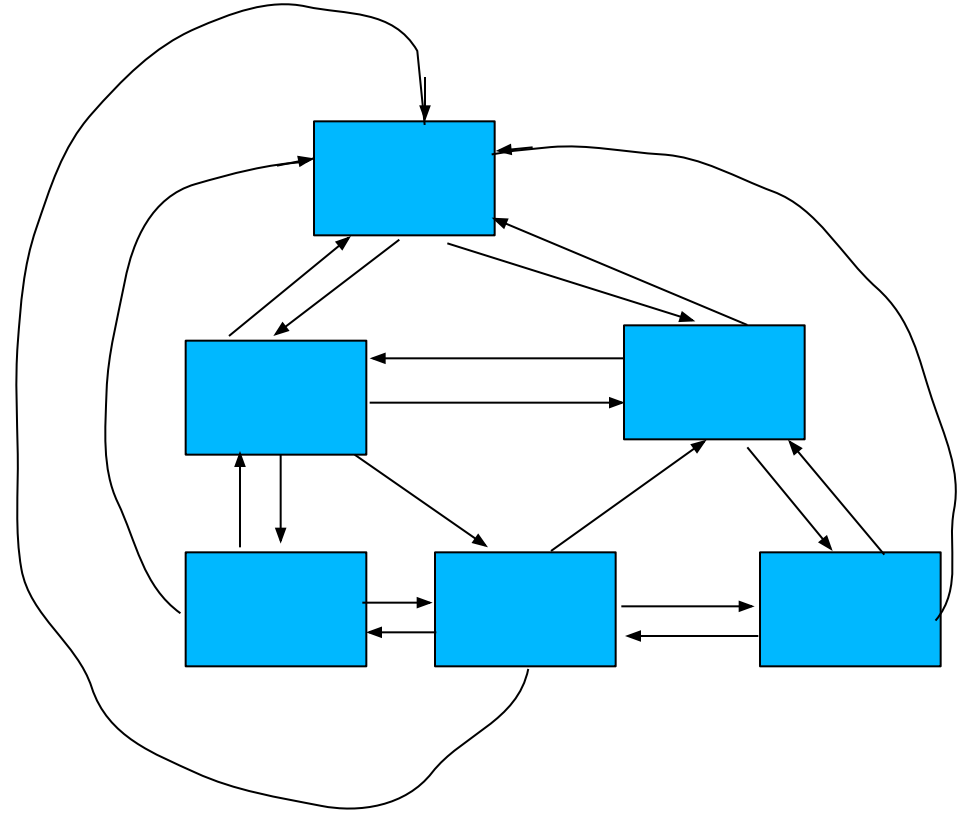
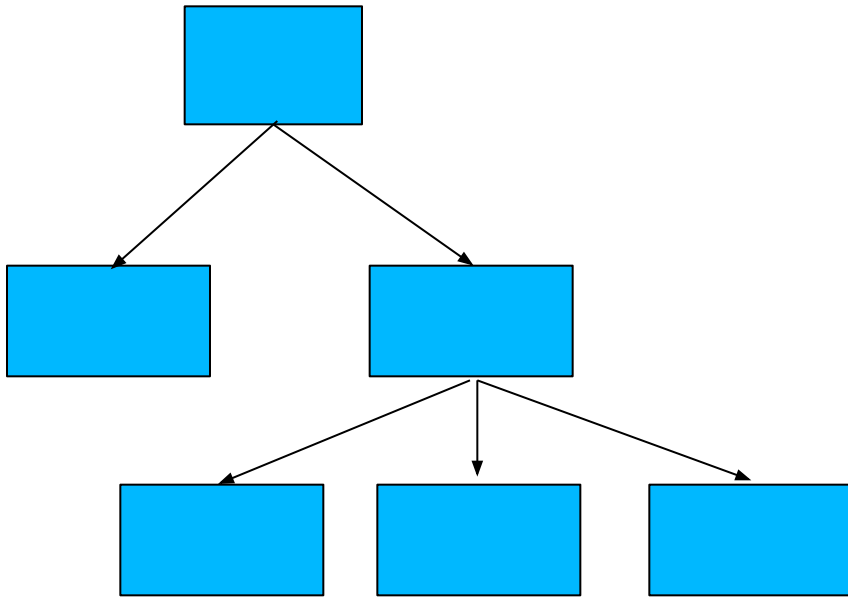
- y modules can be understood separately,

- x reduces the complexity greatly.

- y To understand why this is so,

- x 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.

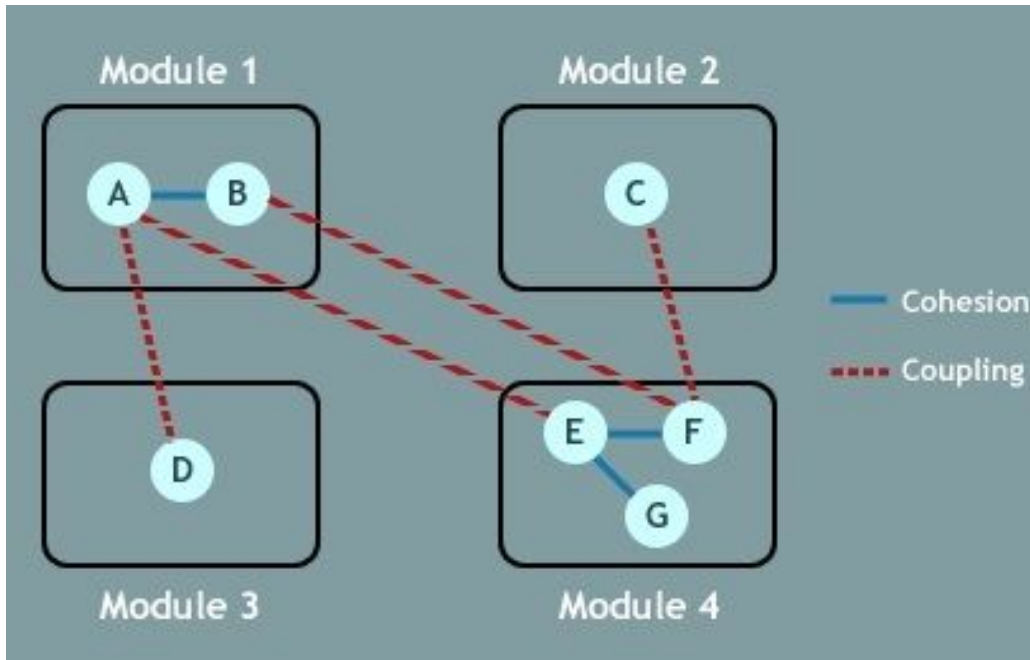
Modularity



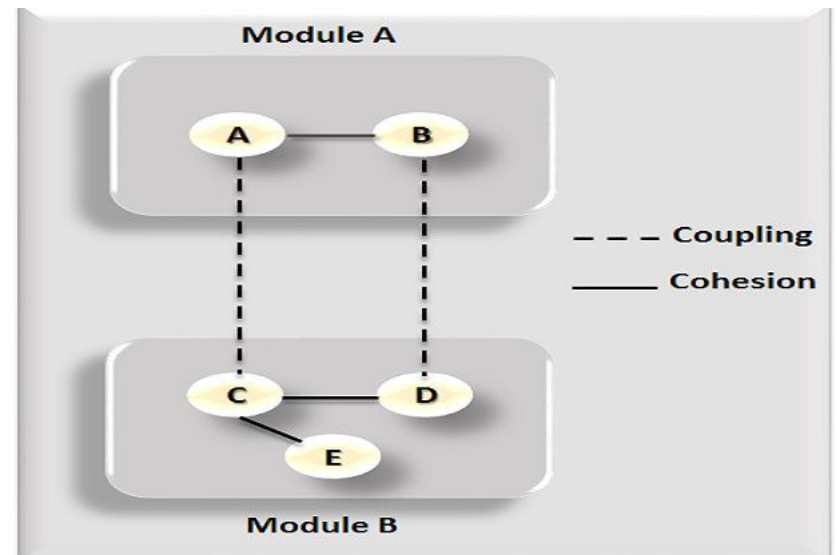
- ❖ Neat arrangement of modules in a hierarchy means:
 - ❑ low fan-out
 - ❑ abstraction

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'.

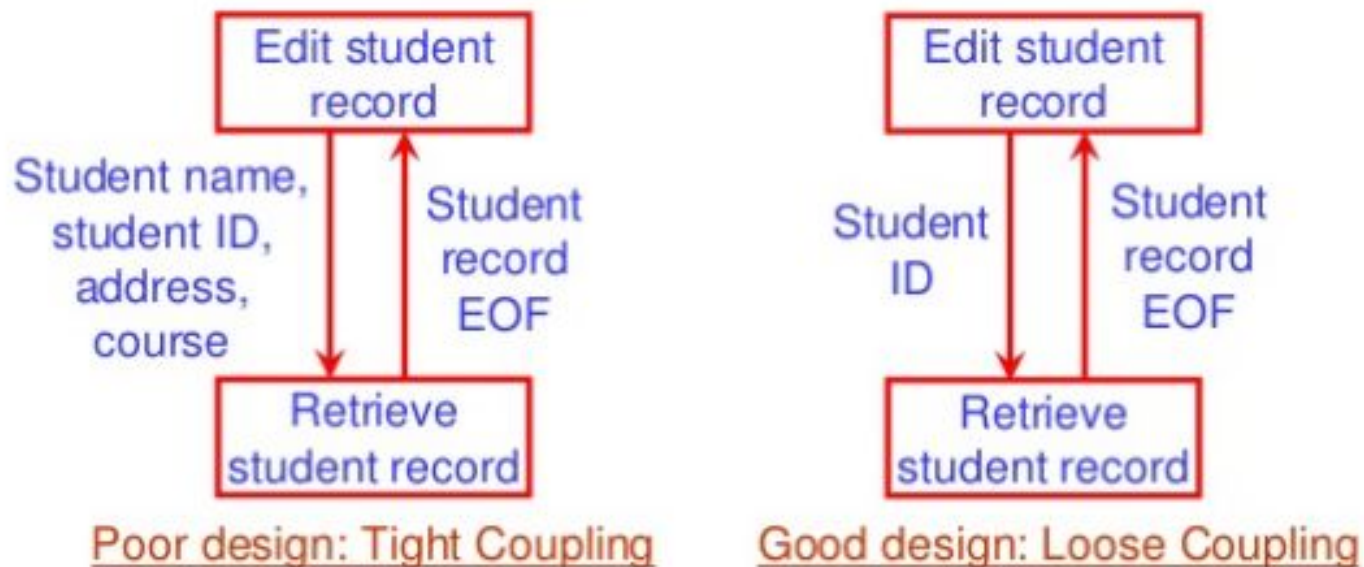


Fig. 6 : Example of coupling

Cohesion and Coupling

Ñ A module having high cohesion and low coupling:

y functionally independent of other modules:

x A functionally independent module has minimal interaction with other modules.

Advantages of Functional Independence

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

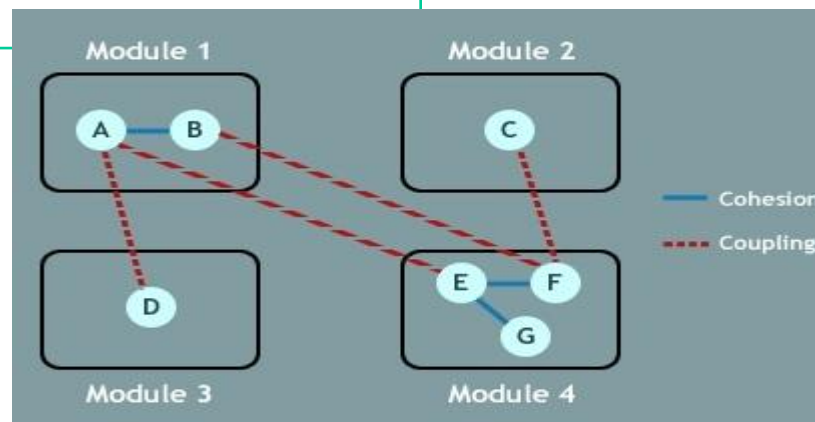
Advantages of Functional Independence

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

Advantages of Functional Independence

- Ñ A functionally independent module:
 - y can be easily taken out and reused in a different program.
 - x each module does some well-defined and precise function
 - x 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

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

Classification of Cohesiveness



functional
sequential
communicational
procedural
temporal
logical
coincidental



**Degree of
cohesion**

◦ coincidental

◦ communicational

◦ temporal

◦ logical

◦ functional

◦ procedural

◦ sequential



bad end

good end

Coincidental cohesion

Ñ The module performs a set of tasks:

y which relate to each other very loosely, if at all.

x the module contains a random collection of functions.

x functions have been put in the module out of pure coincidence without any thought or design.

x 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:
 - y e.g. error handling, data input, data output, etc.
- ~ An example of logical cohesion:
 - y a set of print functions to generate an output report arranged into a single module.

Temporal cohesion

Ñ The module contains tasks that are related by the fact:

y all the tasks must be executed in the same time span.

Ñ Example:

y The set of functions responsible for

x initialization,

x start-up, shut-down of some process, etc.

Procedural cohesion

Ñ The set of functions of the module:

- y all part of a procedure (algorithm)

- y certain sequence of steps have to be carried out in a certain order for achieving an objective,

 - x e.g. the algorithm for decoding a message.

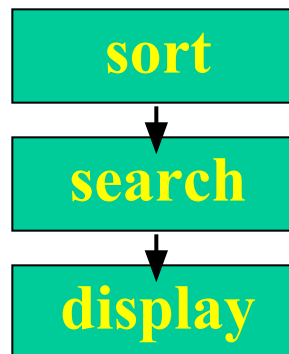
Communicational cohesion



- Ñ All functions of the module:
 - y reference or update the same data structure,
- Ñ Example:
 - y the set of functions defined on an array or a stack.

Sequential cohesion

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



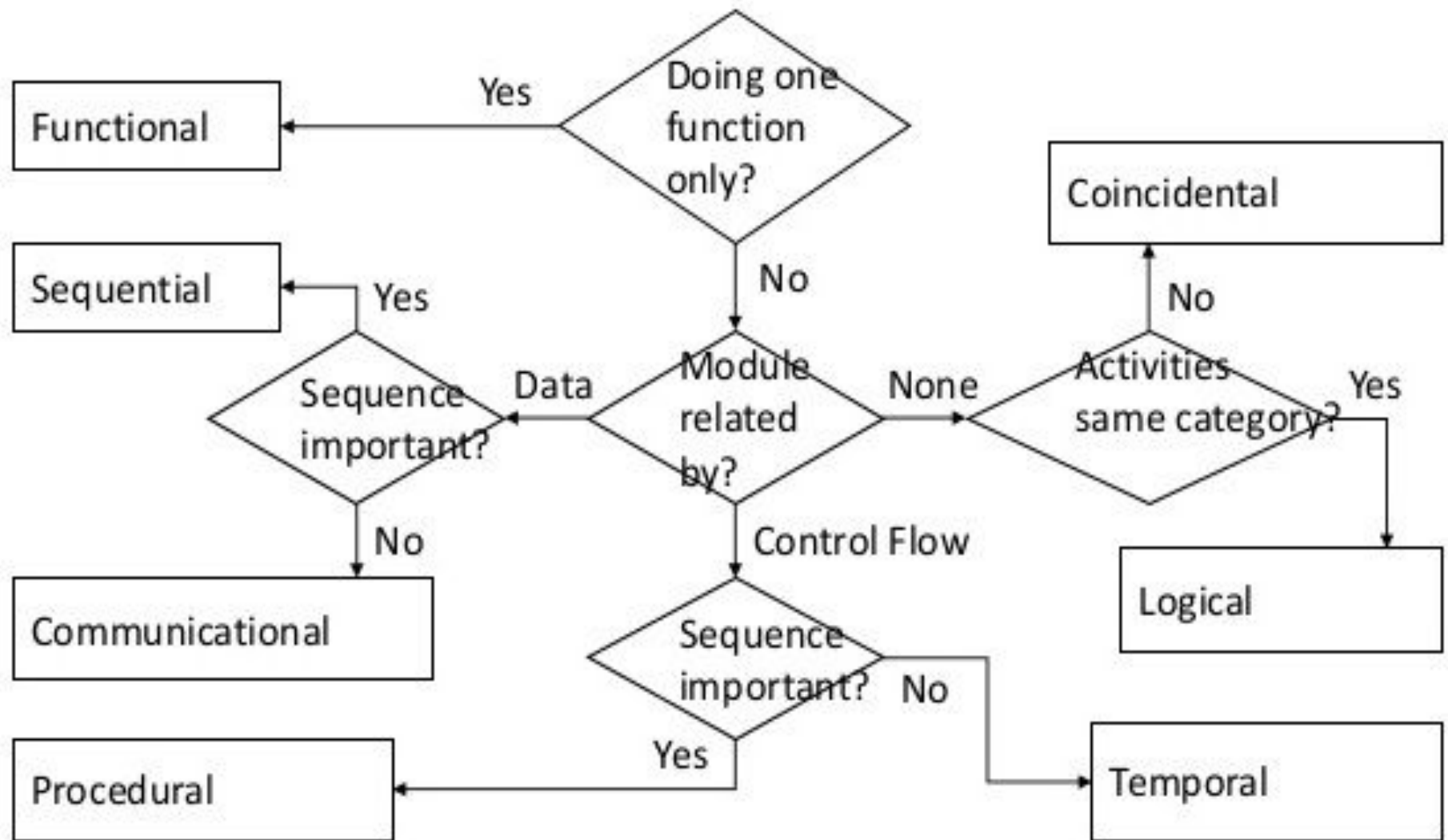
Functional cohesion

~ Different elements of a module cooperate:

y to achieve a single function,
y e.g. managing an employee's pay-roll.

~ When a module displays functional cohesion,
y we can describe the function using a single sentence.

Determining Module Cohesion



Determining Cohesiveness

- Ñ Write down a sentence to describe the function of the module
 - y If the sentence is compound,
 - x it has a sequential or communicational cohesion.
 - y If it has words like "first", "next", "after", "then", etc.
 - x it has sequential or temporal cohesion.
 - y If it has words like initialize,
 - x it probably has temporal cohesion.

Coupling



Ñ Coupling indicates:

y how closely two modules interact or how interdependent they are.

y The degree of coupling between two modules depends on their interface complexity.

Coupling



- ~ There are no ways to precisely determine coupling between two modules:
 - y 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
stamp
control
common
content

**Degree of
coupling**



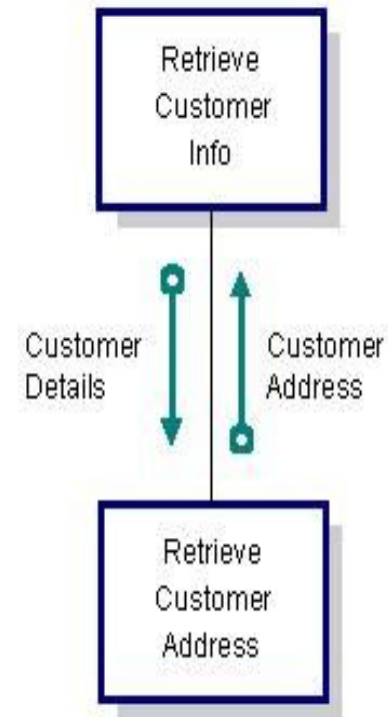
Data coupling

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

Stamp coupling

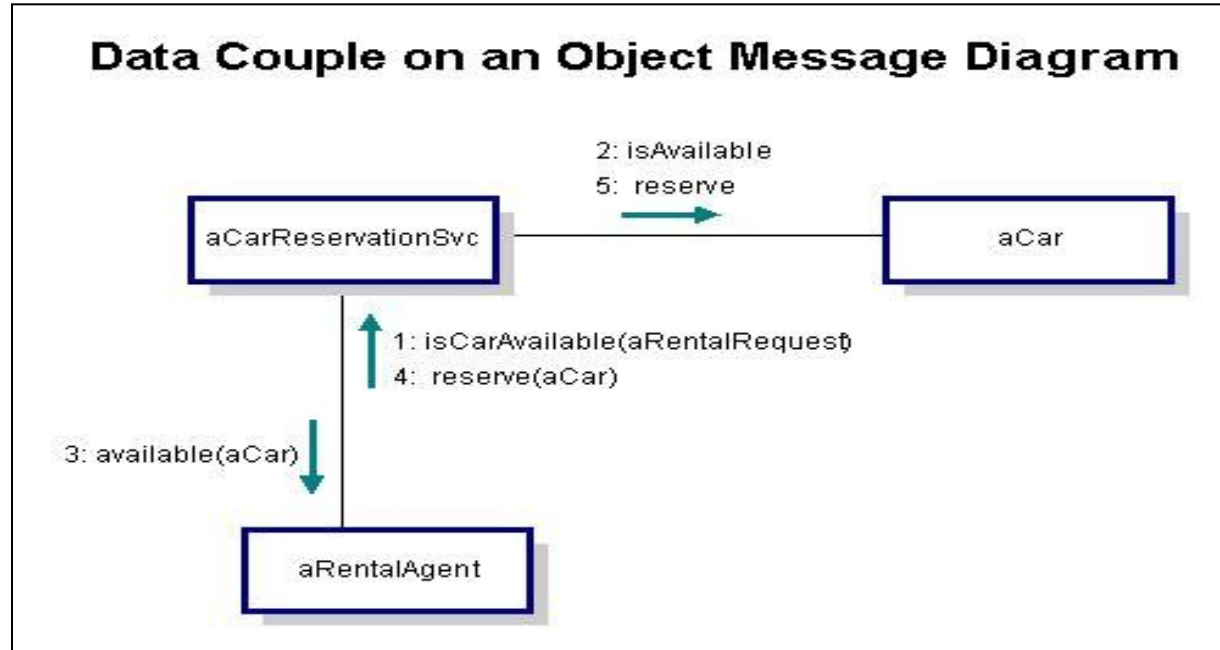
- ~ Two modules are stamp coupled,
 - y if they communicate via a composite data item
 - x such as a record in PASCAL
 - x or a structure in C.

STAMP COUPLE



Control coupling

- Ñ Data from one module is used to direct order of instruction execution in another.
- Ñ Example of control coupling:
 - y a flag set in one module and tested in another module.

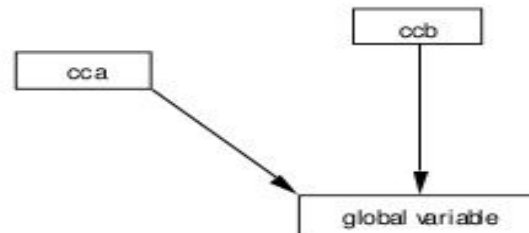


Common Coupling

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

Example of Common Coupling

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

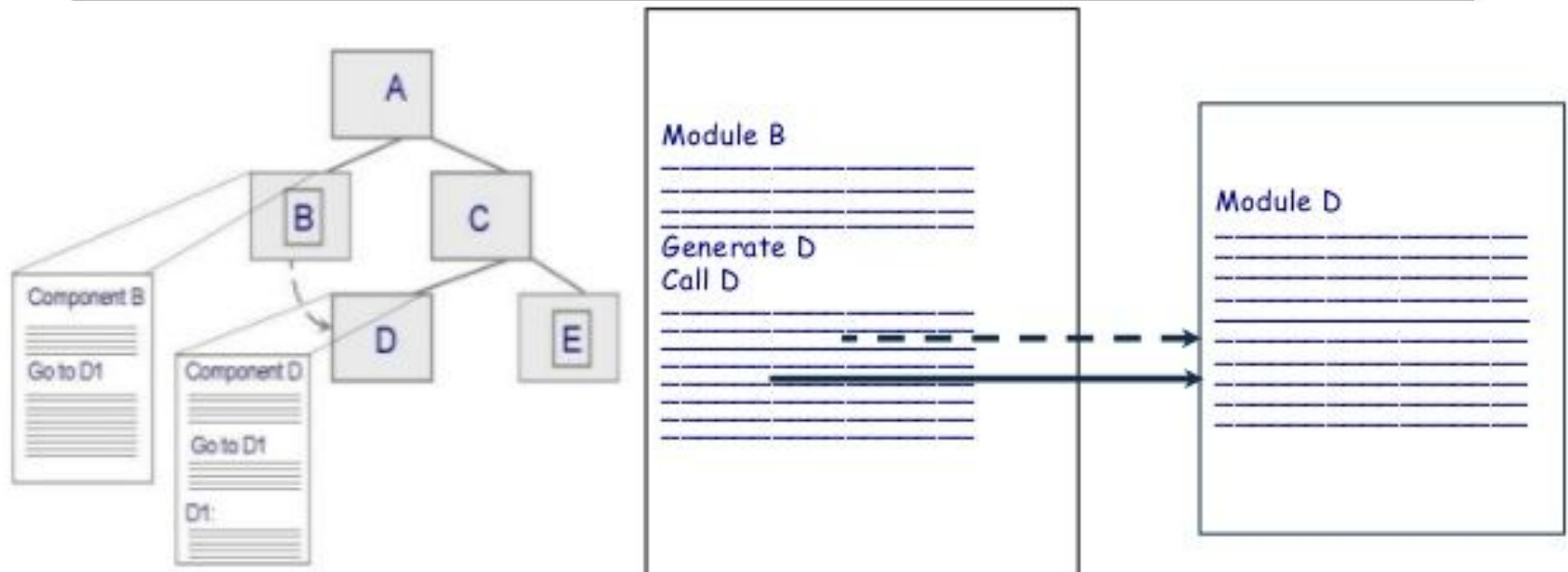


Content coupling

- Ñ Content coupling exists between two modules:
 - y if they share code,
 - y e.g, branching from one module into another module.
- Ñ The degree of coupling increases
 - y 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

Ñ Control hierarchy represents:

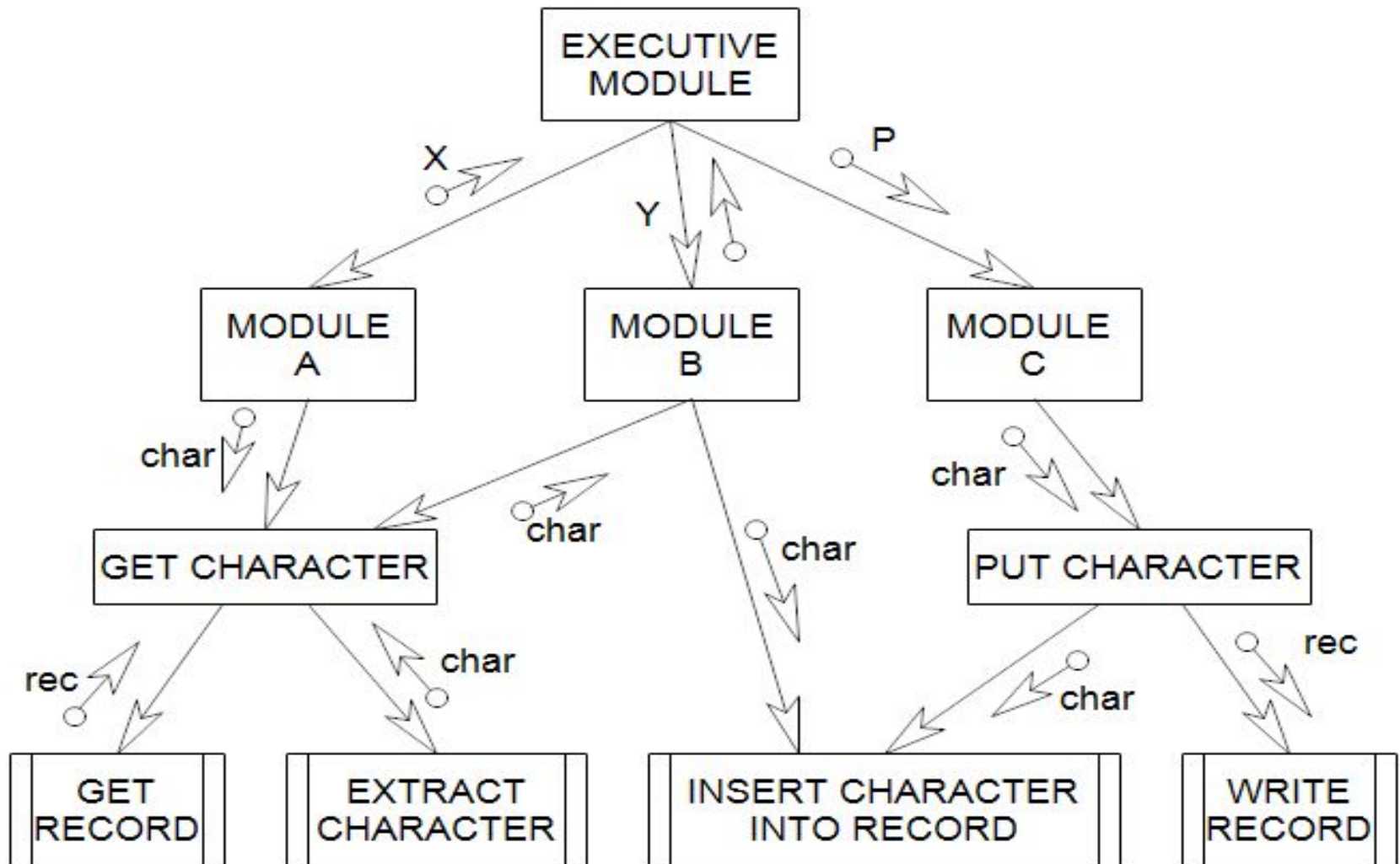
y organization of modules.

y control hierarchy is also called program structure.

Ñ Most common notation:

y a tree-like diagram called structure chart.

Structure Chart



Neat Arrangement of modules



Ñ Essentially means:

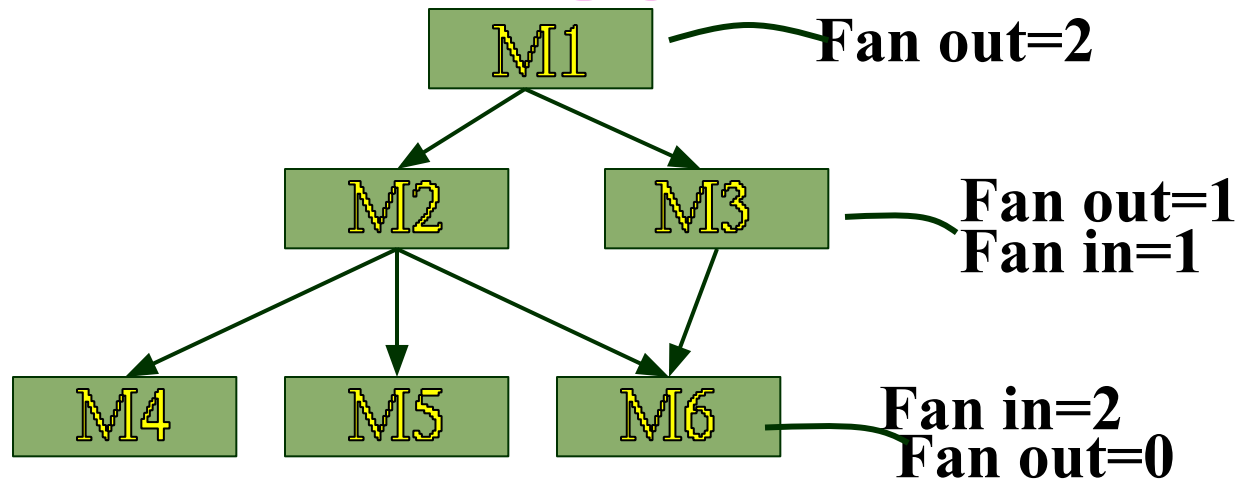
y low fan-out

y abstraction

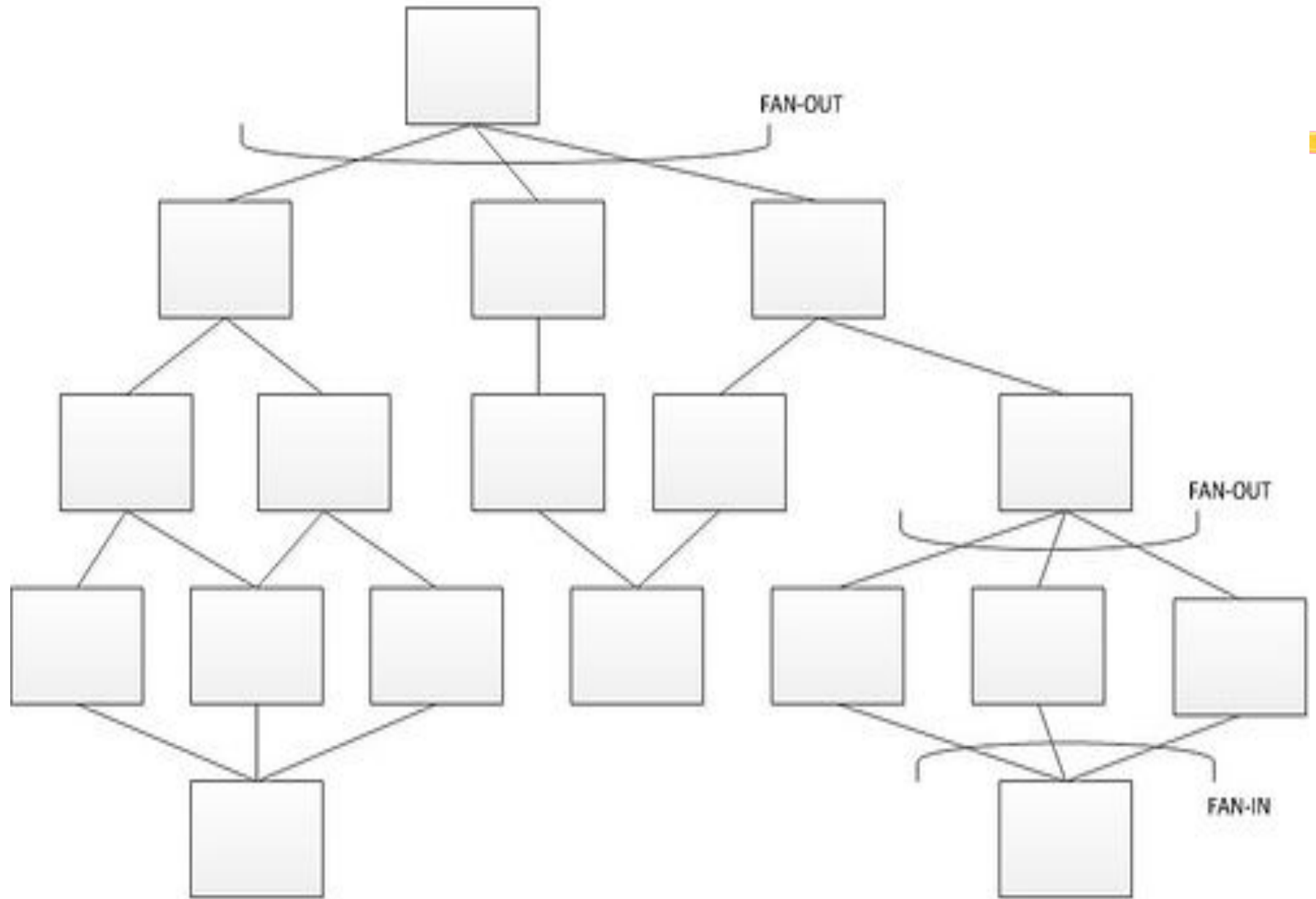
Characteristics of Module Structure

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

Module Structure



Fan-Out/Fan-In



Goodness of Design

Ñ A design having modules:

- y with **high fan-out numbers is not a good design:**

- y a module having **high fan-out lacks cohesion.**

Ñ A module that invokes a large number of other modules:

- y likely to implement several different functions:

- y not likely to perform a single cohesive function.

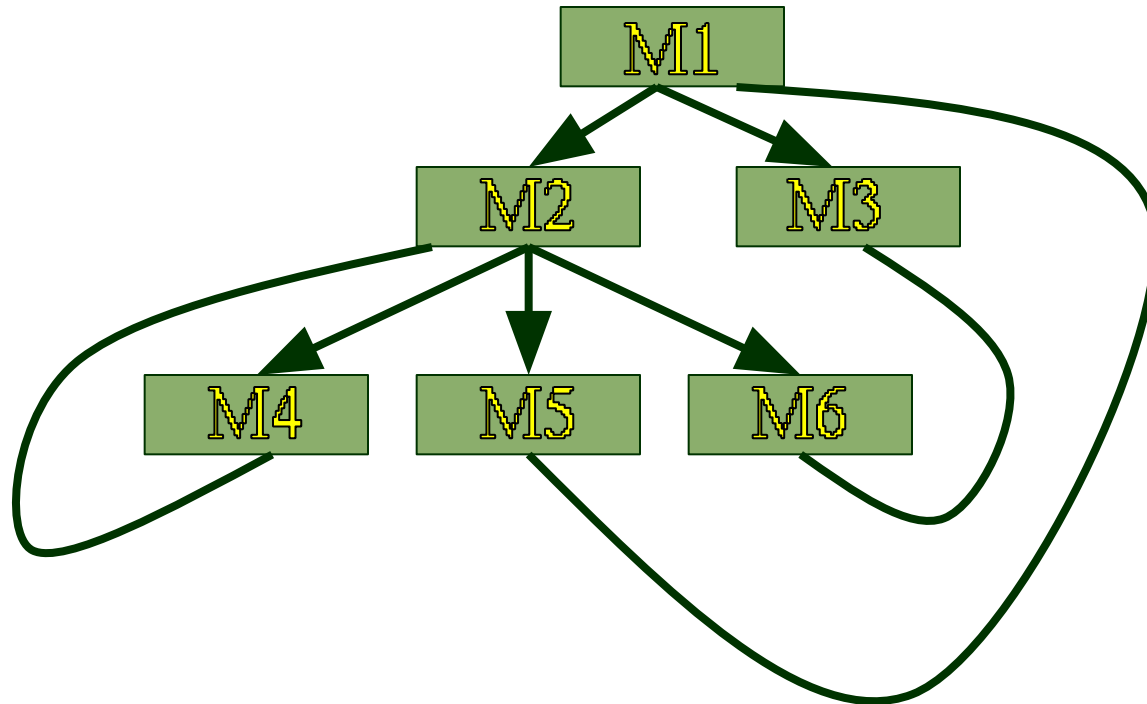
Control Relationships

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

Visibility and Layering

- Ñ A module A is said to be visible by another module B,
 - y if A directly or indirectly calls B (Embedding).
- Ñ The layering principle requires
 - y modules at a layer can call only the modules immediately below it (Sequence).


Bad Design



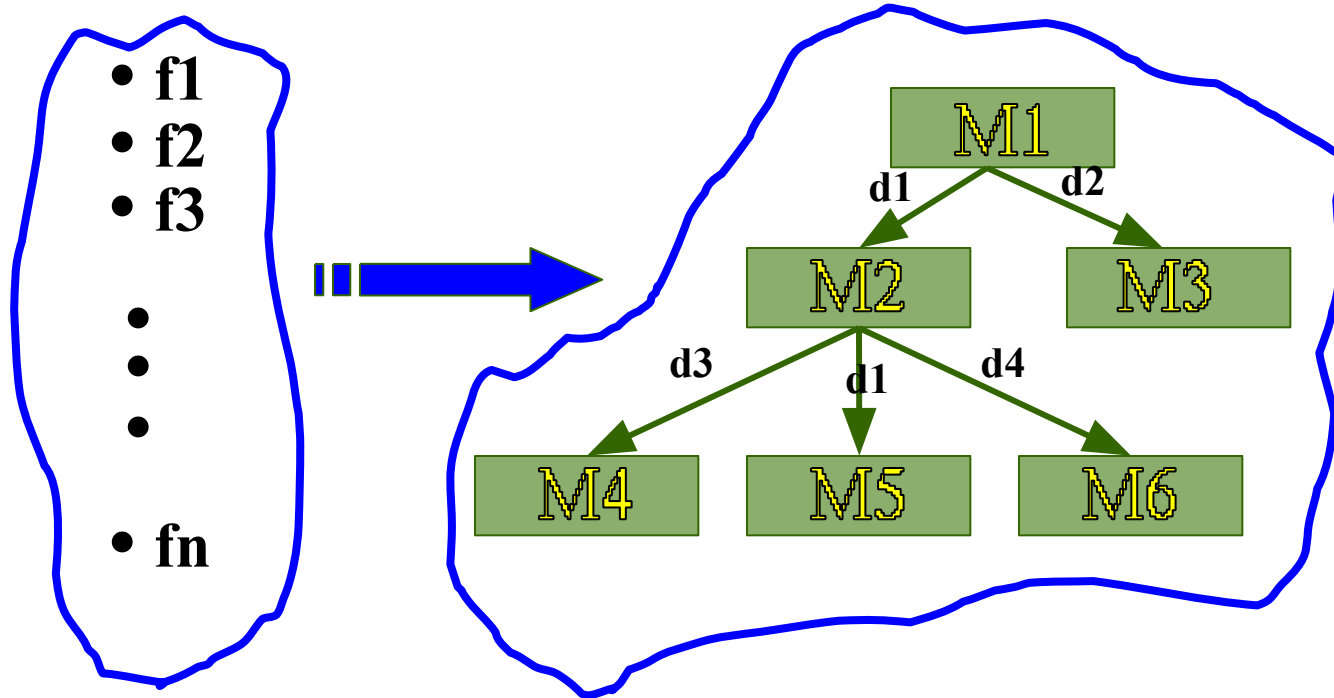
Abstraction

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

High-level Design

- 
- ~ High-level design maps functions into modules $\{f_i\}$ $\{m_j\}$ such that:
- y Each module has high cohesion
 - y Coupling among modules is as low as possible
 - y Modules are organized in a neat hierarchy

High-level Design



Design Approaches

Ñ Two fundamentally different software design approaches:

- y Function-oriented design

- y Object-oriented design

Ñ These two design approaches are radically different.

- y However, are complementary

 - x rather than competing techniques.

- y Each technique is applicable at

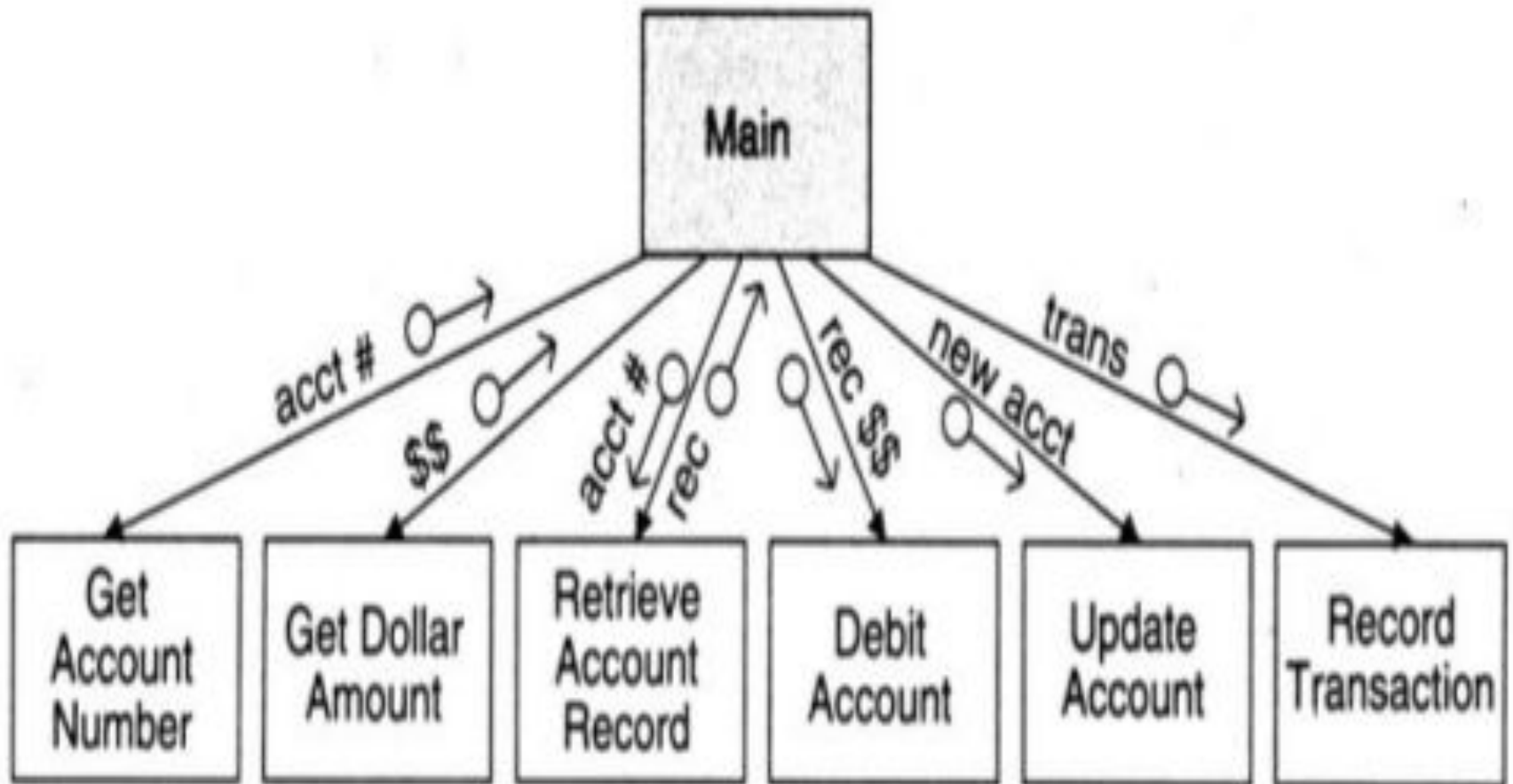
 - x different stages of the design process.

Function-Oriented Design



- Ñ A system is looked upon as something
 - y that performs a set of functions.
- Ñ Starting at this high-level view of the system:
 - y each function is successively refined into more detailed functions.
 - y Functions are mapped to a module structure.

ATM FUNCTION DESIGN



Example



Ñ The function `create-new-library-member`:

- y creates the record for a new member,

- y assigns a unique membership number

- y prints a bill towards the membership

Example



Ñ Create-library-member function consists of the following sub-functions:

- y assign-membership-number

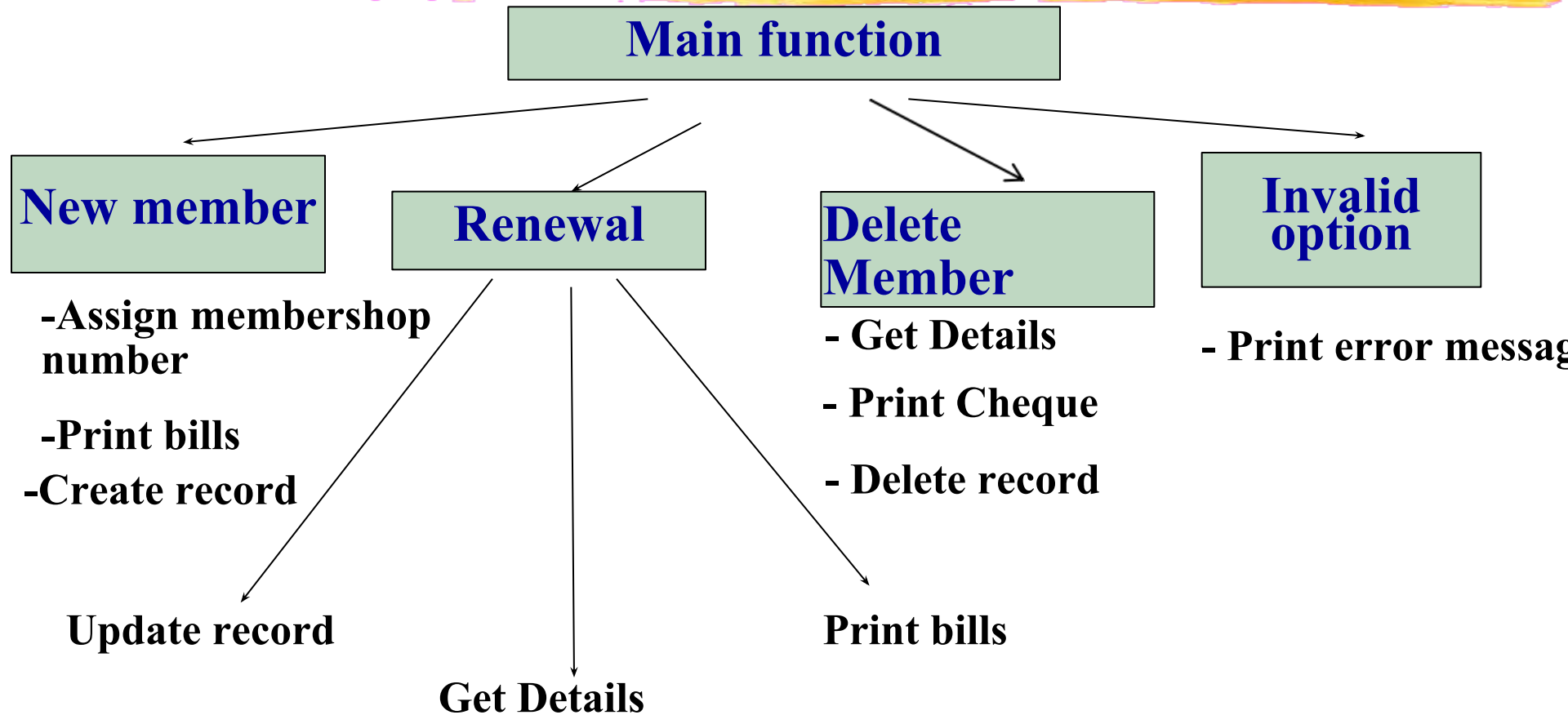
- y create-member-record

- y print-bill

Function-Oriented Design

- Ñ Each subfunction:
 - y split into more detailed subfunctions and so on.
- Ñ The system state is centralized:
 - y accessible to different functions,
 - y member-records:
 - x available for reference and updation to several functions:
 - create-new-member
 - delete-member
 - Renew-member-record

FUNCTION DESIGN



Function-Oriented Design

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

Object-Oriented Design

- Ñ System is viewed as a collection of objects (i.e. entities).
- Ñ System state is decentralized among the objects:
 - y each object manages its own state information.

Object-Oriented Design

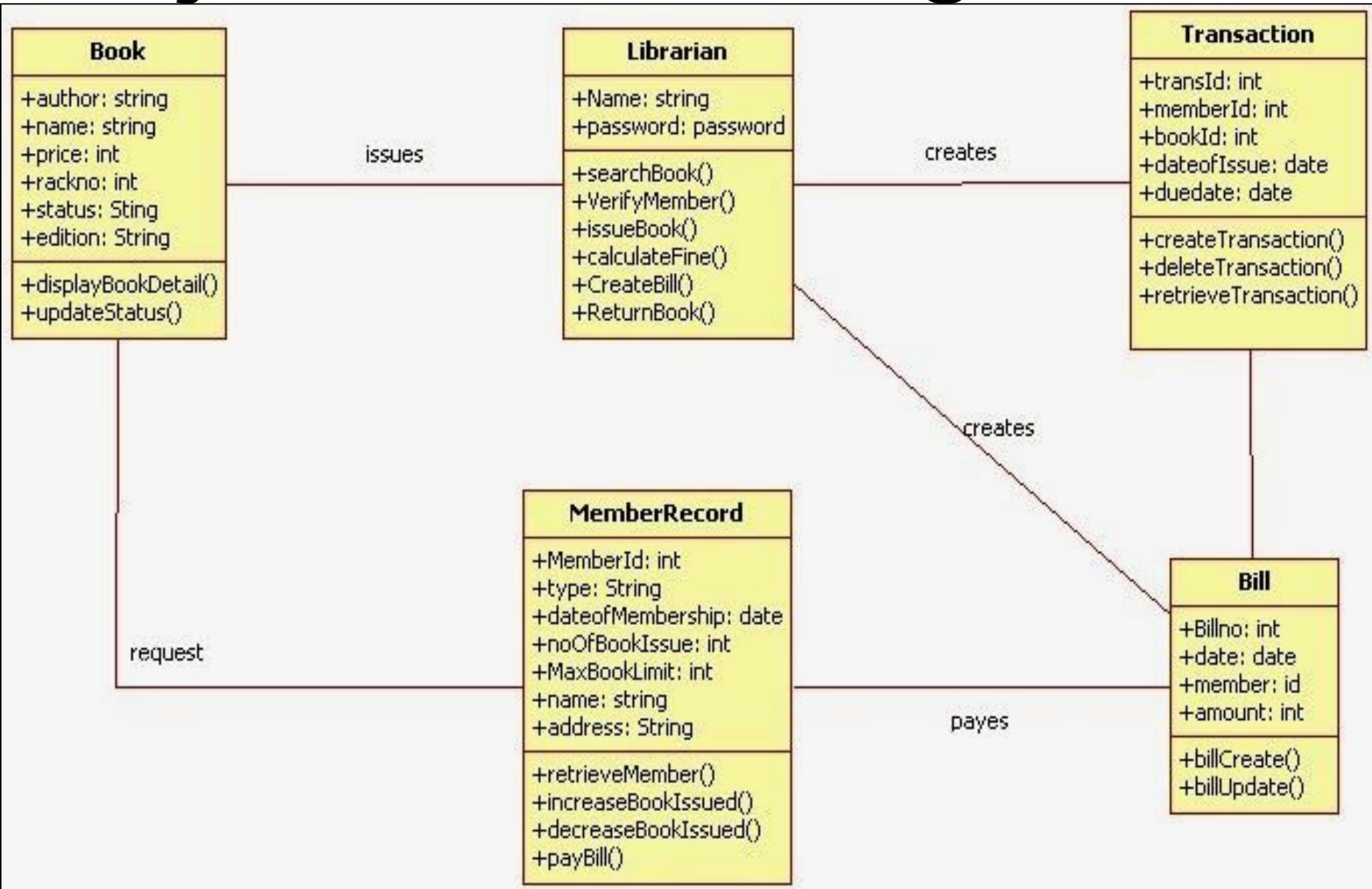
Example

- ~ Library Automation Software:
 - y each library member is a separate object
 - x with its own data and functions.
- y Functions defined for one object:
 - x cannot directly refer to or change data of other objects.

Object-Oriented Design

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

Object Oriented Design



Object-Oriented versus Function-Oriented Design

- Ñ Unlike function-oriented design,
 - y in OOD the basic abstraction is not functions such as “sort”, “display”, “track”, etc.,
 - y but real-world entities such as “employee”, “picture”, “machine”, “radar system”, etc.
- Ñ In OOD:
 - y software is not developed by designing functions such as:
 - x update-employee-record,
 - x get-employee-address, etc.
 - y but by designing objects such as:
 - x employees,
 - x departments, etc.

Object-Oriented versus Function-Oriented Design

Ñ Grady Booch sums up this fundamental difference saying:

y “Identify verbs if you are after procedural design and nouns if you are after object-oriented design.”

Ñ In OOD:

y state information is not shared in a centralized data.

y but is distributed among the objects of the system.

Example:

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

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

Object-Oriented versus Function-Oriented Design

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

Fire-Alarm System:

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

Ñ **/* 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 */

Ñ **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:

- Ñ class detector
 - attributes: status, location, neighbors
 - operations: create, sense-status, get-location, find-neighbors
- Ñ class alarm
 - attributes: location, status
 - operations: create, ring-alarm, get_location, reset-alarm
- Ñ 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:
 - y activities undertaken during the software design phase.
- ~ We identified:
 - y the information need to be produced at the end of the design phase:
 - x so that the design can be easily implemented using a programming language.

Summary



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

Summary



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

Summary



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

Summary



Ñ We looked at the essential philosophy behind these two approaches

y these two approaches are not competing but complementary approaches.