3

Software Design

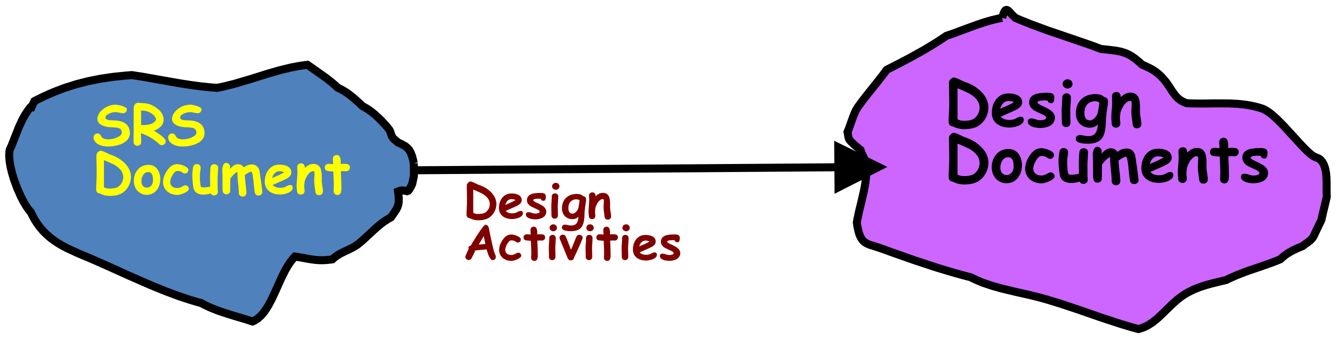
Main Objectives Of The Design Phase

- During the software design phase, the <u>design document </u>is produced, based on the customer requirements as documented in the SRS document.

- The activities carried out during the design phase (called as design process <u>) transform the SRS document into the design document.</u>

What is Achieved during design phase?

- Transformation of SRS document to Design document:
  - A form easily implementable in some programming language.

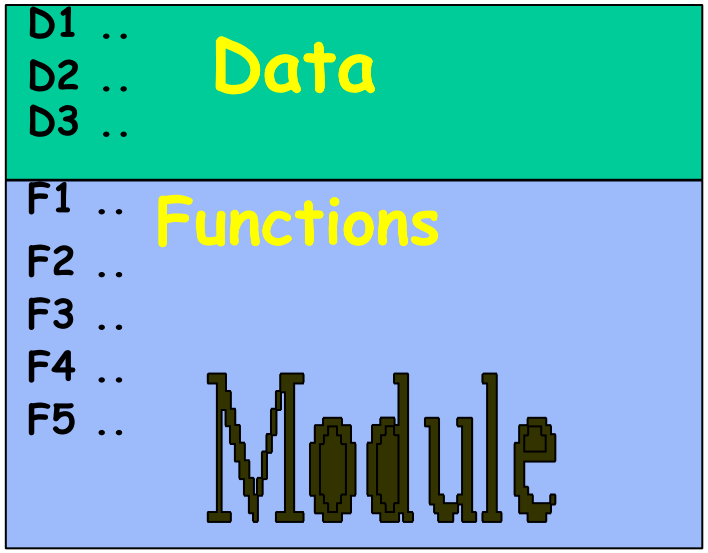


Outcome of the Design Process- Items Designed During Design Phase

- 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

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

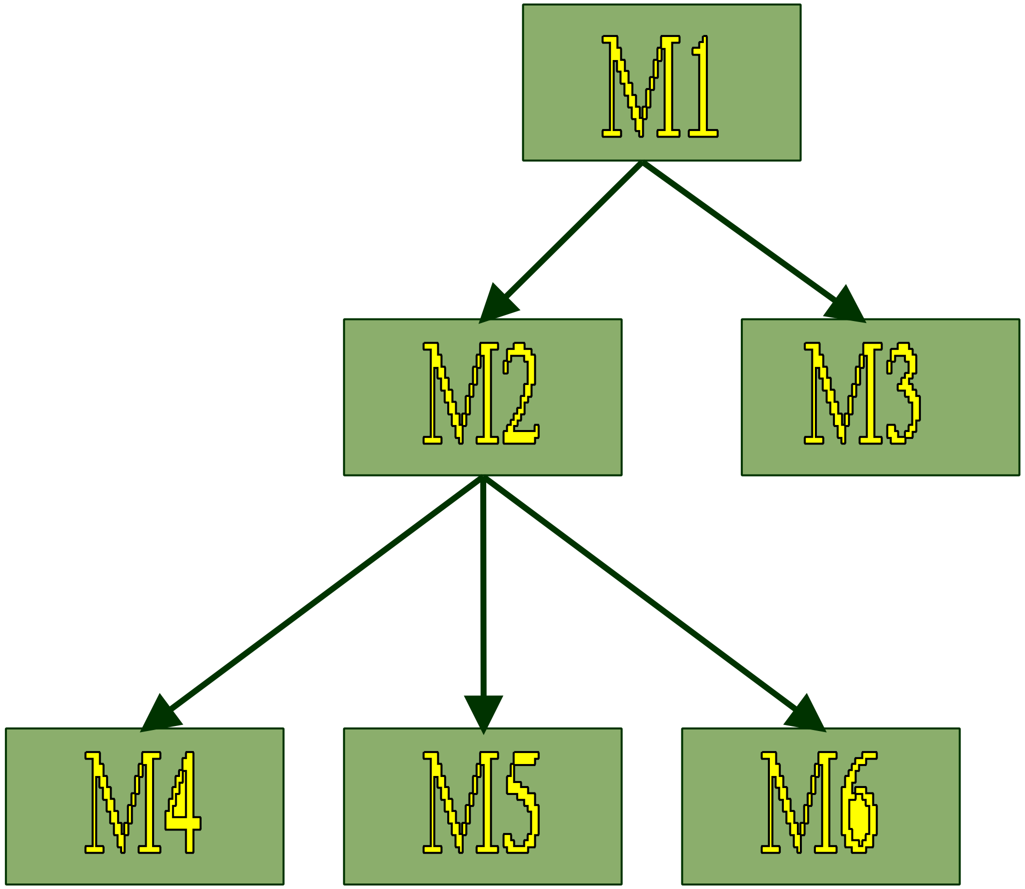


Module Structure

The different modules in the solution should be clearly identified.

Each module should accomplish some well-defined task out of the overall responsibility of the software.

Each module should be named according to the task it performs.



Control relationships among modules:

A control relationship between two modules essentially arises due to function calls across the two modules.

The control relationships existing among various modules should be identified in the design document.

Interfaces among different modules:

The interfaces between two modules identifies the exact data items that are exchanged between the two modules when one module invokes a function of the other module.

Data structures of the individual modules:

Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module.

Suitable data structures for storing and managing the data of a module need to be properly designed and documented.

Algorithms required to implement the individual modules:

Each function in a module usually performs some processing activity.

The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given
to the accuracy of the results, space and time complexities.

Iterative Nature of Design

- Good software designs:
  - Seldom arrived through a single step procedure:
  - But through a series of steps and iterations.

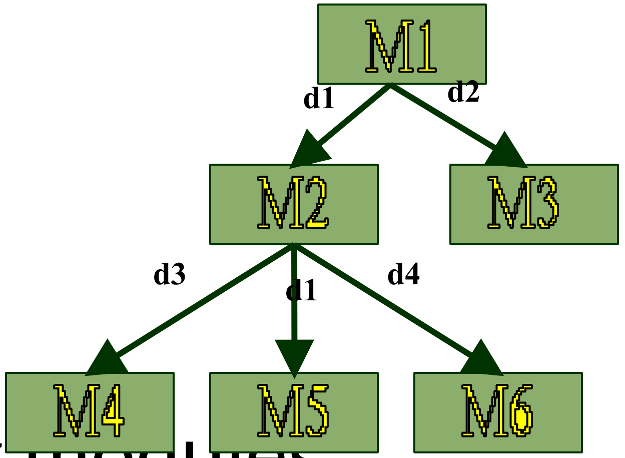
Classification of Design Activities

Stages in Design

- Design activities are usually classified into two stages:
  - **Preliminary (or high-level) design**
  - **Detailed design**

High-level design: Software Architecture

- Identify:
  - modules
  - control relationships among modules
  - interfaces among modules.



High-level design: Software Architecture

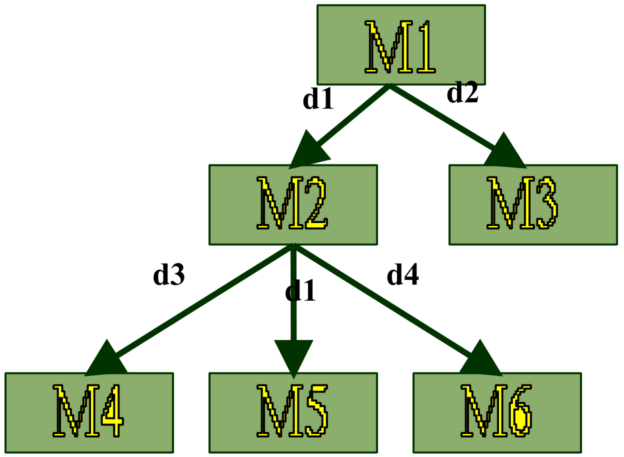
- The outcome of high-level design:
  - program structure, also called software architecture.

- High-level design is a crucial step in the overall design of a software.

- When the high-level design is complete:
  - <u>The problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among themselves, and are arranged in a hierarchy.</u>

High-level Design

- Several notations are available to represent high-level design:
  - A notation used for procedural development is a tree-like diagram called the **structure chart**.
  - Another design representation technique called UML that is used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems.
  - Other notations:
    - Jackson diagram or Warnier-Orr diagram can also be used.



Detailed design

- During detailed design each module is examined carefully to design its data structures and the algorithms.
- For each module, design for it:
  – data structure
  – algorithms

Outcome of detailed design:

- module specification.

- The outcome of the detailed design stage is usually documented in the form of a module specification (MSPEC) document.
- The data structures and algorithms to be used described using MSPEC and can be easily grasped by programmers for initiating coding.

A fundamental question

- How to distinguish between good and bad designs?
  - Unless we know what a good software design is:
    - we can not possibly design one.

Good and bad designs

- There is no unique way to design a software.
- Even while using the same design methodology:
  – different engineers can arrive at very different designs.
- Need to determine which is a better design.

What Is a 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: A Major Concern

- Understandability of a design is a major issue:
  - Largely determines goodness of a design:
  - a design that is easy to understand:
    - also easy to maintain and change.

Understandability of a Design: A Major Concern

- Unless a design is easy to understand,
  - Tremendous effort needed to maintain it
  - We already know that about 60% effort is spent in maintenance.
- If the software is not easy to understand:
  - maintenance effort would increase many times.

How to Improve Understandability?

- Use consistent and meaningful names
  - for various design components,
- It should make use of the principles of decomposition and abstraction in good measures to simplify the design.
- Design solution should consist of:
  - A set of well decomposed modules **(modularity)**,
- Different modules should be neatly arranged in a hierarchy:
  - A tree-like diagram.
  - Called Layering
- A design solution is understandable, if it is modular and the modules are arranged in distinct layers.

Modularity

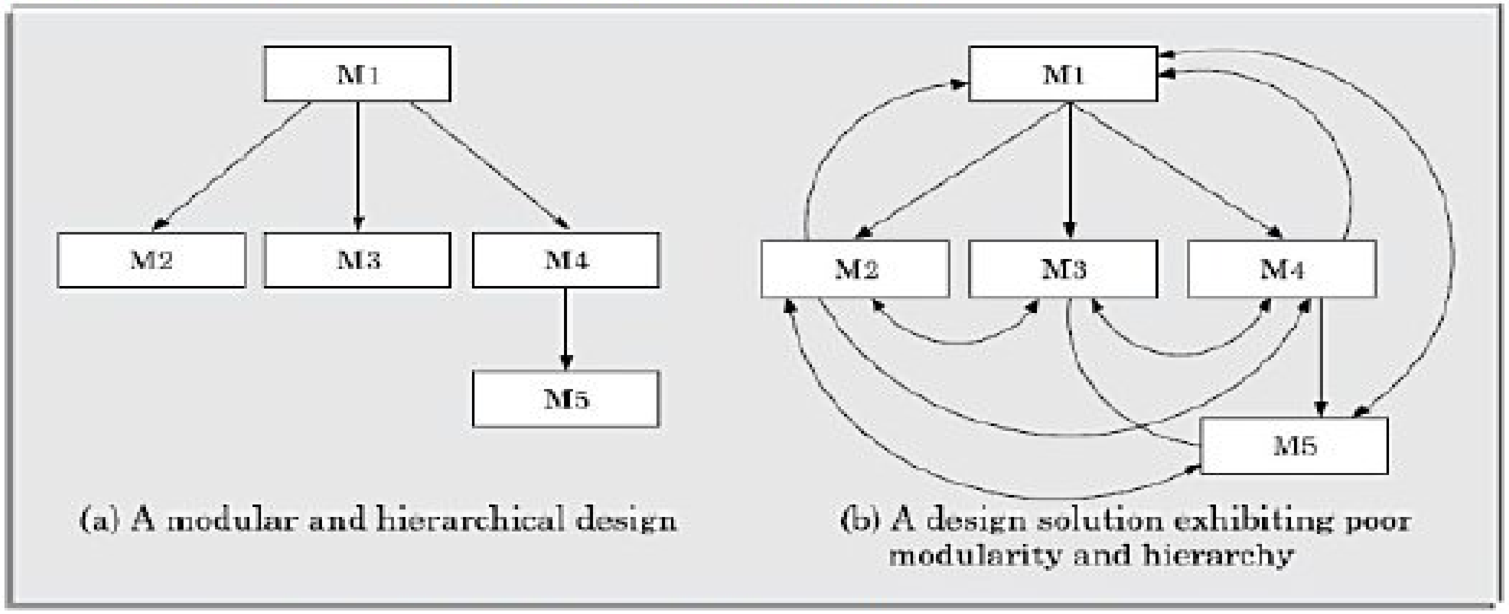
- Modularity is a fundamental attributes of any good design.
  - Decomposition of a problem into a clean set of modules:
  - Modules are almost independent of each other
  - Based on **divide and conquer principle.**

Modularity

- If modules are independent:
  - Each module can be understood separately,
    - reduces 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.

Comparison of Modularity Of Two Alternate Design Solutions?

- Consider 2 alternate design solutions to a problem that are represented in Figure.
- Modules M1 , M2 etc. have been drawn as rectangles. The invocation of a module by another module has been shown as an arrow.



(a) A modular and hierarchical design

(b) A design solution exhibiting poor modularity and hierarchy

Comparison of Modularity Of Two Alternate Design Solutions?

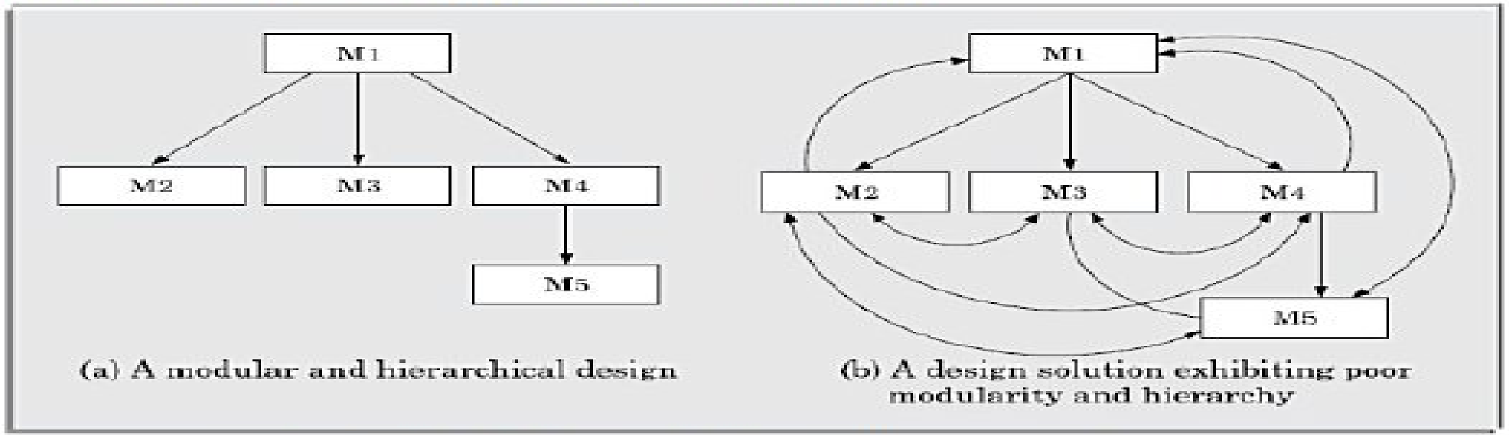
- A software design with high cohesion and low coupling among modules is the effective problem decomposition .

- It increases productivity during program development by bringing down the perceived problem complexity.

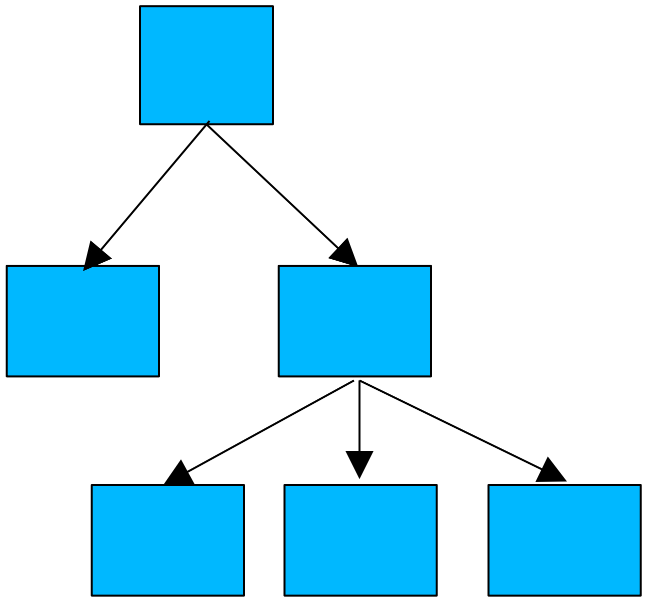
- A design solution is said to be <u>**highly modular**</u>, if the different modules in the solution have high cohesion and their inter-module couplings are low.

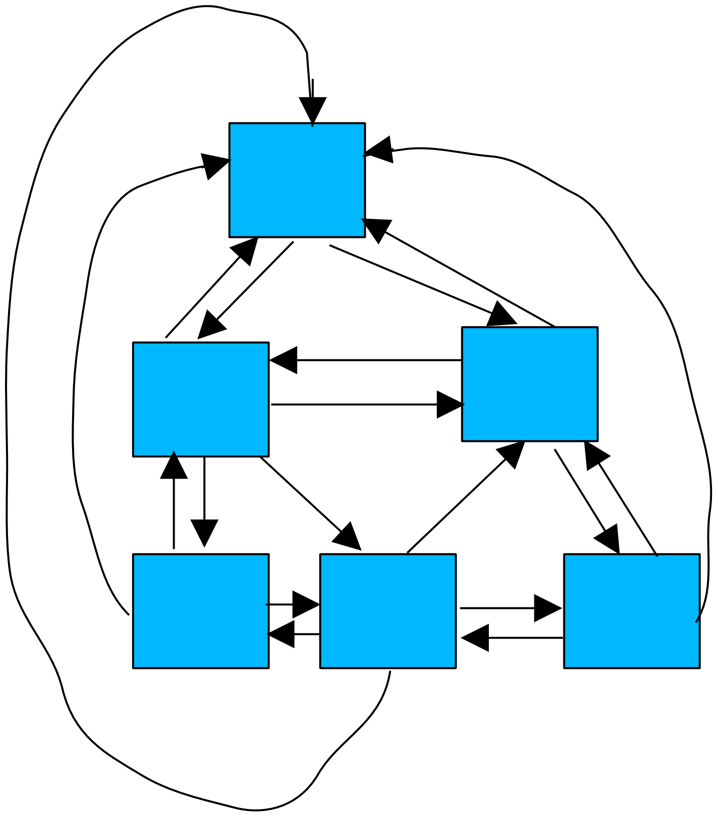
Comparison of Modularity Of Two Alternate Design Solutions?

- The design solution of Figure (a) would be easier to understand since the interactions among the different modules is low.


M1
M2
M3
M4
M5
(a) A modular and hierarchical design
(b) A design solution exhibiting poor modularity and hierarchy

Layering





Layering

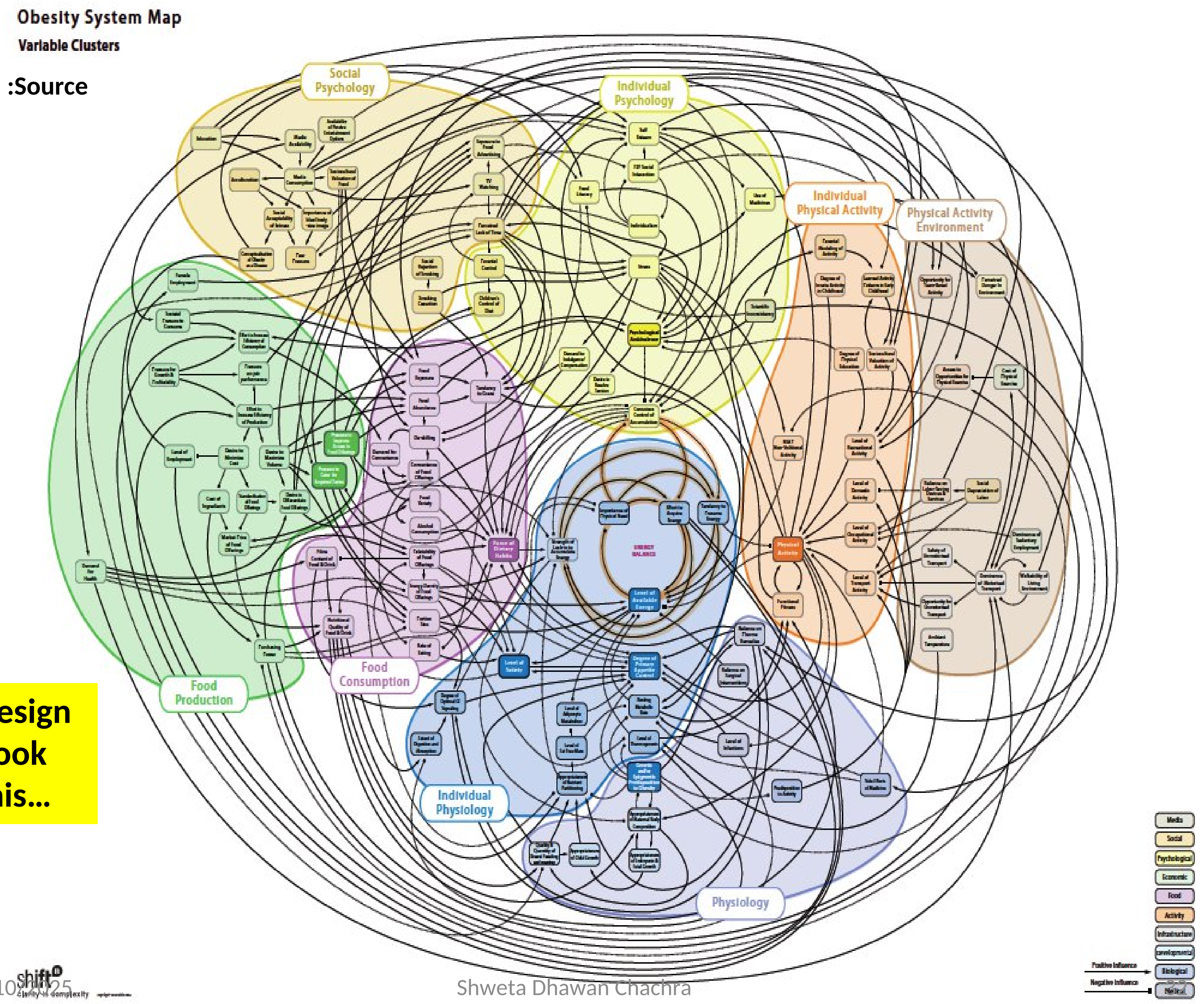
-call relations among different modules are represented graphically, resulting in a tree-like diagram with clear layering.

- In a layered design solution, the modules are arranged in a hierarchy of layers.

- A module can only invoke functions of the modules in the layer immediately below it.

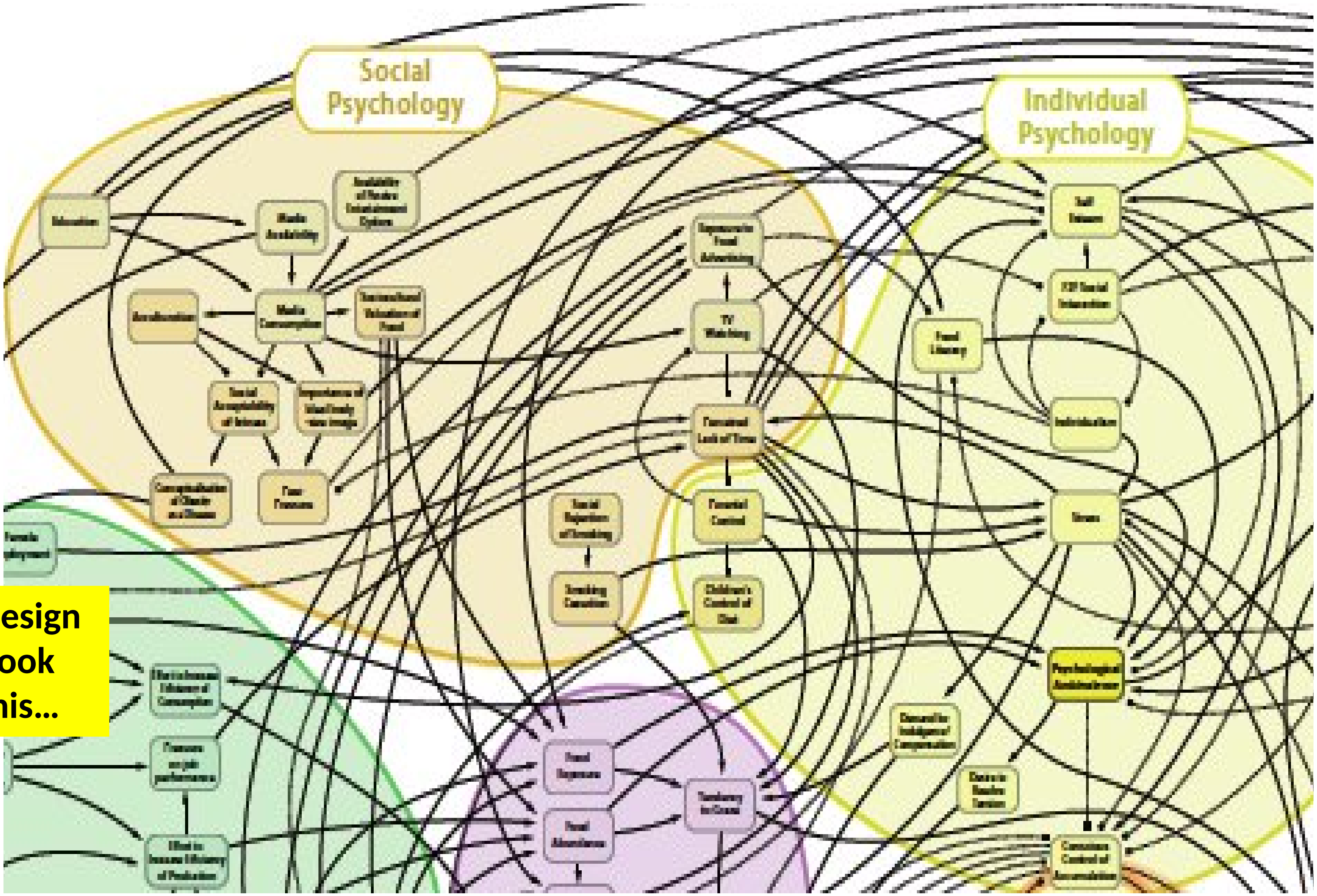
- The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done.

:Source



Bad design may look like this…

:Source



Bad design may look like this…

Layering

- Makes the design solution easily understandable, since to understand the working of a module:
  - ➢ one has to understand how the immediately lower layer modules work without having to worry about the functioning of the upper layer modules.

- When a failure is detected while executing a module, it is obvious that :
  - ➢ The modules below it can possibly be the source of the error.
  - ➢ This greatly simplifies debugging since one would need to concentrate only on a few modules to detect the error.

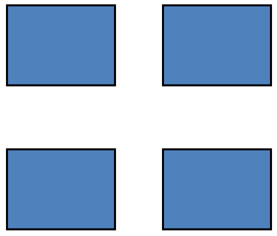
Layering

- A layered design can be considered to be implementing control abstraction, since a module at a lower layer is unaware of (about how to call) the higher layer modules.

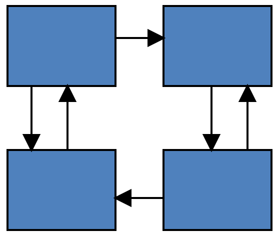
Modularity

- In technical terms, modules should display:
  - **high cohesion**
  - **low coupling.**
- We next discuss:
  - cohesion and coupling.

Coupling: Degree of dependence among components

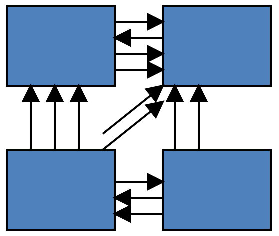


No dependencies



Loosely coupled-some dependencies

High coupling makes modifying parts of the system difficult, e.g., modifying a component affects all the components to which the component is connected.



Highly coupled-many dependencies

Source:
Pfleeger, S., *Software Engineering Theory and Practice. Prentice Hall, 2001.*

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.**

Analogy

- Suppose you listened to a talk by some speaker.
- You would call the speech to be cohesive, if all the sentences of the speech played some role in giving the talk a single and focused theme.

- When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion.

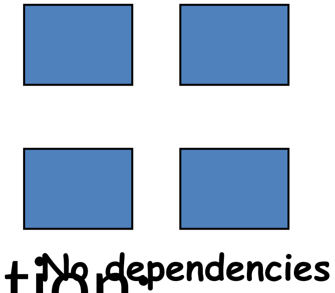
- If the functions of the module do very different things and do not co-operate with each other to perform a single piece of work, then the module has very poor cohesion.

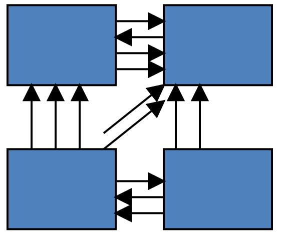
Cohesion and Coupling

- A module having **high cohesion and low coupling**:
  - **Called functionally independent** of other modules:
    - A functionally independent module needs very little help from other modules and therefore has minimal interaction with other modules.

Advantages of Functional Independence

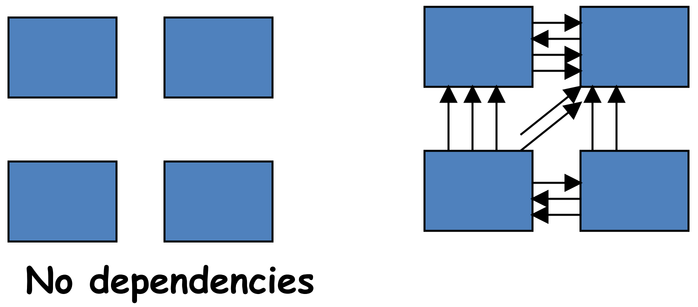
- Better understandability
- Complexity of design is reduced,
- Different modules easily understood in isolation:
  - Modules are independent




Highly coupled-many dependencies

Why Functional Independence is Advantageous?

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



Reuse: An Advantage 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.

Measuring Functional Independence

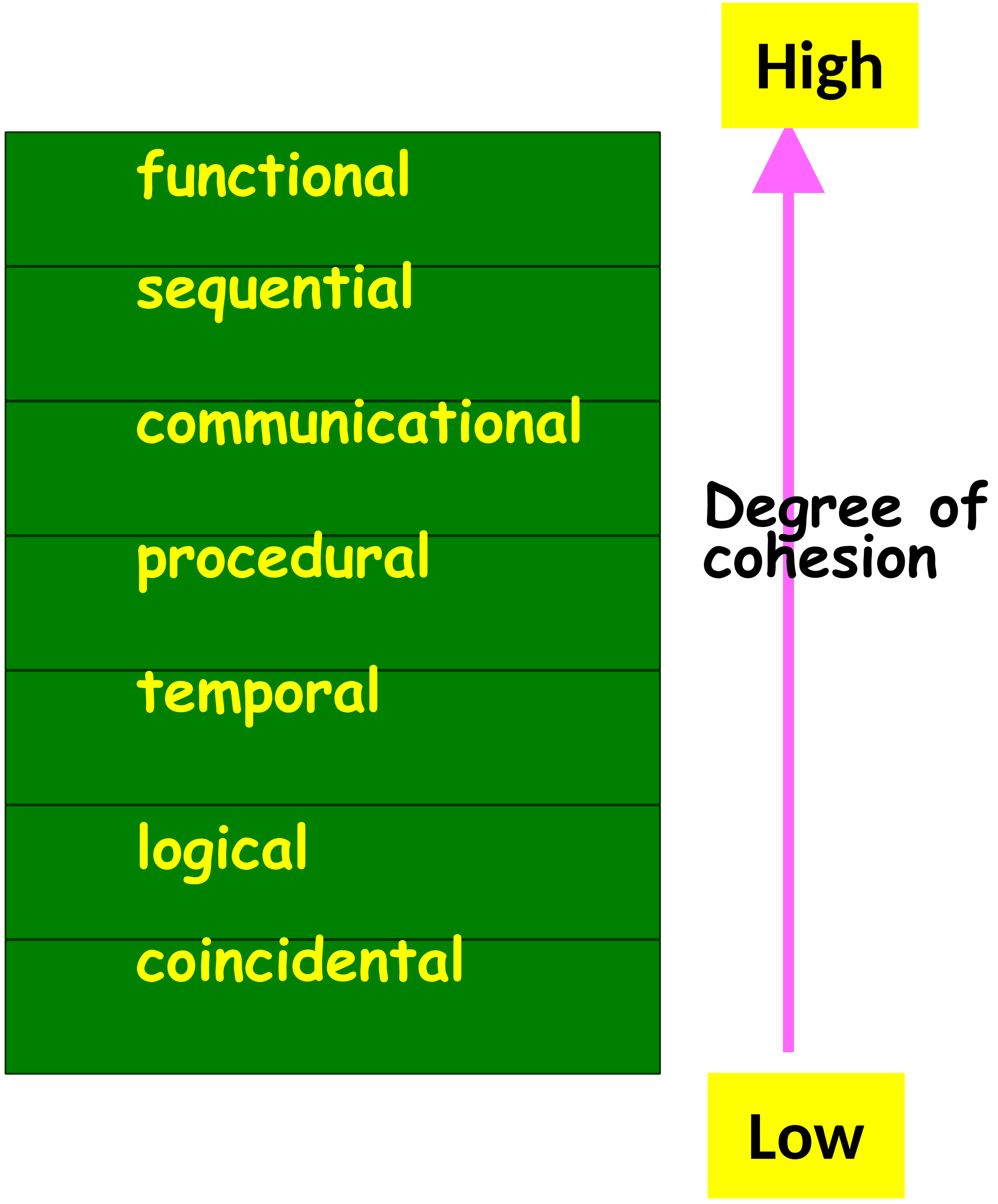
- Unfortunately, there are no ways:
  - to quantitatively measure the degree of cohesion and coupling:
  - At least classification of different kinds of cohesion and coupling:
    - will give us some idea regarding the degree of cohesiveness of a module.

Classification of Cohesiveness

- Classification can have scope for ambiguity:
  - 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.

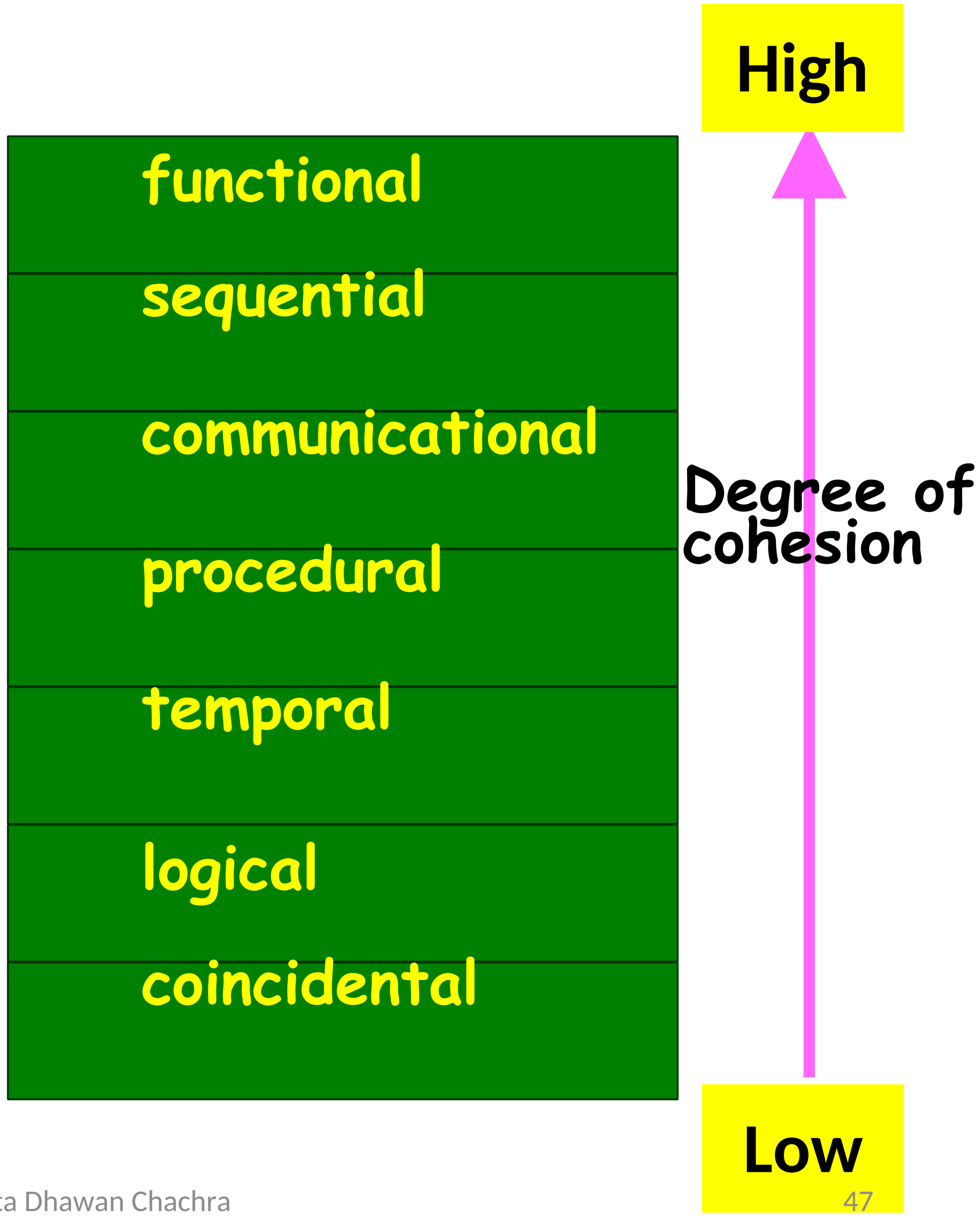
Classification of Cohesiveness

The different classes of cohesion that modules can possess are:



Classification of Cohesiveness

- The cohesiveness increases from coincidental to functional cohesion.
  - ❑ That is, coincidental is the worst type of cohesion and
  - ❑ functional is the best cohesion possible.

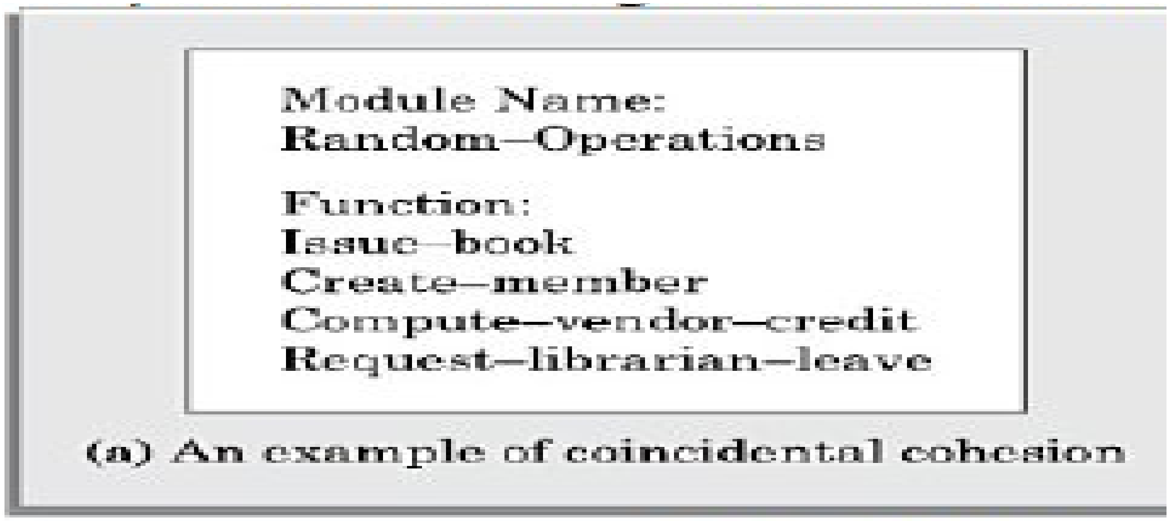


Coincidental cohesion

- The module performs a set of tasks:
  - which relate to each other very loosely, if at all.
    - That is, the module contains a random collection of functions.
    - **functions have been put in the module out of pure coincidence without any thought or design.**

Coincidental cohesion

- The designs made by novice programmers often possess this category of cohesion,
  - since they often bundle functions to modules rather arbitrarily.



Module Name:
Random–Operations

Function:
Issue–book
Create–member
Compute–vendor–credit
Request–librarian–leave

(a) An example of coincidental cohesion

Coincidental Cohesion - example

```
Module AAA{

        Print-inventory();

        Register-Student();

        Issue-Book();
};
```

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.

Logical Cohesion

```
module print{
        void print-grades(student-file){ …}

        void print-certificates(student-file){…}

        void print-salary(teacher-file){…}
}
```

Temporal cohesion

- The module contains functions so that:
  - **all the functions must be executed in the same time span.**
- Example:
  - The set of functions responsible for
    - initialization,
    - start-up, shut-down of some process, etc.

```
init() {
        Check-memory();
        Check-Hard-disk();
        Check-Keyboard();
        Check-Monitor();
        Initialize-Ports();
        Load-Kernel()
        Display-Login-Screen();
}
```

Temporal Cohesion – Example

When a computer is booted, several functions need to be performed. These include initialisation of memory and devices, loading the operating system, etc.

Procedural cohesion

- The set of functions of the module:
  - all parts of a procedure (algorithm)
  - <u>certain sequence of steps have to be carried out in a certain order for achieving an objective,</u>
    - e.g. the algorithm for decoding a message.

Procedural cohesion

- A module is said to possess procedural cohesion:
  - if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data.

Order Processing In A Trading House

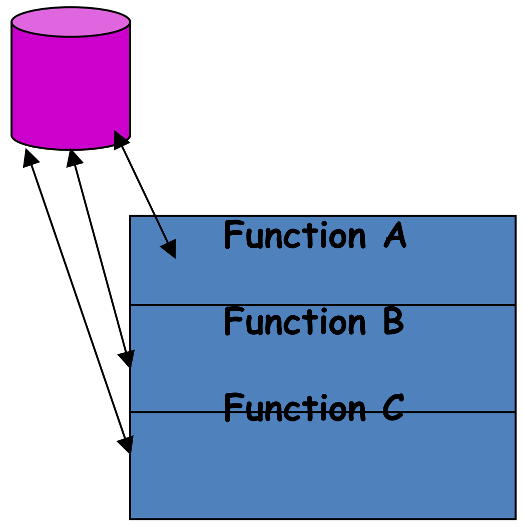
login()
place-order()
check-order()
printbill()
place-order-on-vendor()
update-inventory()
logout()
All do different thing and operate on different data. Normally executed one after the other during typical order processing by a sales clerk.

Communicational cohesion

- **All functions of the module:**
  - Reference or update the same data structure,
- **Example:**
  - The set of functions defined on an array or a stack.

Communicational Cohesion

```
handle-Student- Data() {
        Static Struct  Student-data[10000];
        Store-student-data();
        Search-Student-data();
        Print-all-students();
};
```

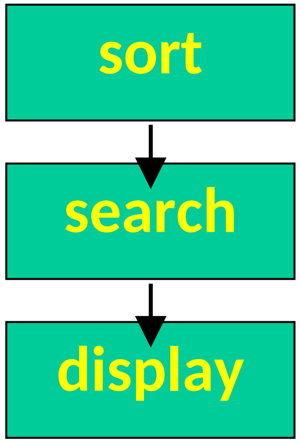


Communicational
Access same data

Different functions in the module access and manipulate data stored in an structure named **Student-data** defined within the module.

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:



Sequential cohesion

- A module is said to possess sequential cohesion:
  - if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence.

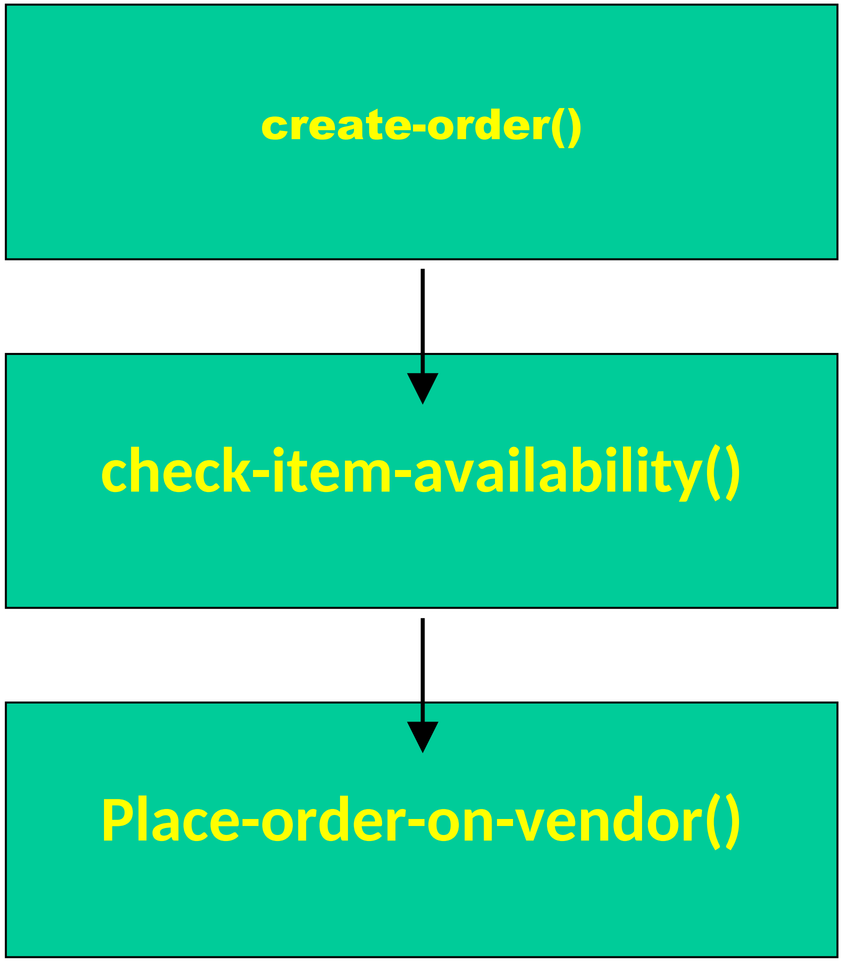
On-line Store

- After a customer requests for some item, it is first determined if the item is in stock.
- Observe that :

❑ function create-order() creates an order that is processed by the

❑ function check-item-availability() (whether the items are available in the required quantities in the inventory) is input to

❑ place-order-on-vendor().



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.**

Functional cohesion

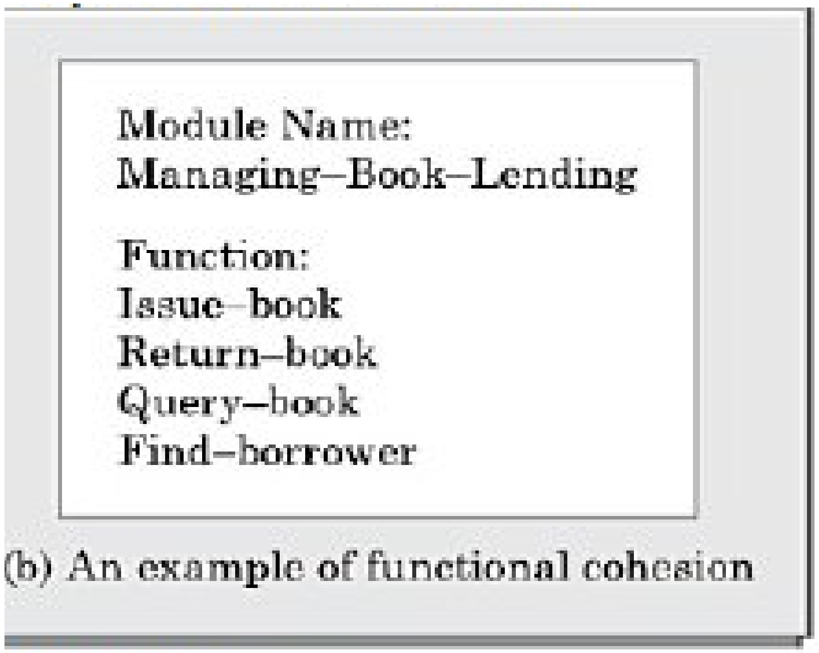
- A module containing all the functions required to manage employees’ pay-roll e.g.:
  - computeOvertime()
  - computeWorkHours()
  - computeDeductions()

  work together to generate the payslips of the employees.

Functional cohesion

- All functions together manage all activities concerned with book lending

- **We can describe the function using a single sentence.**
- **“It manages the book lending procedure of the library.”**



Module Name:
Managing–Book–Lending

Function:
Issue–book
Return–book
Query–book
Find–borrower

(b) An example of functional cohesion

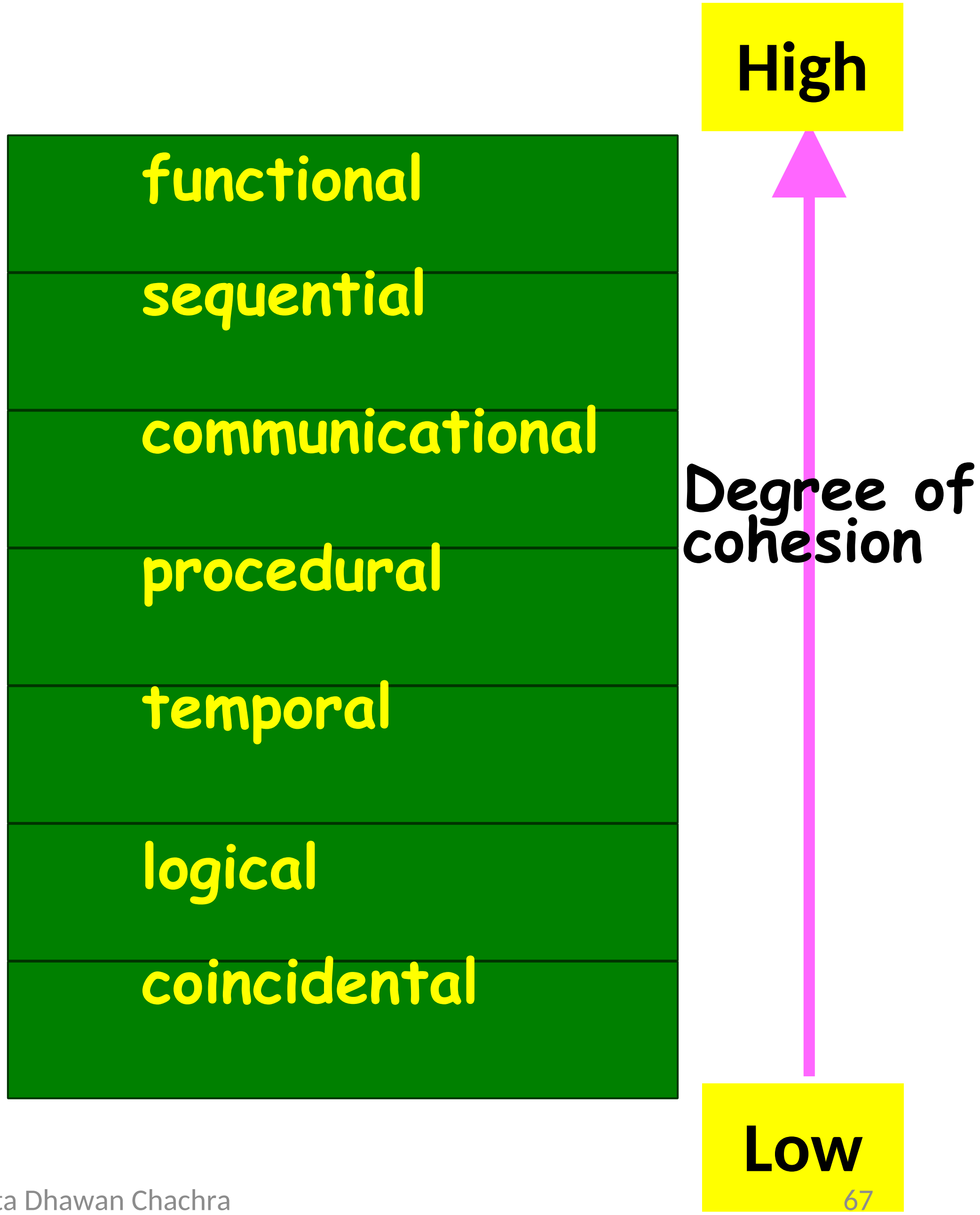
Write down a sentence to describe the function of the module

Determining Cohesiveness

- 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”, “setup”, “shut down”, etc.,
  - it probably has temporal cohesion.

Classification of Cohesiveness

- The cohesiveness increases from coincidental to functional cohesion.
  - That is, coincidental is the worst type of cohesion and
  - functional is the best cohesion possible.



Coupling

- Cohesion is a measure of the functional strength of a module, whereas,

- Coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

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.**

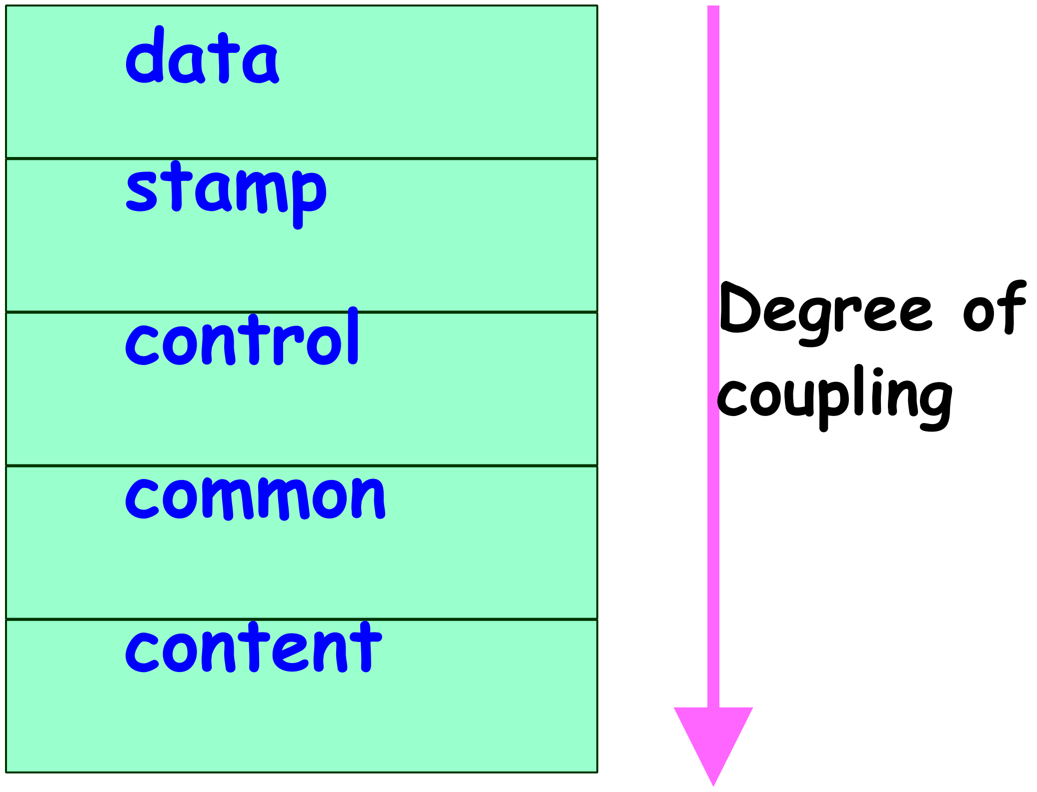
Coupling

- Two modules are said to be highly coupled, if either of the following two situations arise:
  - If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled.
  - If the interactions occur through some shared data, then also we say that they are highly coupled.

Coupling

- There are no ways to precisely measure 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 stamp coupled,
  - if they communicate via a composite data item
    - or an array or structure in C.
    - record in PASCAL

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.

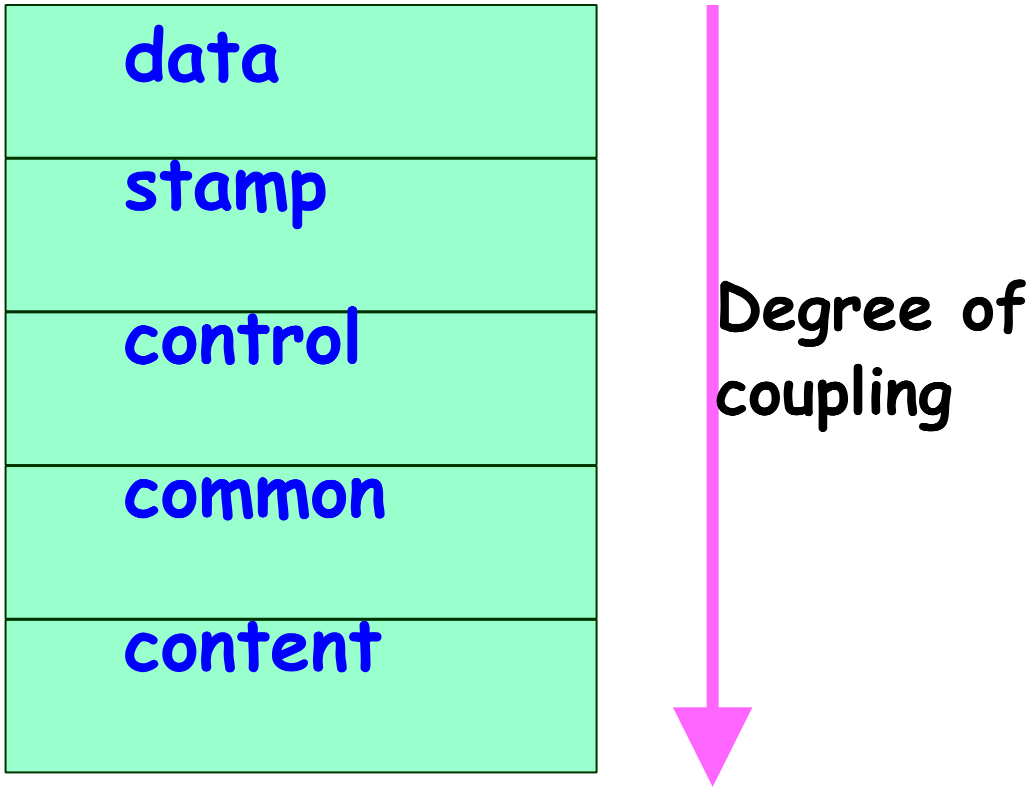
Common Coupling

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

Content coupling

- Content coupling exists between two modules:
  - if they share code,
  - e.g, branching from one module into another module.
  - e.g, jump from one module into the code of another module can occur
- The degree of coupling increases
  - from data coupling to content coupling.

Classes of coupling



The degree of coupling increases from data coupling to content coupling.

Classes of coupling

High coupling among modules not only makes a design solution:

- ❑ difficult to understand and maintain,
- ❑ but it also increases development effort and
- ❑ also makes it very difficult to get these modules developed independently by different team members.

MCQs

Q1)The extent of data exchanges between two modules is called which one of the following?

a. Coupling

b. Cohesion

c. Structure

d. Union

- Ans)a

MCQs

Q2)During the detailed design of a module, which one of the following is designed?

a. Data structures and algorithms

b. Control structure

c. Data flow structure

d. Module interfaces

Ans)a

MCQs

Q3)Which one of the following types of cohesion can be considered as the best form of cohesion?

a. Logical
b. Coincidental
c. Temporal
d. Functional

Ans)d

MCQs

Q4)Which one of the following is the worst type of module cohesion?

a. Logical Cohesion

b. Temporal Cohesion

c. Functional Cohesion

d. Coincidental Cohesion

- Ans)d

MCQs

Q5)If all tasks must be executed in the same time-span, what type of cohesion is being exhibited?

a. Functional Cohesion

b. Temporal Cohesion

c. Sequential Cohesion

d. None of these

Ans)b

Q6)Which one of the following is the correct ordering of the coupling of modules from strongest (least desirable) to weakest (most desirable)?
a. content, common, control, stamp, data
b. common, content, control, stamp, data
c. content, data, common ,stamp, common
d. data, control, common, stamp, content

Ans)a

Q7)In which one of the following types of coupling, complete data structures are passed from one module to another?

- a. Control Coupling
- b. Stamp Coupling
- c. External Coupling
- d. Content Coupling

- Ans)b

Q8)Which of the following is the best (most desirable) type of module coupling?

- a. Control coupling
- b. Stamp coupling
- c. Data coupling
- d. Content coupling

- Ans)c

Q9)Which one of the following is the worst type of module coupling?

- a. Control coupling
- b. Stamp coupling
- c. External coupling
- d. Content coupling

- Ans)d

Q10)The modules in a good software design should have which one of the following characteristics?
a. High cohesion, low coupling
b. Low cohesion, high coupling
c. Low cohesion, low coupling
d. High cohesion, high coupling

Ans)a

GATE | GATE-CS-2009 | Question 19

Q11) The coupling between different modules of a software is categorized as follows:
I. Control coupling II. Data coupling III. Content coupling IV. Stamp coupling V. Common coupling
Coupling between modules can be ranked in the order of strongest (least desirable) to weakest (most desirable) as follows:
(A) I-II-III-IV-V
(B) V-IV-III-II-I
(C) I-III-V -II-IV
(D) III-V-I-IV-II

Ans)d

Design Approaches

- **Function Oriented Design**
- **Object-Oriented Design**

Design Approaches

- 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** (top-down decomposition).
  - 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
- Split these into further subfunctions, etc.

Function-Oriented Design

- 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

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.

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-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.

Object-Oriented versus Function-Oriented Design

- 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."**

Object-Oriented versus Function-Oriented Design

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

Example

- For Function Oriented Design-In an employee pay-roll system, the following can be **global data:**
  - names of the employees,
  - their code numbers,
  - basic salaries, etc.
- In contrast, in object oriented systems:
  - data is distributed among different employee objects of the system.

Object-Oriented versus Function-Oriented Design

- in OOD, Objects communicate by message passing.
  - one object may discover the state information of another object by interrogating it.

Object-Oriented versus Function-Oriented Design

- Of course, somewhere or other the functions must be implemented:
  - the functions are usually associated with specific real-world entities (objects)
  - directly access only part of the system state information.

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.

Object-Oriented versus Function-Oriented Design

- 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 2000 rooms in the building.

Fire-Alarm System

- Different rooms of the building:
  - fitted with smoke detectors and fire alarms.
- The fire alarm system would monitor:
  - status of the smoke detectors.

Fire-Alarm System

- 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 neighbouring 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.

Fire-Alarm System

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

```
/* Global data (system state) accessible by various functions */
BOOL           detector_status[2000];
int            detector_locs[2000];
BOOL           alarm-status[2000]; /* alarm activated when  set */
int            alarm_locs[2000]; /* room number where alarm is located */
int            neighbor-alarms[2000][10];/*each detector has at most*/
                                         /* 10 neighboring alarm locations */
 interrogate_detectors();
 get_detector_location();
 determine_neighbor();
 ring_alarm();
 reset_alarm();
 report_fire_location();
```

Function-Oriented Approach

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

- **Appropriate number of instances of the class detector and alarm are created.**

Object-Oriented versus Function-Oriented Design

- In a function-oriented program :
  - the system state is centralized
  - several functions accessing these data are defined.
- In the object oriented program,
  - the state information is distributed among various sensor and alarm objects.

Object-Oriented versus Function-Oriented Design

- Use OOD to design the classes:
  - then applies top-down function oriented techniques
    - to design the internal methods of classes.

Object-Oriented versus Function-Oriented Design

- Though outwardly a system may appear to have been developed in an object oriented fashion,
  - but inside each class there is a small hierarchy of functions designed in a top-down manner.