IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

NPTEL

Software Design

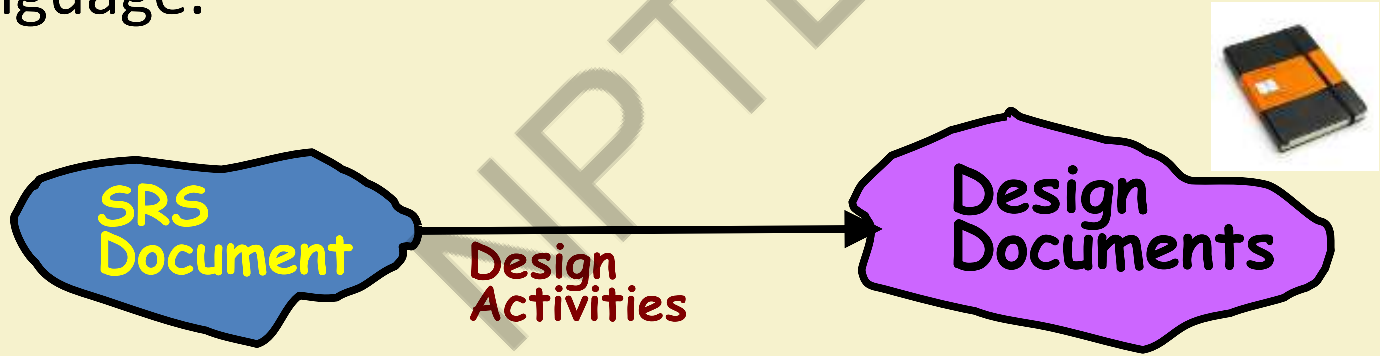
Rajib Mall

CSE Department

IIT KHARAGPUR

What is Achieved during design phase?

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

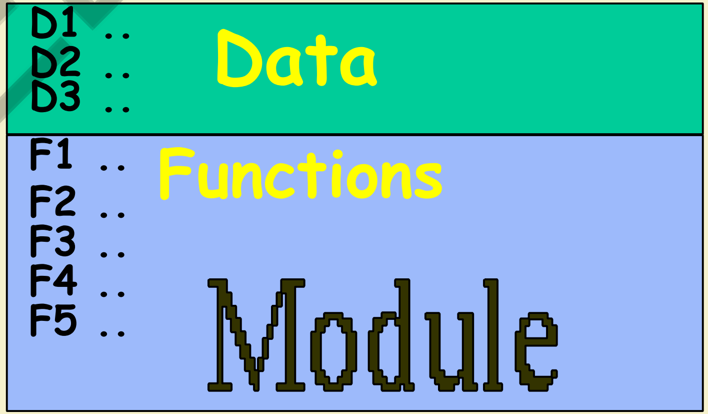


Items Designed During Design Phase

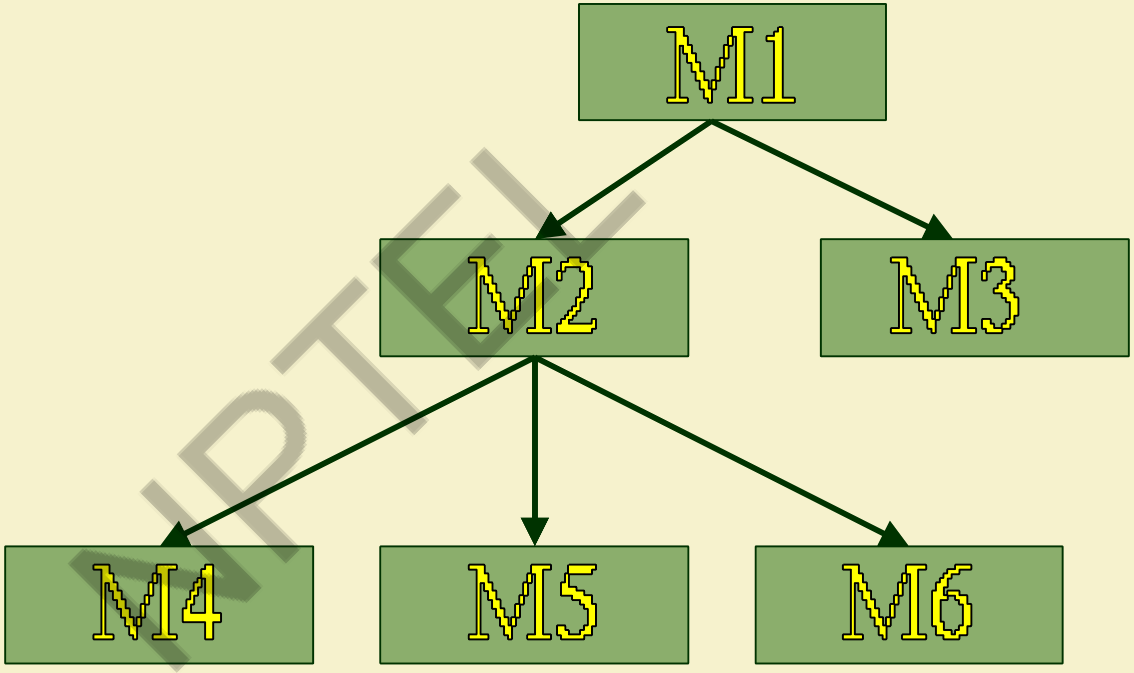
- Module structure,
- Control relationship among the modules
  - call relationship or invocation relationship
- Interface among different modules,
  - data items exchanged among different modules,
- Data structures of individual modules,
- algorithms for individual modules.

Module

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



Module Structure



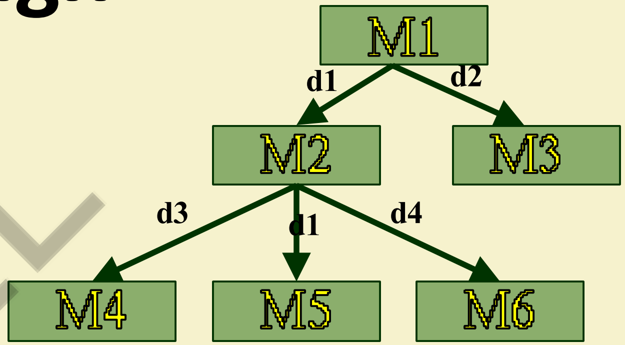
Iterative Nature of Design

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

Stages in Design

- Design activities are usually classified into two stages:
  - **Preliminary (or high-level) design**
  - **Detailed design.**
- Meaning and scope of the two stages:
  - vary considerably from one methodology to another.

High-level design



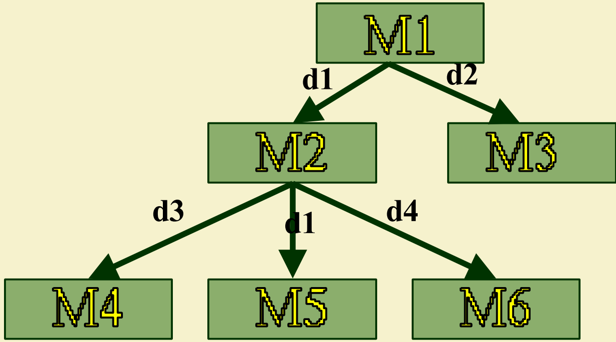
- Identify:
  - modules
  - control relationships among modules
  - interfaces among modules.

High-level design

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

High-level Design

- Several notations are available to represent high-level design:
  - Usually a tree-like diagram called **structure chart** is used.
  - Other notations:
    - Jackson diagram or Warnier-Orr diagram can also be used.



Detailed design

- For each module, design for it:
  - data structure
  - algorithms
- Outcome of detailed design:
  - module specification.

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.

What Is Good Software Design?

- 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.

What Is a Good Software Design?

- 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,
- 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

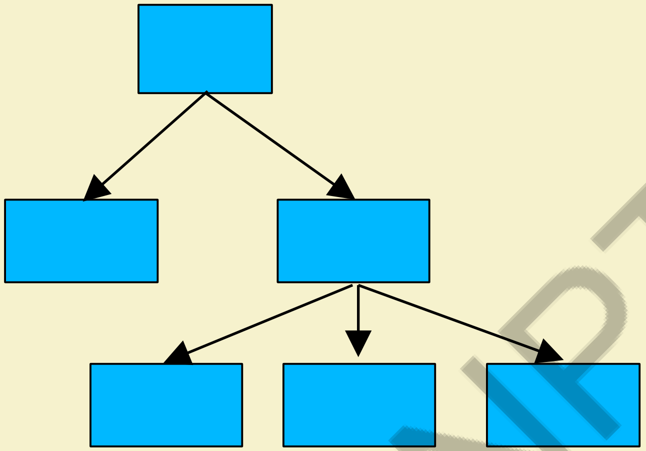
Modularity

- 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.**

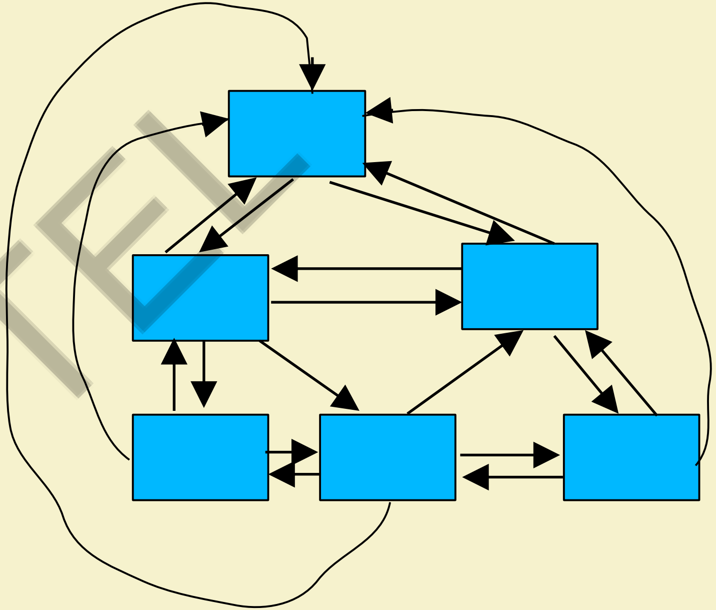
Modularity

- 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.

Layering



Superior

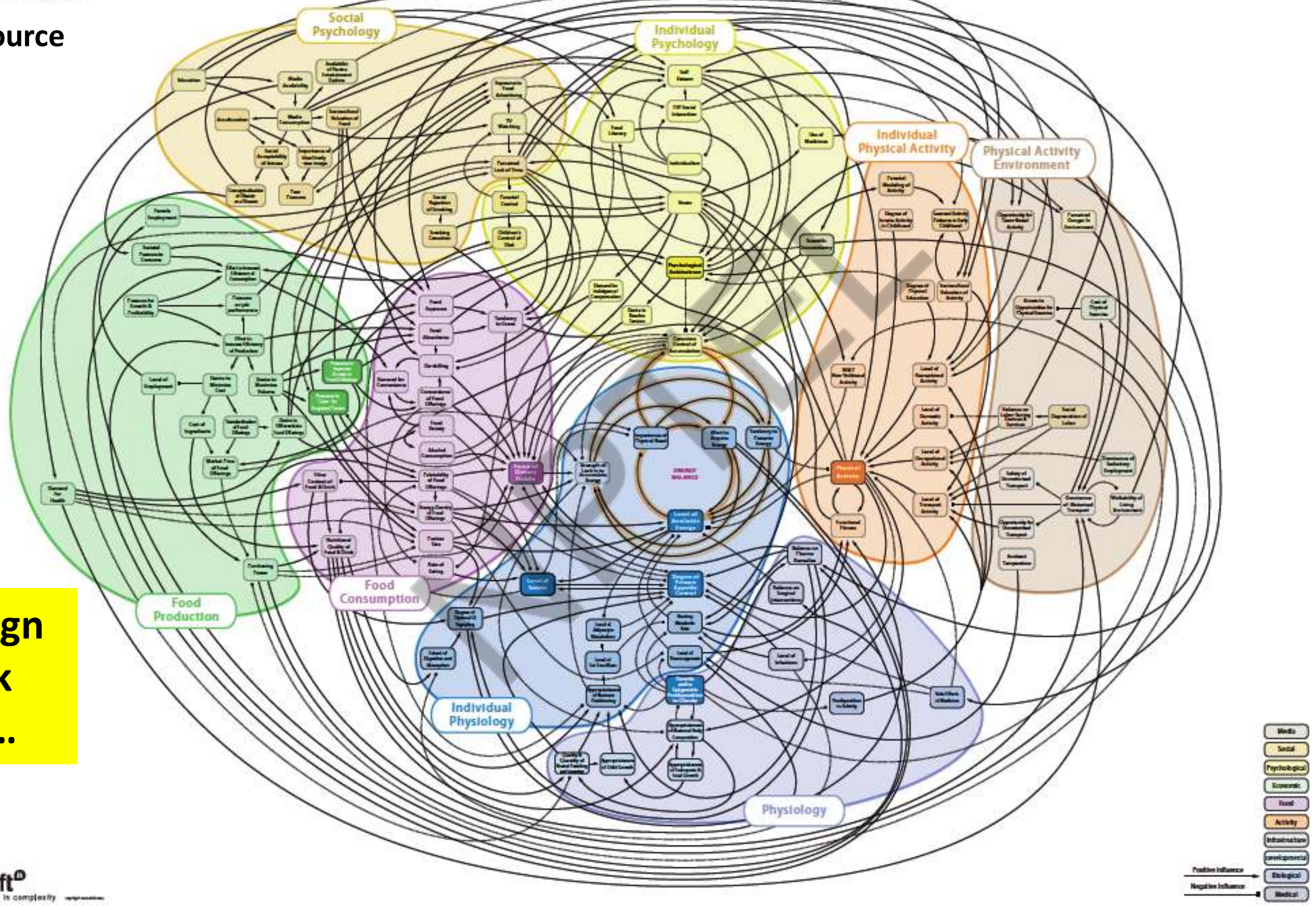


Inferior

Obesity System Map

Variable Clusters

:Source



Bad design may look like this…

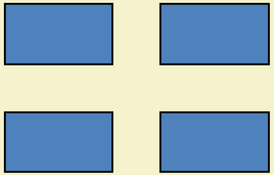
Modularity

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

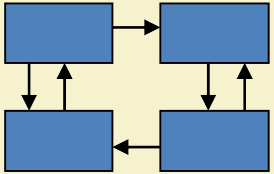
Modularity

- Arrangement of modules in a hierarchy ensures:
  - **Low fan-out**
  - **Abstraction**

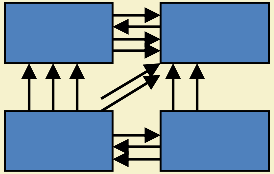
Coupling: Degree of dependence among components



No dependencies



Loosely coupled-some dependencies



Highly coupled-many 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.

Source:

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

Cohesion and Coupling

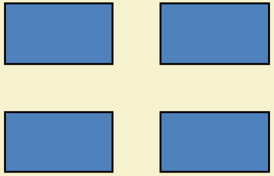
- Cohesion is a measure of:
  - functional strength of a module.
  - **A cohesive module performs a single task or function.**
- Coupling between two modules:
  - **A measure of the degree of interdependence or interaction between the two modules.**

Cohesion and Coupling

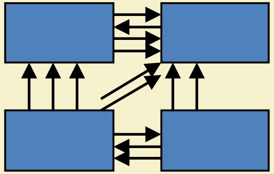
- A module having **high cohesion and low coupling**:
  - **Called functionally independent** of other modules:
    - A functionally independent module needs very little help from other modules and therefore has minimal interaction with other modules.

Advantages of Functional Independence

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



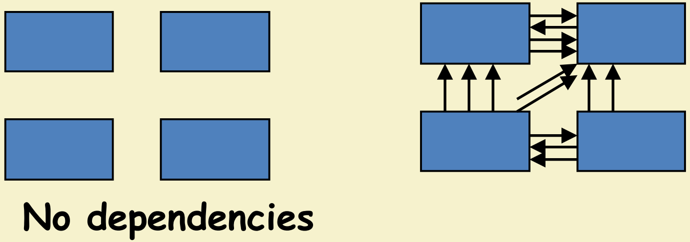
No dependencies



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.

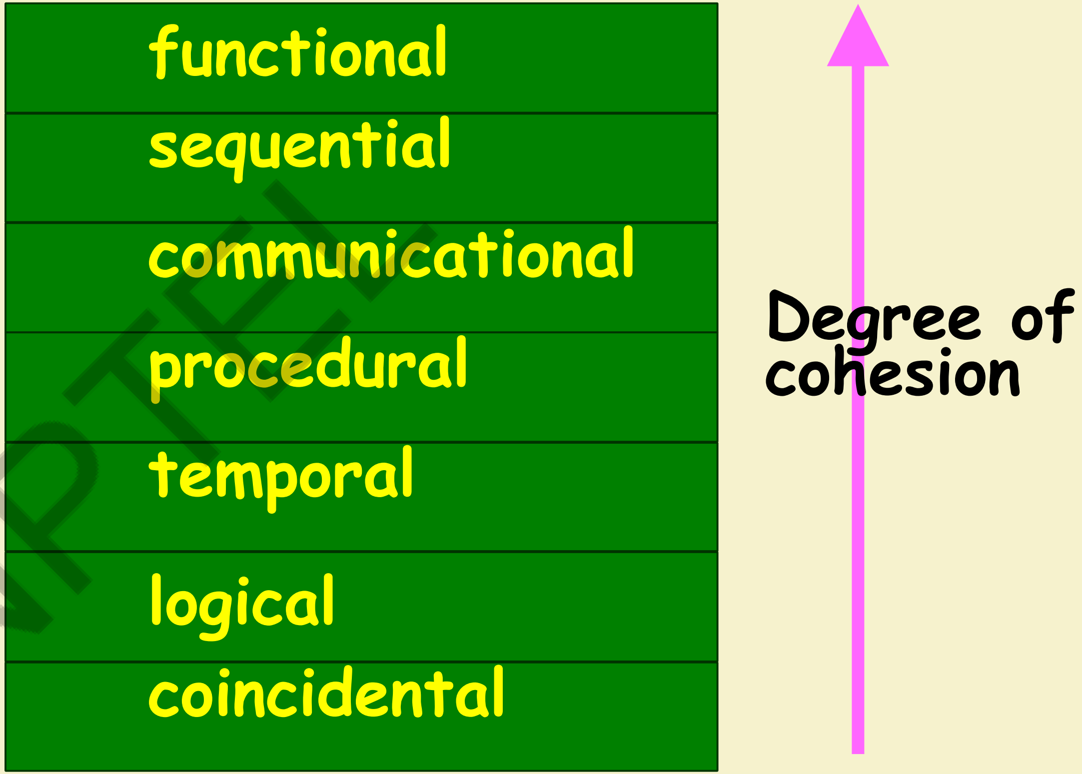
Measuring Functional Independence

- 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



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 - 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();
        Initialize-Ports();
        Display-Login-Screen();
}
```

Temporal Cohesion – Example

Procedural cohesion

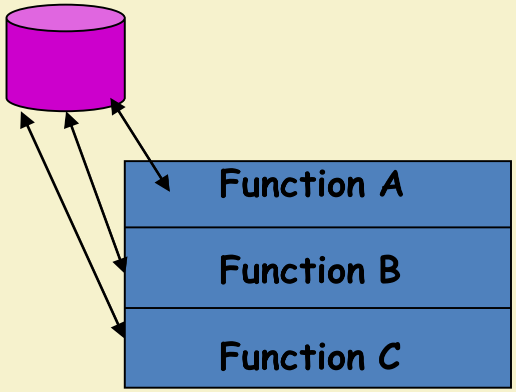
- The set of functions of the module:
  - all part of a procedure (algorithm)
  - certain sequence of steps have to be carried out in a certain order for achieving an objective,
    - e.g. the algorithm for decoding a message.

Communicational cohesion

- 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

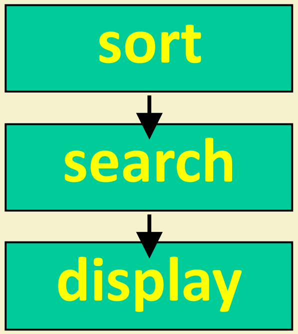
```
handle-Student- Data() {
        Static Struct  Student-data[10000];
        Store-student-data();
        Search-Student-data();
        Print-all-students();
};
```



Communicational
Access same data

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.**

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,
  - it probably has temporal cohesion.

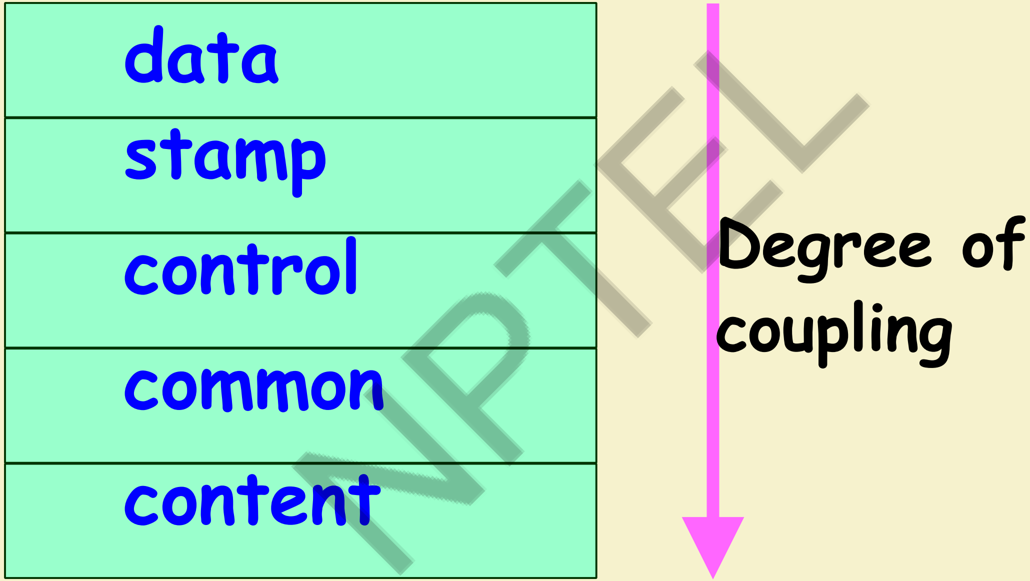
Coupling

- 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 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.

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

Hierarchical Design

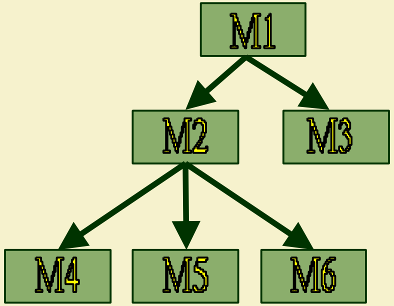
- Control hierarchy represents:
  - organization of modules.
  - control hierarchy is also called program structure.
- Most common notation:
  - a tree-like diagram called structure chart.

Good Hierachical Arrangement of modules

- Essentially means:
  - **low fan-out**
  - **abstraction**

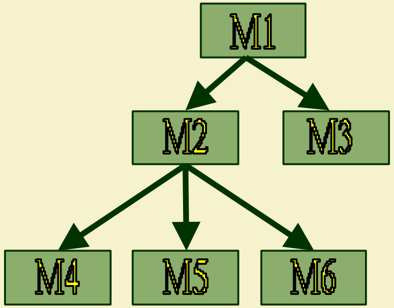
Characteristics of Module Structure

- **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.

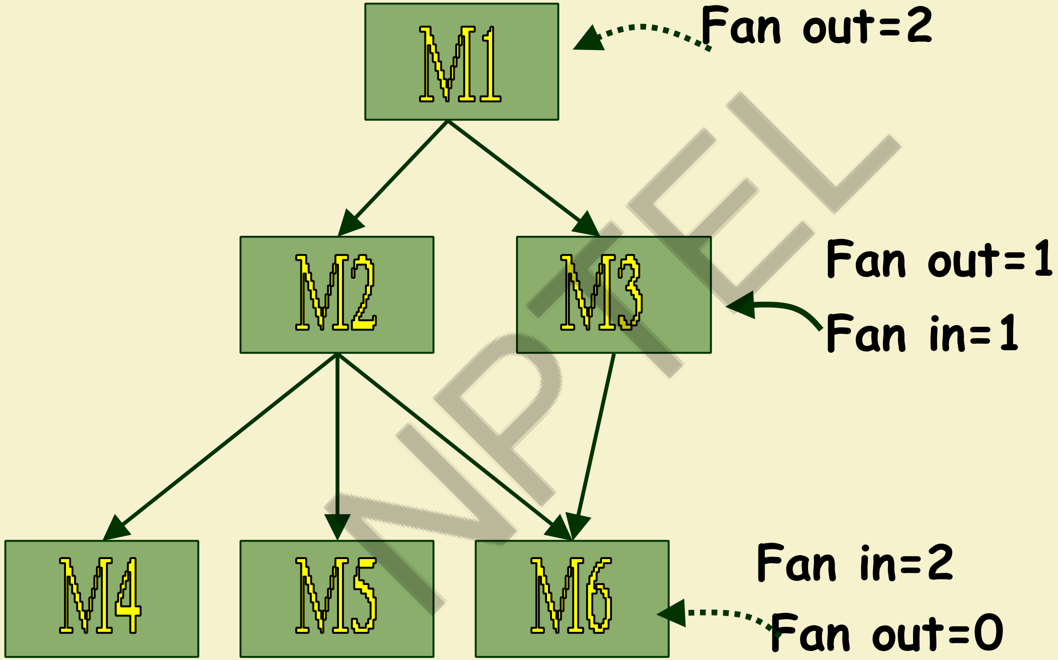


Characteristics of Module Structure

- Fan-in:
  - indicates how many modules directly invoke a given module.
  - High fan-in represents code reuse and is in general encouraged.

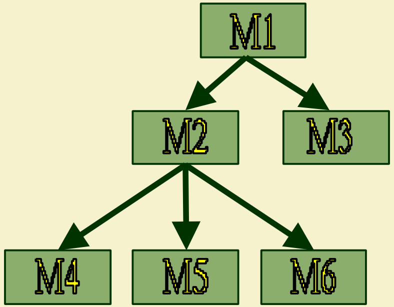


Module Structure

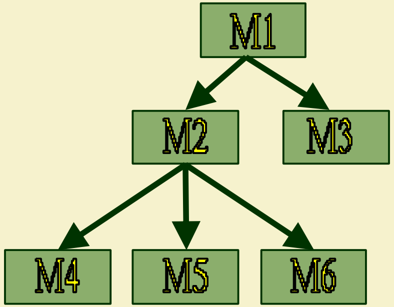


Goodness of Design

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



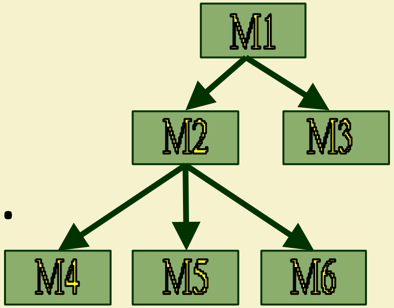
Large Fan Out



- 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

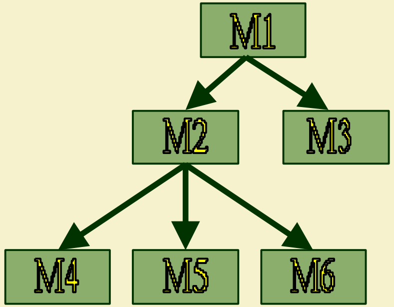
- A module that controls another module:
  - said to be superordinate to the later module.



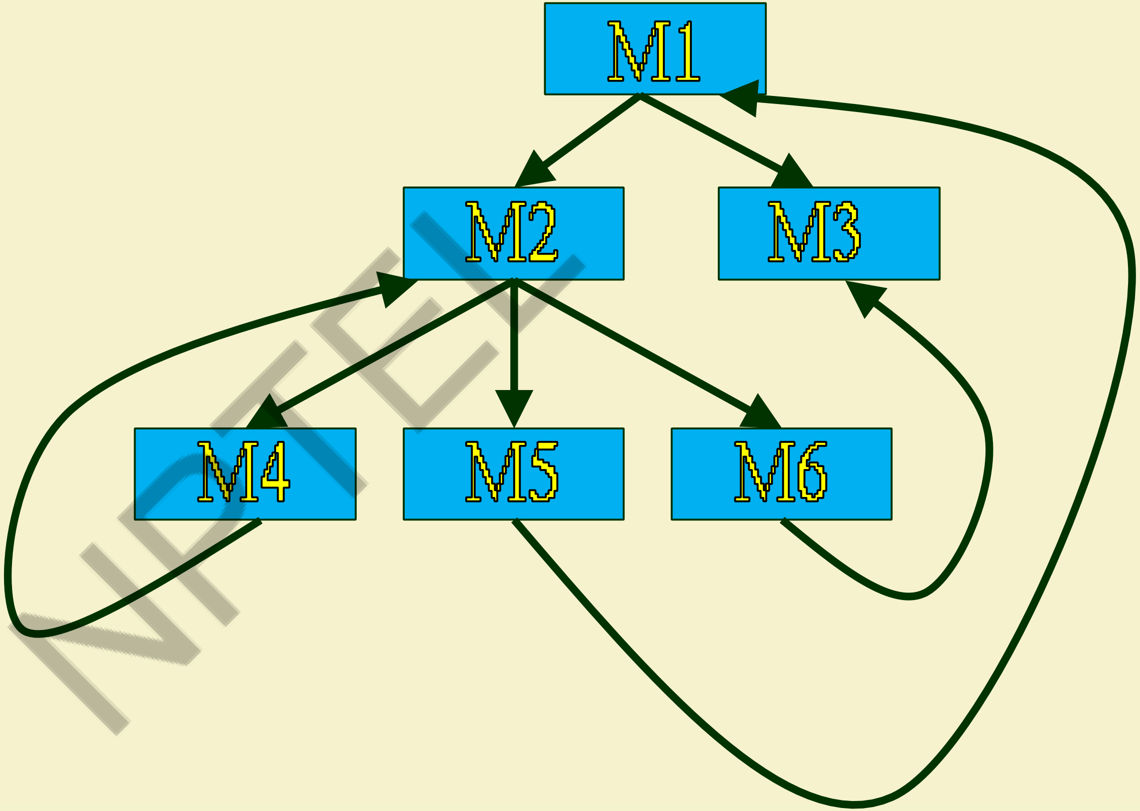
- Conversely, a module controlled by another module:
  - said to be subordinate to the later module.

Visibility and Layering

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

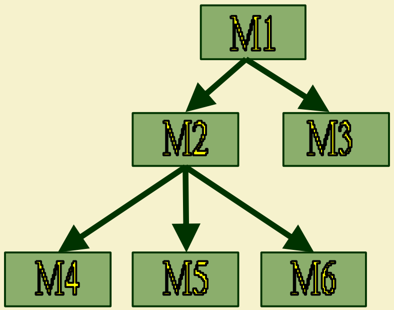


Bad Design



Abstraction

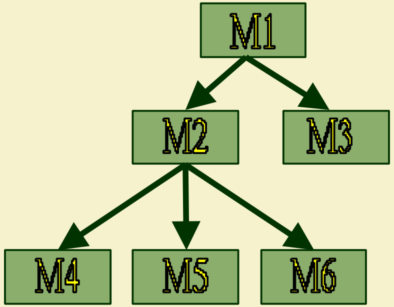
- **Lower-level modules:**
  - Perform input/output and other low-level functions.



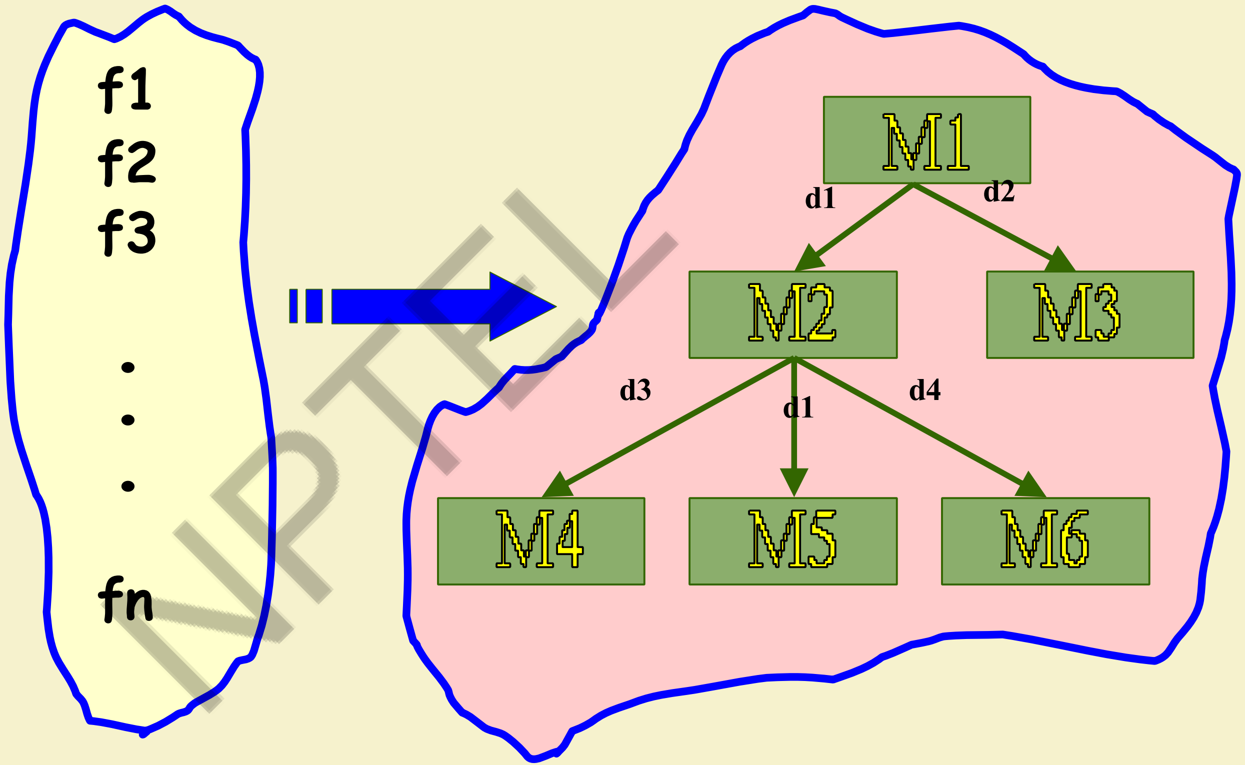
- **Upper-level modules:**
  - Perform more managerial functions.

Abstraction

- The principle of abstraction requires:
  - lower-level modules do not invoke functions of higher level modules.
  - Also known as layered design.



High-level Design



Design Approaches

- Two fundamentally different software 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

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

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.

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

- 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

- 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.

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 design:
    - 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 examined the essential philosophy behind these two approaches
  - these two approaches are not really competing but complementary approaches.

Thank You!!