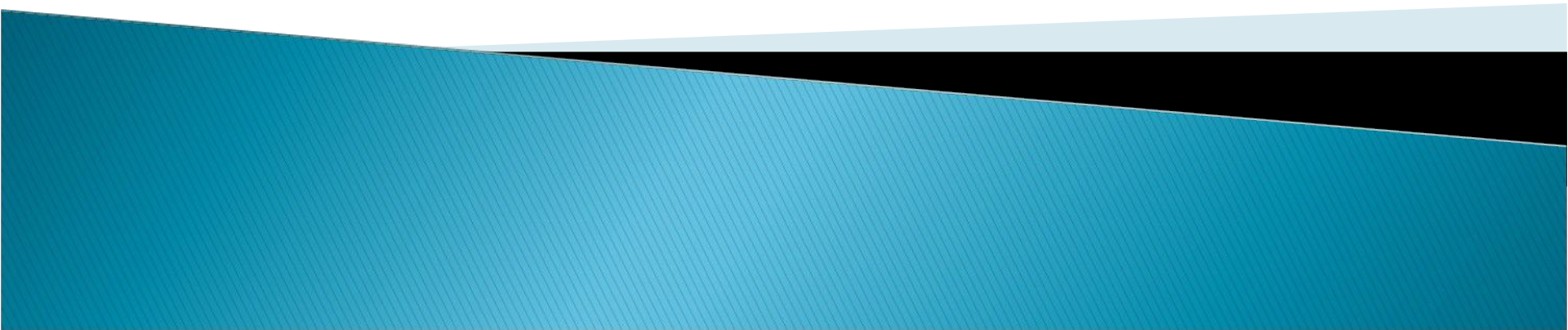
Design Concepts and Principles



Design model

Components of a design model :

- Data Design
  - Transforms information domain model into data structures required to implement software
- Architectural Design
  - Defines relationship among the major structural elements of a software
- Interface Design
  - Describes how the software communicates with systems that interact with it and with humans.
- Procedural Design
  - Transforms structural elements of the architecture into a procedural description of software components

- Software Design -- An iterative process transforming requirements into a “blueprint” for constructing the software.

Goals of the design process:

3. The design must **implement all of the explicit requirements** contained in the analysis model and it must accommodate all of the implicit requirements desired by the customer.
4. The design must be **a readable, understandable guide** for those who generate code and for those who test and subsequently support the software.
5. The design should **address the data, functional and behavioral domains** from an implementation perspective

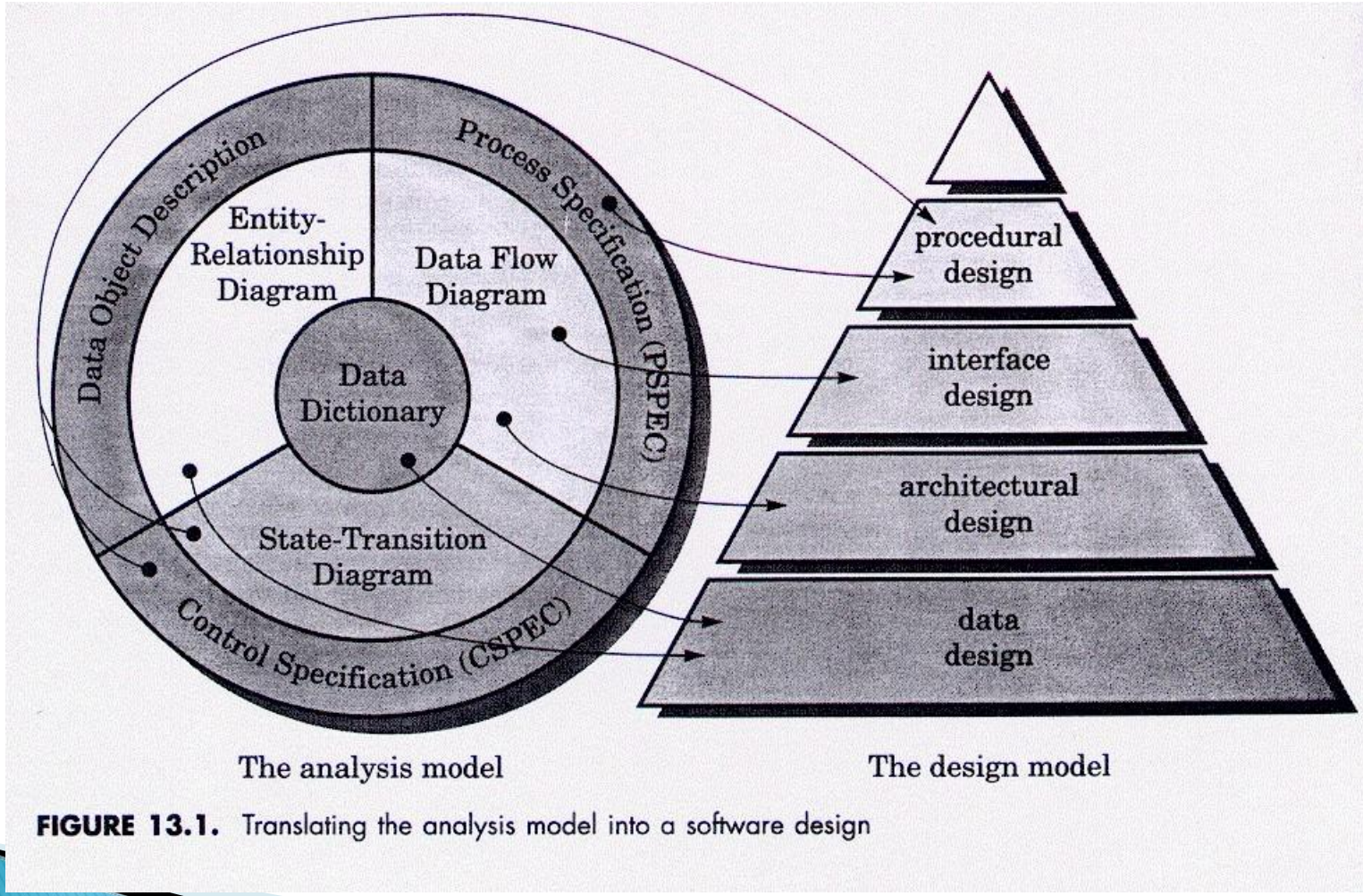


FIGURE 13.1. Translating the analysis model into a software design

Design Concepts(Abstraction)

- **Defintion:** “Abstraction permits one to concentrate on a problem at some level of abstraction without regard to low level detail.
- At the highest level of abstraction a solution is stated in broad terms using the language of the problem environment.
- At lower level, a procedural orientation is taken.
- At the lowest level of abstraction the solution is stated in a manner that can be directly implemented.

Design Concepts(Abstraction)

Types of abstraction :

1. **Procedural Abstraction :**

A named sequence of instructions that has a specific & limited function

Eg: Word OPEN for a door

2. **Data Abstraction :**

A named collection of data that describes a data object. Data abstraction for door would be a set of attributes that describes the door

(e.g. door type, swing direction, weight, dimension)

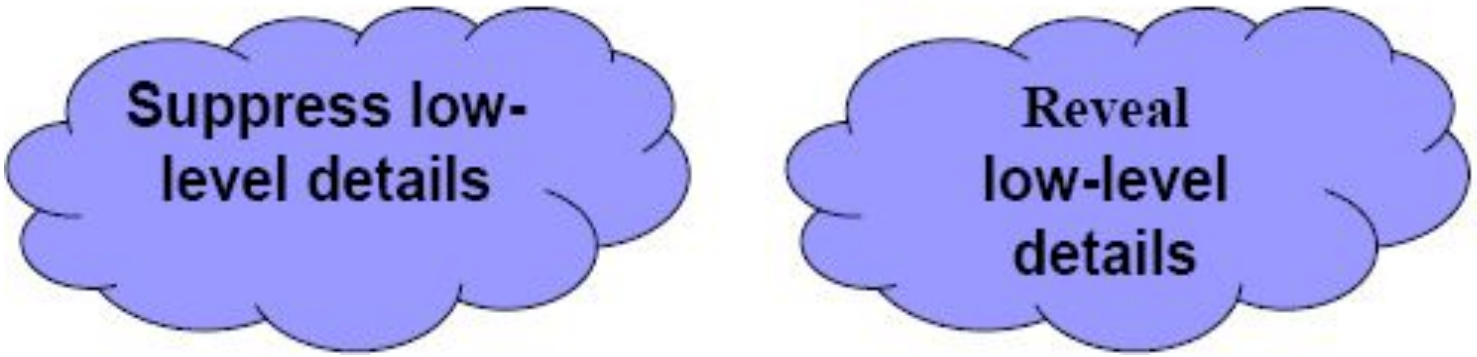
Design Concepts(Refinement)

2. Refinement

- Process of elaboration.
- Start with the statement of function defined at the abstract level, decompose the statement of function in a stepwise fashion until programming language statements are reached.

Design Concepts(Refinement)

abstraction & refinement are complementary concepts



Design Concepts(Modularity)

- In this concept, software is divided into separately named and addressable components called modules
- Follows "divide and conquer" concept, a complex problem is broken down into several manageable pieces
- Let p_1 and p_2 be two problems.
- Let E1 and E2 be the effort required to solve them --

If C(p1)>C(p2)
Hence E(p1)>E(p2)

Design Concepts(Modularity)

Now--

Complexity of a problem that combines p1 and p2 is greater than complexity when each problem is consider

$C(p_1+p_2) > C(p_1)+C(p_2)$,

Hence

$E(p_1+p_2) > E(p_1)+E(p_2)$

It is easier to solve a complex problem when

you break it into manageable pieces

Design Concepts(Modularity)

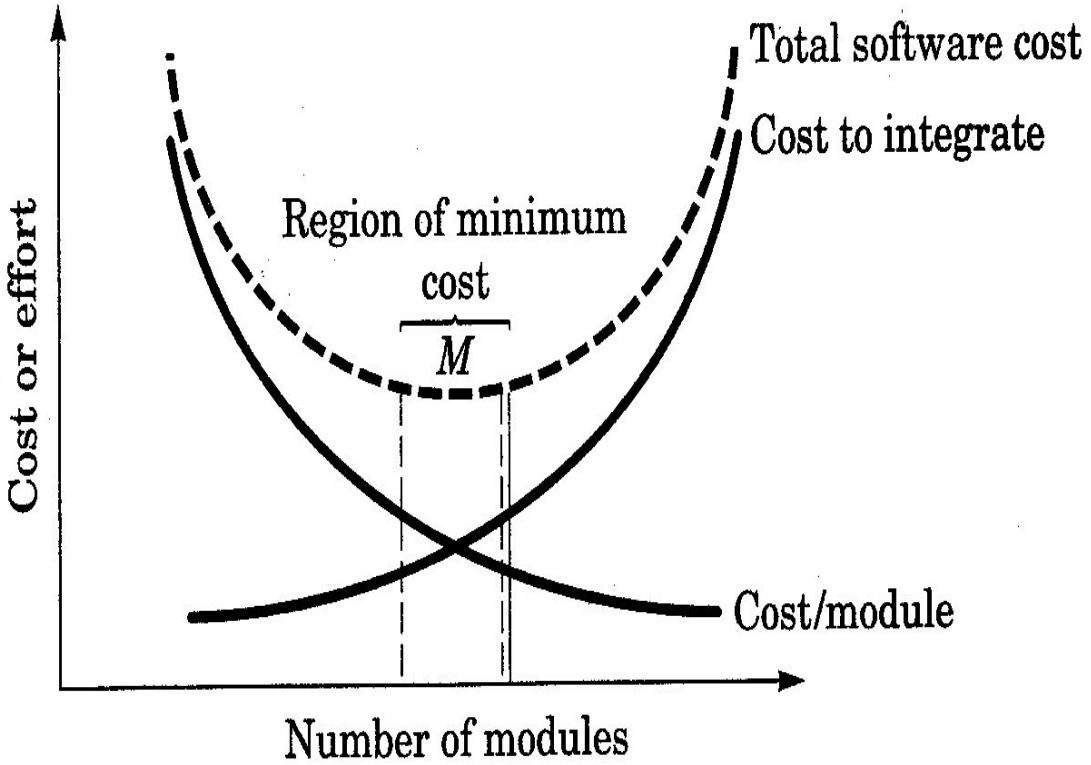


FIGURE 13.2.
Modularity and software cost

Design Concepts(Modularity)

5 criteria to evaluate a design method with respect to its modularity-----

Modular understandability

module should be understandable as a standalone unit (no need to refer to other modules)

Modular continuity

If small changes to the system requirements result in changes to individual modules, rather than system wide

changes, the impact of side effects will be minimized

Modular protection

- If an error occurs within a module then those errors are localized and not spread to other modules.

Design Concepts(Modularity)

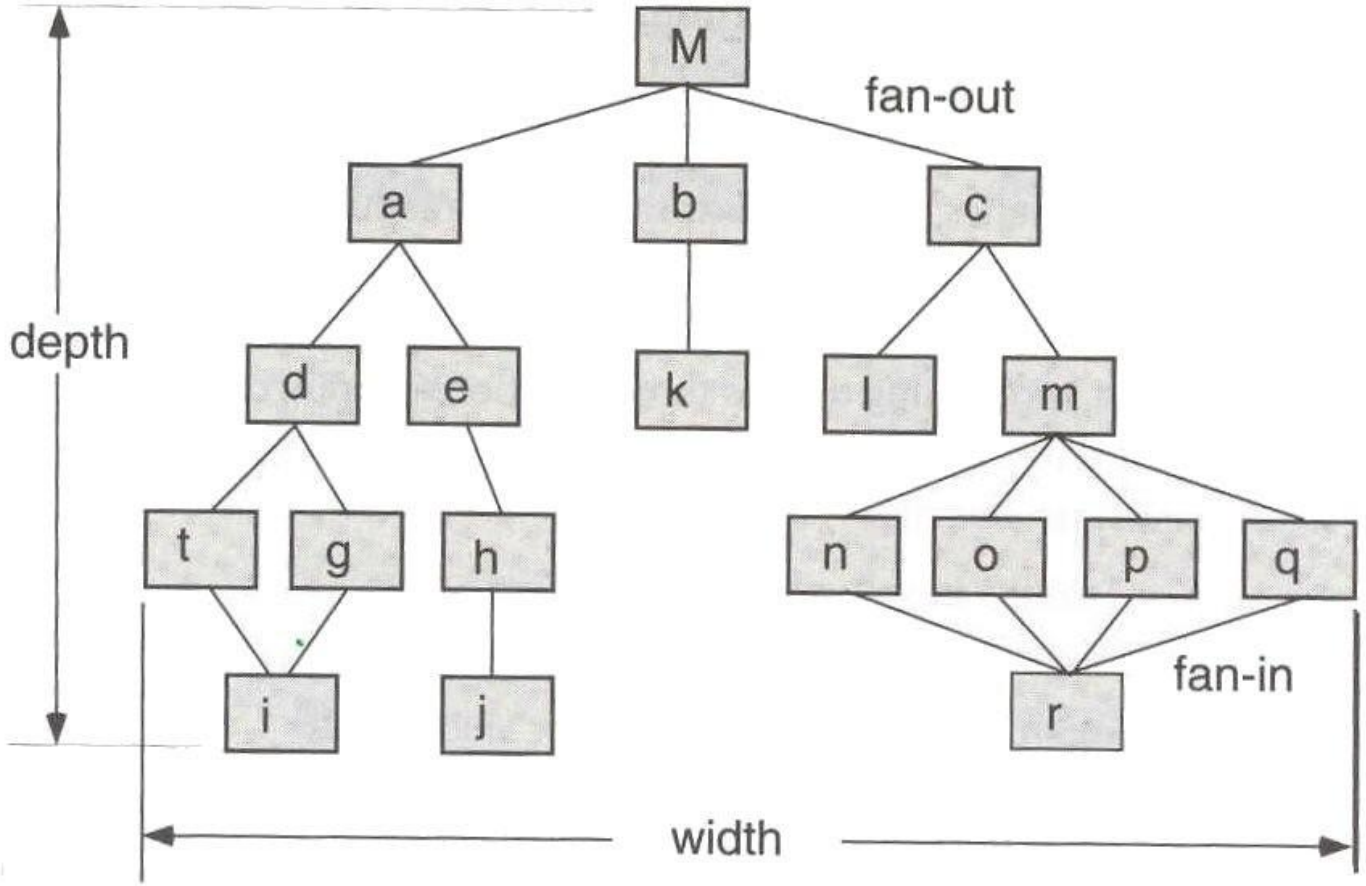
ModularComposability

Design method should enable reuse of existing components.

ModularDecomposability

Complexity of the overall problem can be reduced if the designmethod provides a systematic mechanism to decompose a problem into sub problems

Design Concepts(Control Hierarchy)



Design Concepts(Control Hierarchy)

- Also called program structure
- Represent the organization of program components.
- Does not represent procedural aspects of software such as decisions, repetitions etc.
- **Depth** –No. of levels of control (distance between the top and bottom modules in program control structure) **W idth** - Span of
- control.
- **Fan-out** -no. of modules that are directly controlled by another module
- **Fan-in** - how many modules directly control a given module
- **Super ordinate** -module that control another module
- **Subordinate** - module controlled by another

Design Concepts(Control Hierarchy)

- Visibility -set of program components **that may be** called or used as data by a given component
- Connectivity – A module that directly causes another module to begin execution is connected to it.

Design Concepts (Software Architecture)

- Software architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components.
- A set of properties should be specified as part of an architectural design:
- **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects) and the manner in which those components are packaged and interact with one another. **E**
- **xtra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, reliability, security, adaptability, and other system characteristics.
- **Families o f related systems.** the design should have the ability to reuse architectural building blocks.

Design Concepts (Data Structure)

Data structure is a representation of the logical relationship among individual elements of data.

- **Scalar item** –
  A single element of information that may be addressed by an identifier .
- Scalar item organized as a list- **array**
- Data structure that organizes non-contiguous scalar items-**linked list**

Design Concepts (Software Procedure)

Software procedure focuses on the processing details of each module individually.

Design Concepts (Information hiding)

- Information (data and procedure) contained within a module should be inaccessible to other modules that have no need for such information.
- Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

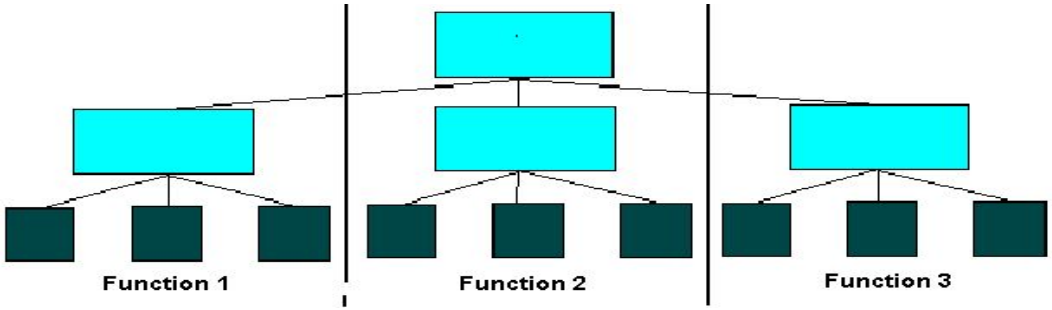
Design Concepts (Structural partitioning)

If the architectural style of a system is hierarchical program structure can be partitioned -----

3. Horizontal partitioning
4. Vertical partitioning

Design Concepts (Structural partitioning)

Horizontal partitioning



Separate branches can be defined for each major function Eg : 3 partitions

1.Input
2.Data Transformation
3.Output

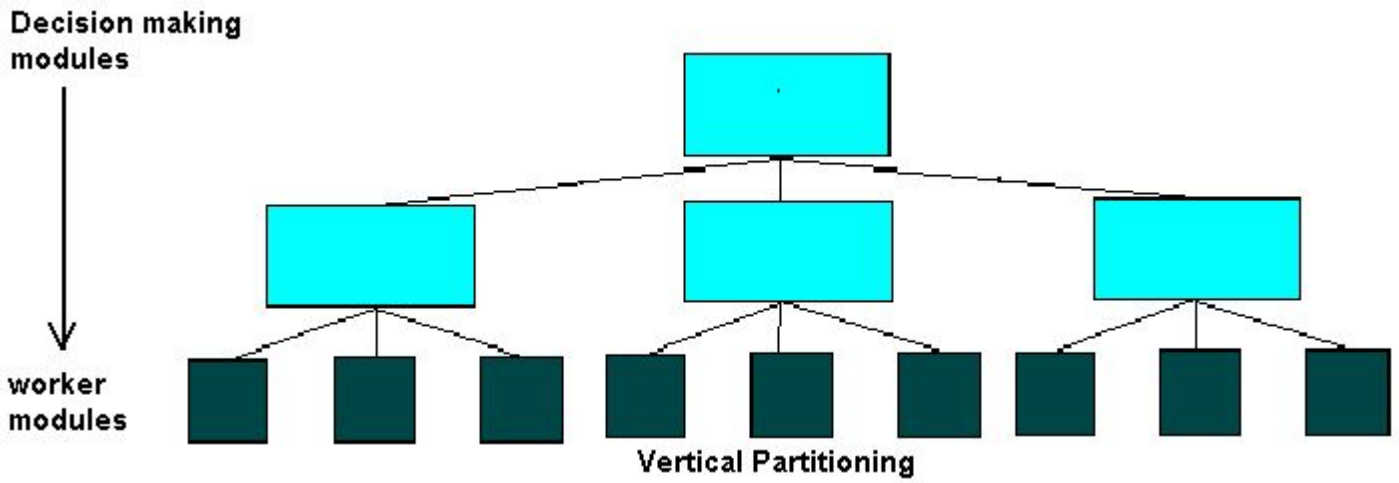
Advantages

- Easier to test
- Easier to maintain
- Propagation of fewer side effects
- Easier to add new features

Design Concepts (Structural partitioning)

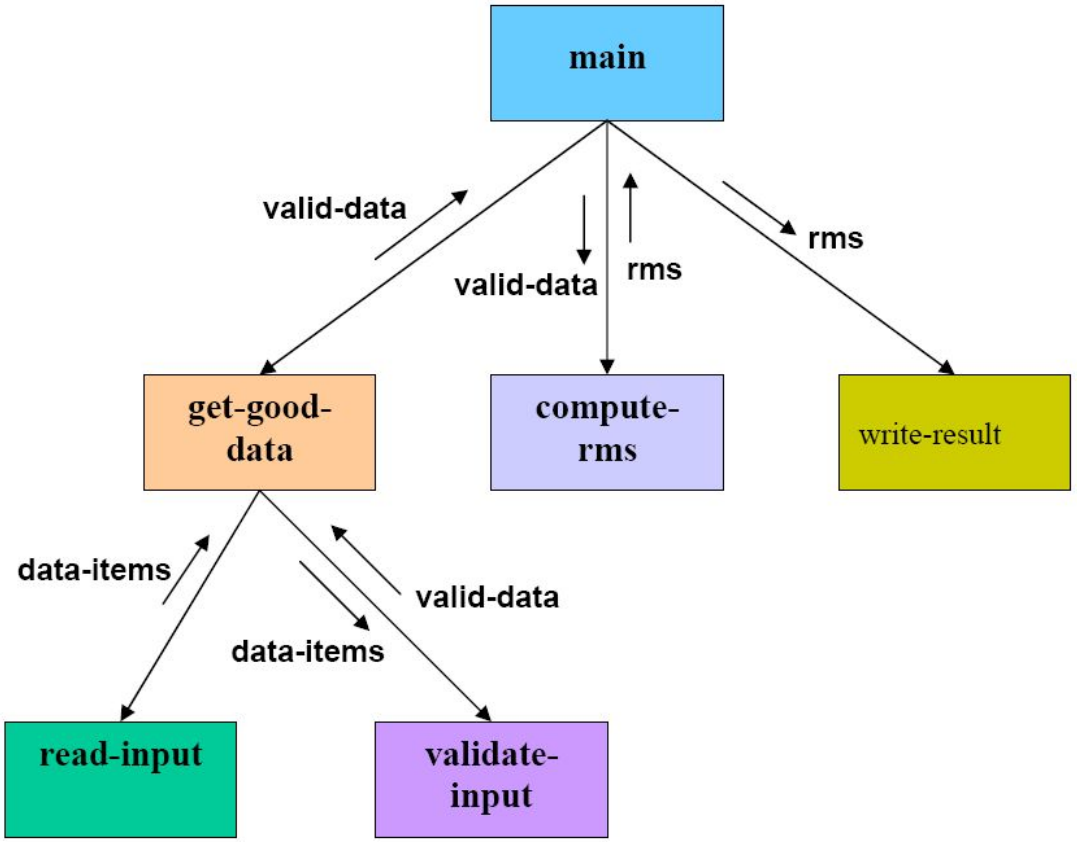
Vertical Partitioning

- Control and work modules are distributed top down
- Top level modules perform control functions
- Lower modules perform computations (input processing and output



Design Concepts (Structural partitioning)

Eg of Horizontal Partitioning



Design Concepts (Structural partitioning)

Eg of Vertical Partitioning

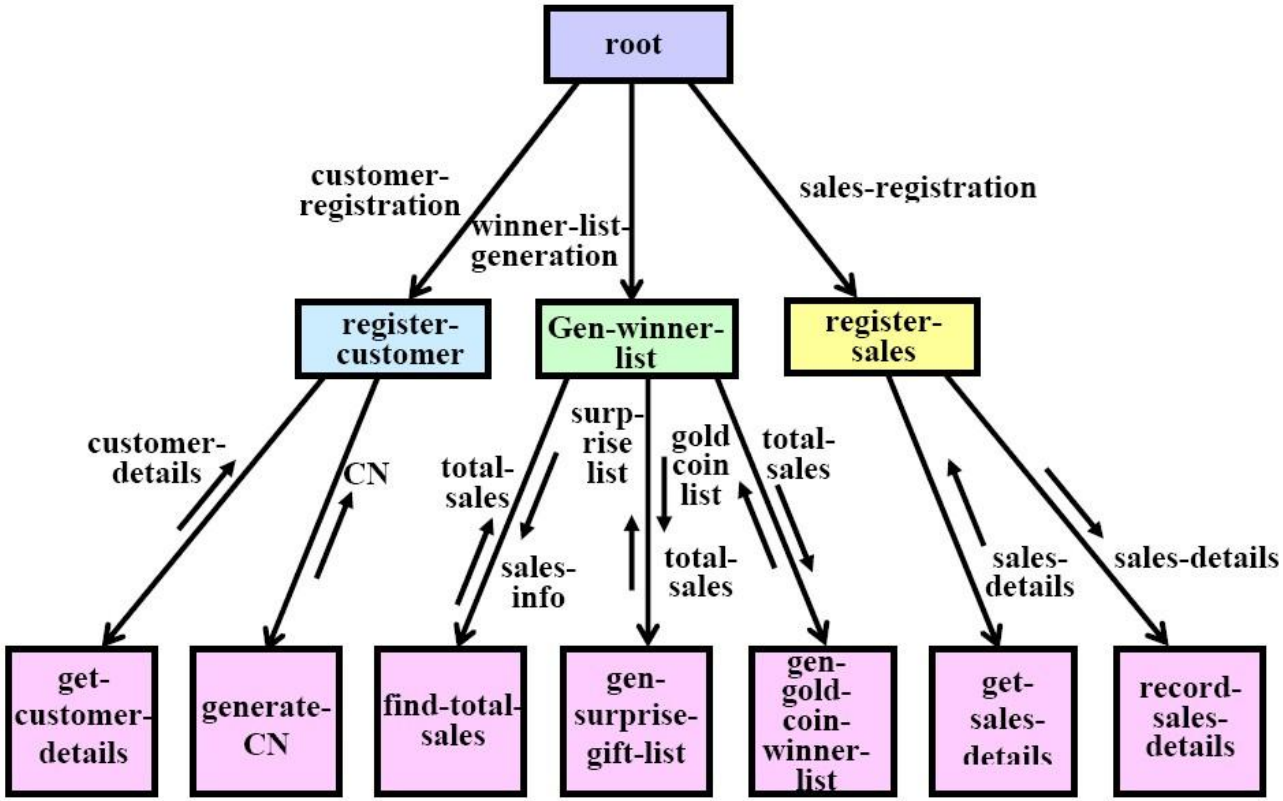


Fig. 36.13 Structure chart for the supermarket prize scheme

Functional Independence

- Functional independence is achieved by developing modules with “single minded” function and an aversion to excessive interaction with other modules.

- Measured using 2 qualitative criteria:

5. Cohesion : Measure of the relative strength of a module.
6. Coupling : Measure of the relative interdependence among modules.

Cohesion

- Strength of relation of elements within a module
- Element- Group of instructions or a data definition.
- Strive for high cohesion

Different types of cohesion:

- Functional Cohesion (Highest):

A functionally cohesive module contains elements that all contribute to the execution of one and only one problem-related task.
Examples of functionally cohesive modules are

Compute cosine of an angle

Calculate net employee salary

Cohesion

2. Sequential Cohesion :

A *sequentially cohesive* module is one whose elements are involved in activities such that output data from one activity serves as input data to the next.

Eg: Module read and validate customer record

- Read record
- Validate customer record

Here output of one activity is input to the second

Cohesion

3. Communicational cohesion

A *communicationally cohesive* module is one whose elements contribute to activities that use the same input or output data.

Suppose we wish to find out some facts about a book

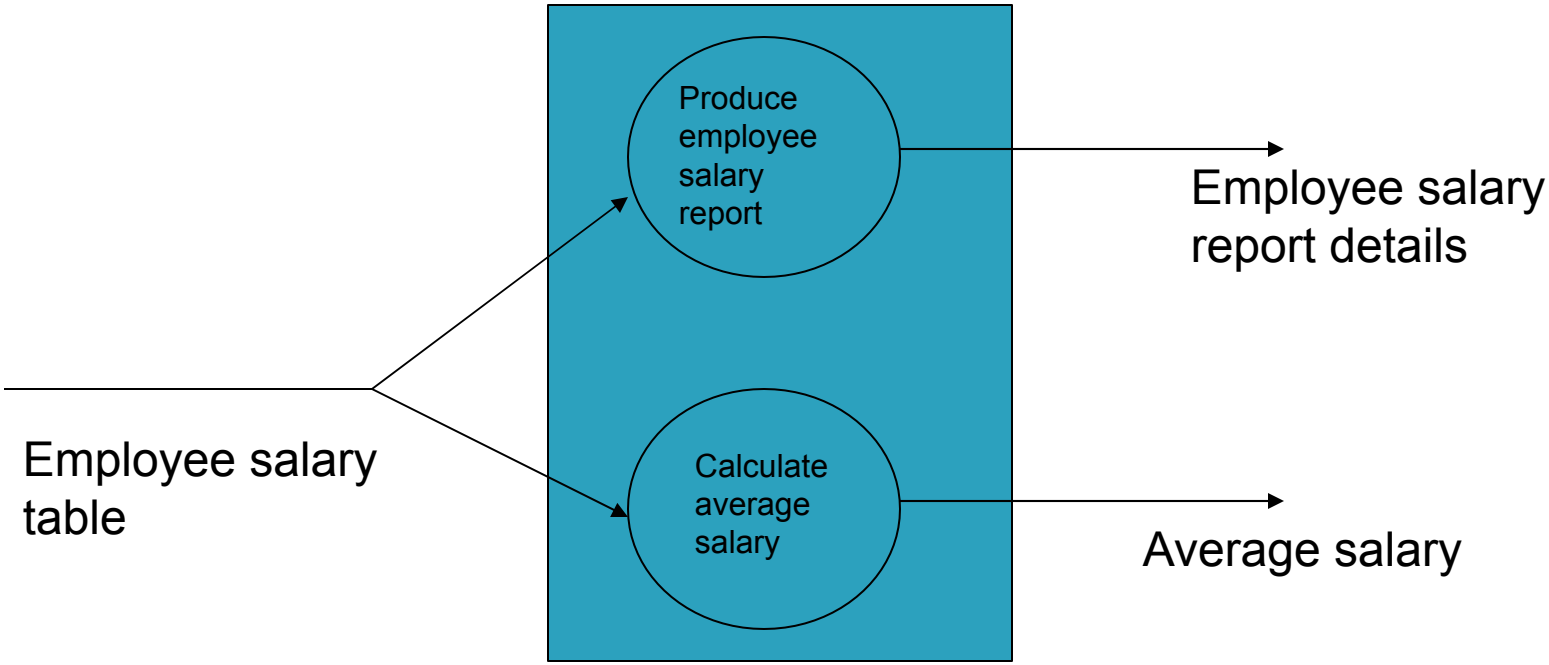
For instance, we may wish to

- FIND TITLE OF BOOK
- FIND PRICE OF BOOK
- FIND PUBLISHER OF BOOK
- FIND AUTHOR OF BOOK

These four activities are related because they all work on the same input data, the book, which makes the "module" communicationally cohesive.

Cohesion

Eg: module which produces employee salary report and calculates average salary



Cohesion

4. Procedural Cohesion

A procedurally cohesive module is one whose elements are involved in different and possibly unrelated activities in which control flows from each activity to the next.

A piece of coding ties the activities together Eg:

Module that averages two completely unrelated tables, **TABLE- A** and **TABLE-B**, which both just happen to have 100 elements each.

Cohesion

```
module compute table-A-avg and table-B-avg  uses
        table-A, table-B
        returns table-A-avg, table-B-avg
           table-A-total : = 0
           table-B-total : = 0
           for i = i to 100
              add table-A (i) to table-A-total  add
              table-B (i) to table-B-total
           endfor
           table-A-avg : = table-A-total/100
           table-B-avg : = table-B-total/100
     endmodule
```

Cohesion

5. Temporal Cohesion

A temporally cohesive module is one whose elements are involved in activities that are related in time.

A module **INITIALIZE** initializes many different functions in a mighty sweep, causing it to be broadly related to several other modules in the system.

module initialize

updates a-counter, b-counter, items table, totals table, switch-a, switch-b **r**
ew ind tape-a
set a-counter **to** 0
rew ind tape-b **set** b-counter **to** 0
cle ar items table
cle ar totals table
set switch-a **to off**
set switch-b **to on**
e ndmodule

Cohesion

6. Logical Cohesion

A logically cohesive module is one whose elements contribute to activities of the same general category in which the activity or activities to be executed are selected from outside the module.

A logically cohesive module contains a number of activities of the same general kind.

To use the module, we pick out just the piece(s) we need.

Cohesion

Eg. A module maintain employee master file

Uses input flag/* to choose which function*/

If **input flag =1**

{Add a new record to master file} If

input flag =2

{delete an employee record } If

input flag=3

{edit employee record }

-

-

Endmodule

Cohesion

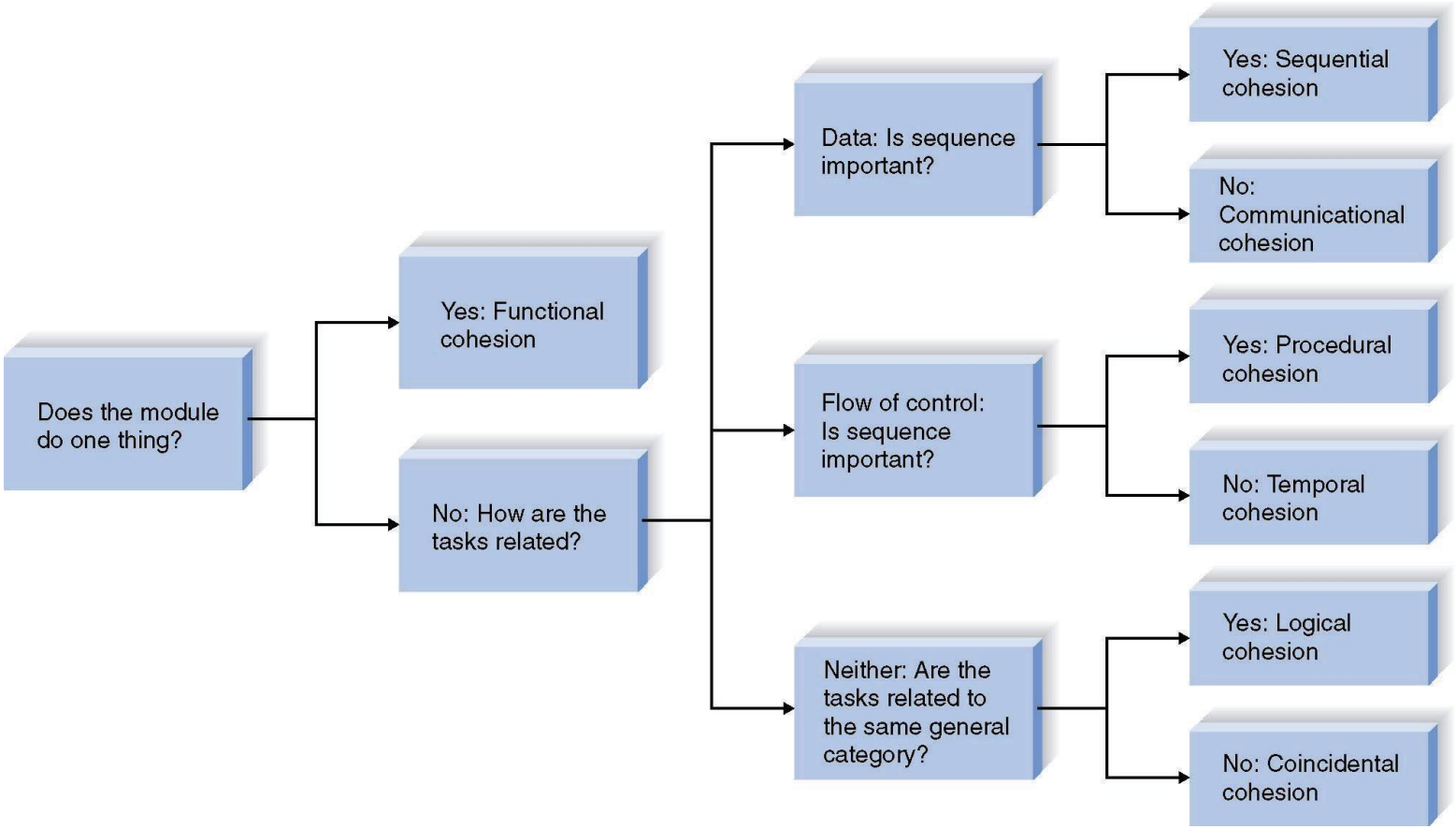
7. Coincidental Cohesion (Lowest)

A coincidentally cohesive module is one whose elements contribute to activities with no meaningful relationship to one another.

Eg. When a large program id modularized by arbitrarily segmenting the program into several small modules.

Cohesion

A decision tree to show module cohesion



Coupling

- Measure of interconnection among modules in a software structure
- Strive for lowest coupling possible.

Coupling

Types of coupling

1. Data coupling (Mo st desirable)

Two modules are data coupled, if they communicate through parameters where each parameter is an elementary piece of data.

e.g. an integer, a float, a character, etc. This data item should be problem related and not be used for control purpose. **2.**

Stamp Co upling

Two modules are said to be stamp coupled if a data structure is passed as parameter but the called module operates on some but not all of the individual components of the data structure.

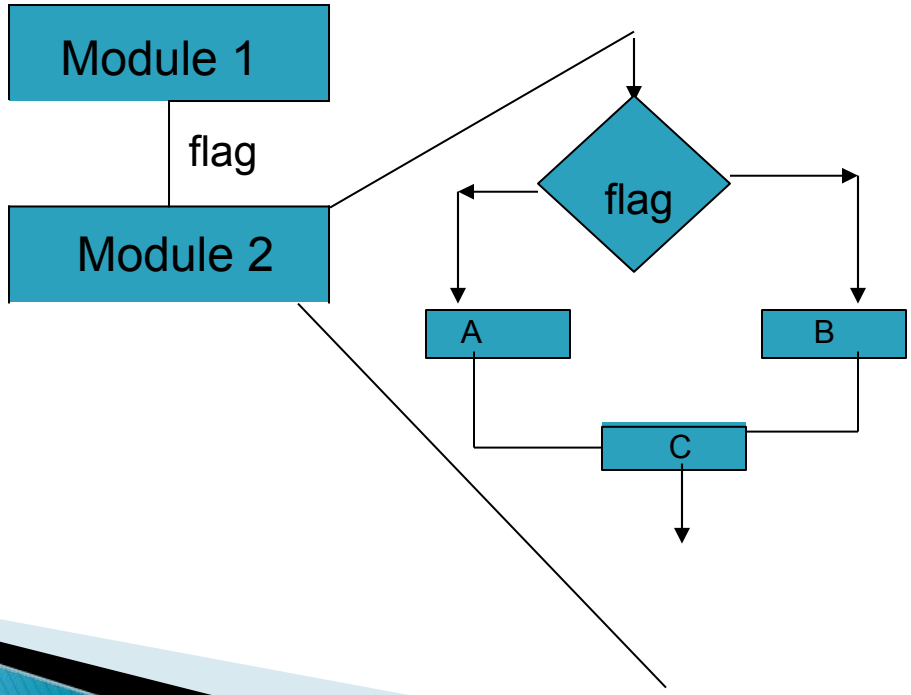
Coupling

3. Control Coupling

Two modules are said to be control coupled if one module passes a control element to the other module.

This control element affects /controls the internal logic of the called module

Eg: flags



Coupling

4. Common Coupling
Takes place when a number of modules access a data item in a global data area.

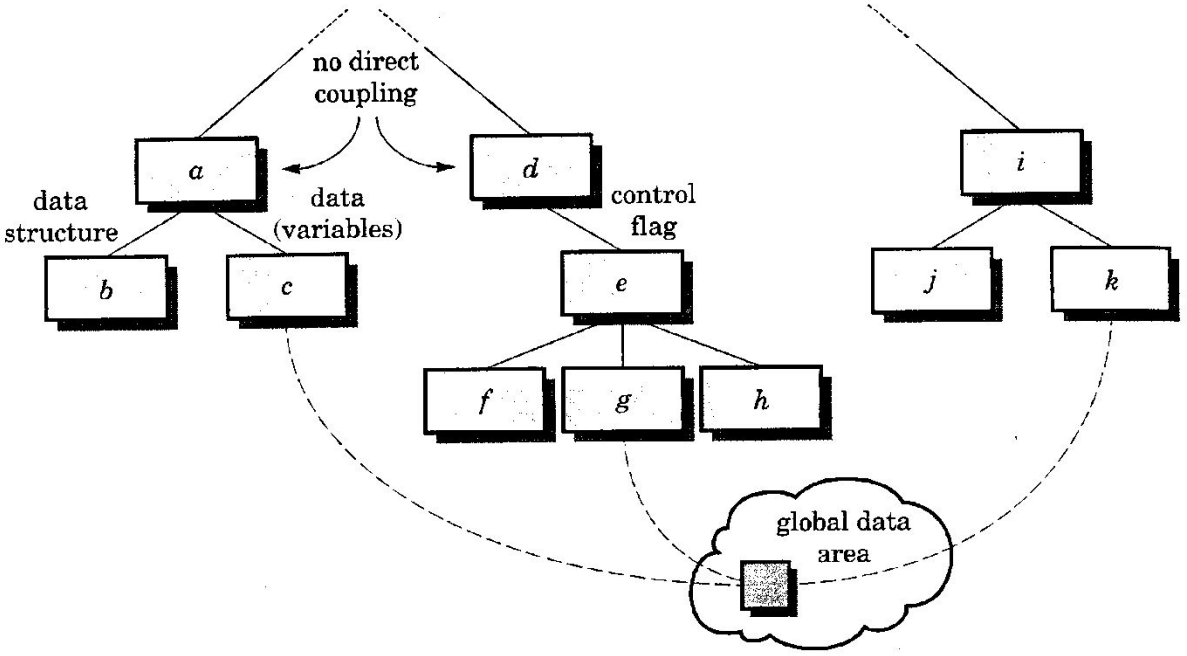


FIGURE 13.8. Types of coupling

Modules c, g and k exhibit common coupling

Coupling

5. Content coupling (Least desirable)
Two modules are said to be content coupled if one module branches into another module or modifies data within another. Eg:

```
int func1(int a)
{ printf("func1");
  a+=2;
  goto F2A;
return a;
}

void func2(void)
{ printf("func2");
F2A : printf("At F2A")
}
```

Design heuristics for effective modularity

- Evaluate 1st iteration to reduce coupling & improve cohesion
- Minimize structures with high fan-out
- Keep scope of effect of a module within scope of control of that module

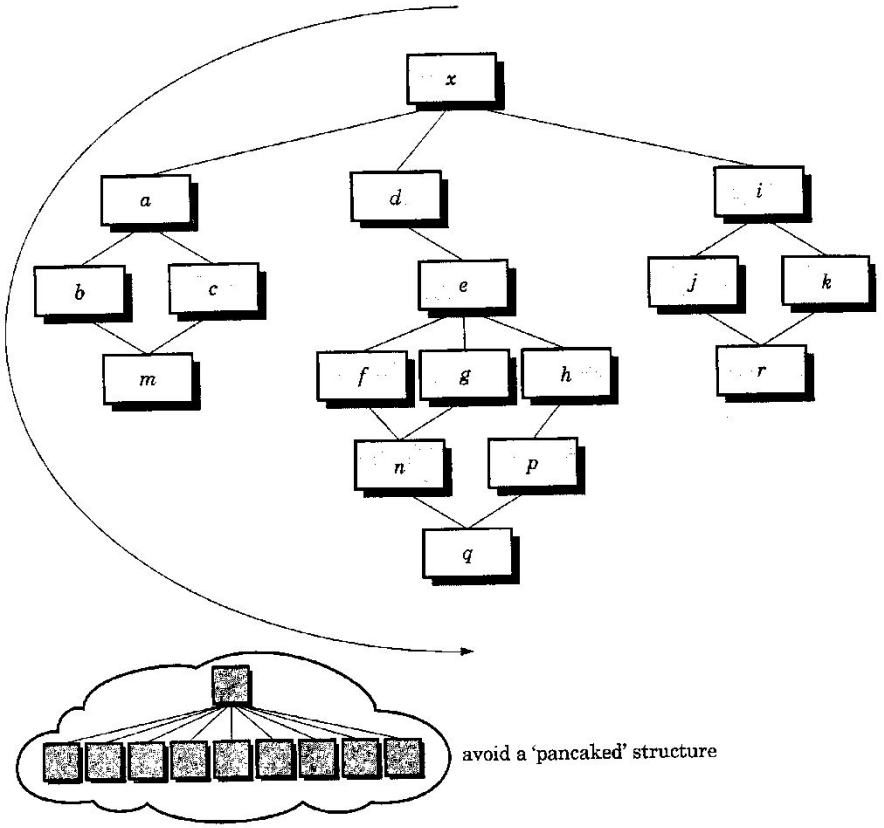


FIGURE 13.9. Program structures

Design heuristics for effective modularity

- Evaluate interfaces to reduce complexity
- Define modules with predictable function A module is predictable when it can be treated as a black box.
- Strive for controlled entry -- no jumps into the middle of things

Design principles:

- The design process should not suffer from ‘tunnel vision’ A good designer should consider alternative approaches.
- The design should be traceable to the analysis model
- The design should not reinvent the wheel
  Time is short and resources are limited! Hence use well tested and reusable s/w components
- The design should ‘minimize the intellectual distance” between the S/W and the problem as it exists in the real world
  That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
- The design should exhibit uniformity & integration
  A design is uniform if it appears that one person developed the whole thing.

Design principles:

- The design should be structured to accommodate change

  Upcoming modifications and improvements should not lead to drastic changes or redesign of software
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered

  Use message/progress bars whenever possible
- Design is not coding, coding is not design
- The design should be assessed for quality as it is being created, not after the fact
- The design should be reviewed to minimize conceptual (semantic) errors