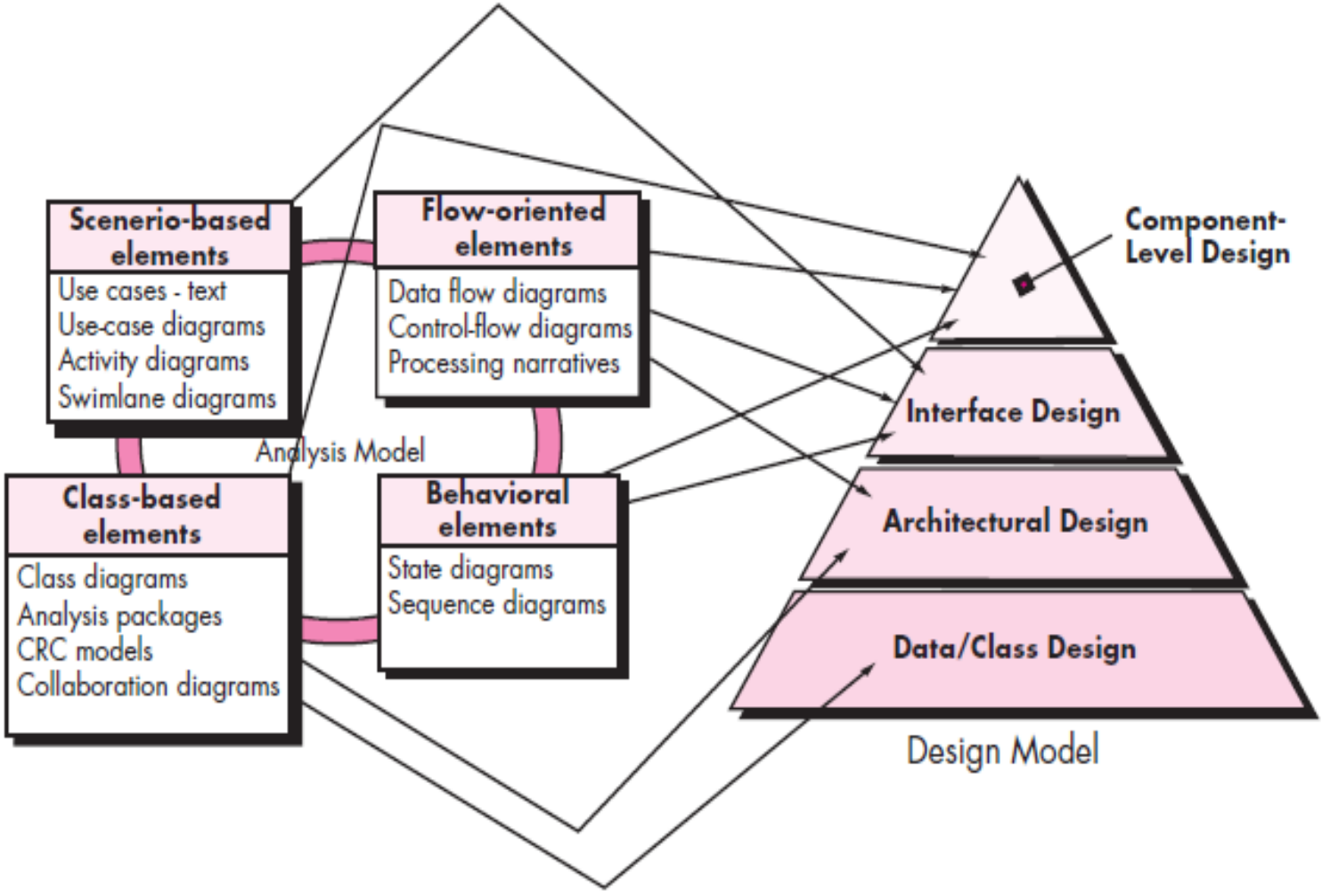
DESIGN ENGINEERING

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.
- Component level Design
  - Transforms structural elements of the architecture into a procedural description of software components


Scenerio-based elements
Use cases - text
Use-case diagrams
Activity diagrams
Swimlane diagrams
Flow-oriented elements
Data flow diagrams
Control-flow diagrams
Processing narratives
Analysis Model
Class-based elements
Class diagrams
Analysis packages
CRC models
Collaboration diagrams
Behavioral elements
State diagrams
Sequence diagrams
Component-Level Design
Interface Design
Architectural Design
Data/Class Design
Design Model

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

Goals of the design process:

1. 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.
2. The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
3. The design should address the data, functional and behavioral domains from an implementation perspective

Quality Guidelines

1. A design should exhibit an architecture that

(1) has been created using recognizable architectural styles or patterns,

(2) is composed of components that exhibit good design characteristics

(3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems

3. A design should contain distinct representations of data, architecture, interfaces, and components.

4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.

5. A design should lead to components that exhibit independent functional characteristics.

6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.

7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

8. A design should be represented using a notation that effectively communicates its meaning

Quality attributes

- Functionality – ability of the system to do the work for which it was intended
- Usability – ease of use (user friendliness)
- Reliability – failure free software operation
- Performance - responsiveness
- Supportability – identifying and resolving issue when software fails

Design Attribute for a Good Software

- Cohesion
- Coupling
- Layering
- Abstraction
- Fan-in - Is the number of modules that call a given module.
- Fan-out - Is the numbers of modules that called by a given module.
- Scope of re-use
- Error of isolation
- Clear Decomposition
- Modularity - refers to the extent to which a software may be divided into smaller modules
- Correctness, maintainability, understandability and efficiency

Design Concepts

Abstraction

- At the highest level of abstraction a solution is stated in broad terms
- At lower levels of abstraction, a more detailed description of the solution is provided.

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

- *Software architecture* refers to “the overall structure of the software and the ways in which that structure provides conceptual integrity for a system”
- 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.
- **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, reliability, security, adaptability, and other system characteristics.
- **Families of related systems.** The design should have the ability to reuse architectural building blocks.

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

Design Concepts(Modularity)

Five 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)

Modular Composability

Design method should enable reuse of existing components.

Modular Decomposability

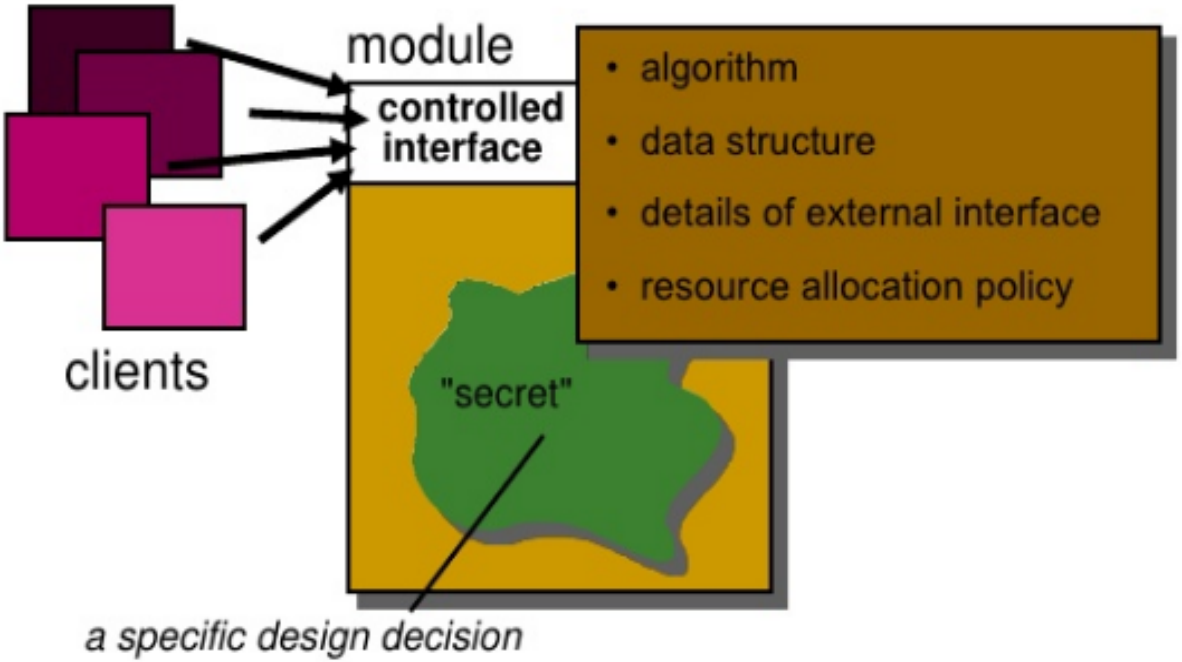
Complexity of the overall problem can be reduced if the design method provides a systematic mechanism to decompose a problem into sub problems

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.

Information hiding



Design Concepts

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:

1. Cohesion : Measure of the relative strength of a module.
2. Coupling : Measure of the relative interdependence among modules.

Cohesion

- Definition
  - The degree to which all elements of a component are directed towards a single task.
  - The degree to which all elements directed towards a task are contained in a single component.
  - The degree to which all responsibilities of a single class are related.
- All elements of component are directed toward and essential for performing the same task.

Cohesion

Different types of cohesion :

1. 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

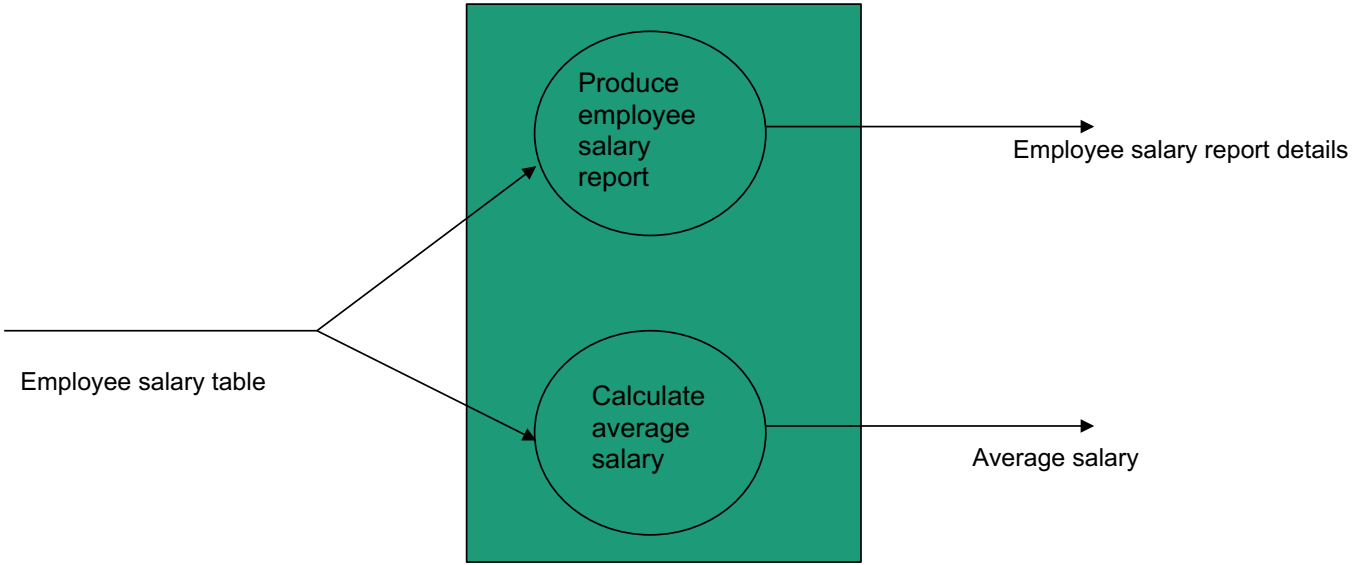
For instance, we may wish to

- 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.

Eg:

addRecord

changeRecord

deleteRecord

Cohesion

5. Temporal Cohesion

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

Eg:

An exception handler that

- Closes all open files
- Creates an error log
- Notifies user

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.

Eg:

Read from file

Read from database

Cohesion

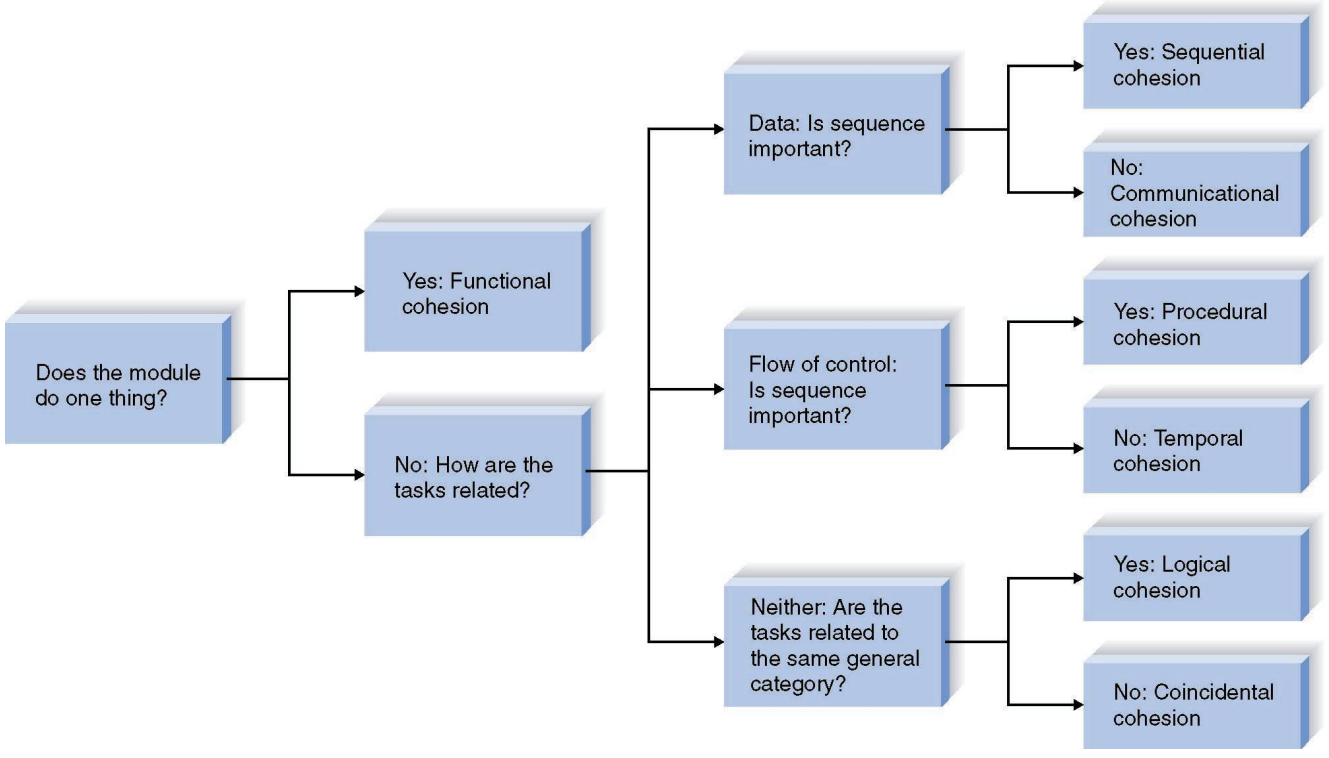
7. Coincidental Cohesion (Lowest)

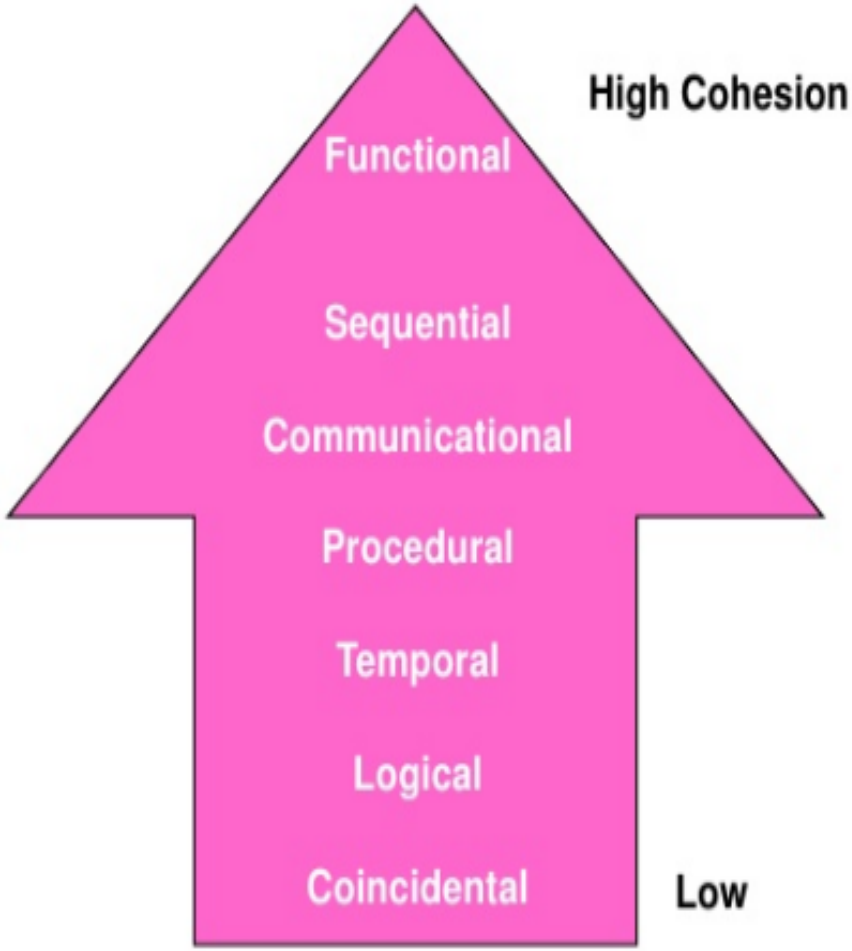
A coincidentally cohesive module is one whose elements contribute to activities with no meaningful relationship to one another.

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

Cohesion

A decision tree to show module cohesion




High Cohesion
Functional
Sequential
Communicational
Procedural
Temporal
Logical
Coincidental
Low

Coupling

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

Coupling

Types of coupling

1. Data coupling (Most 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 Coupling

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.

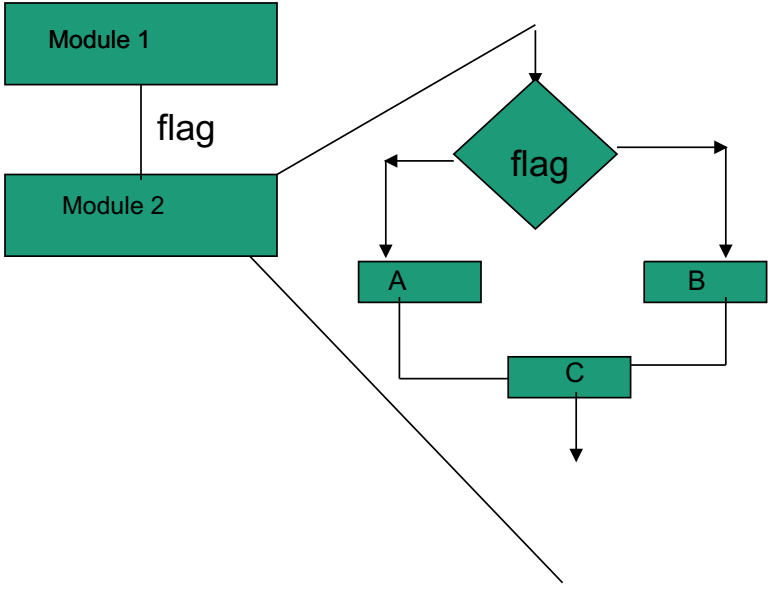
Coupling

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.

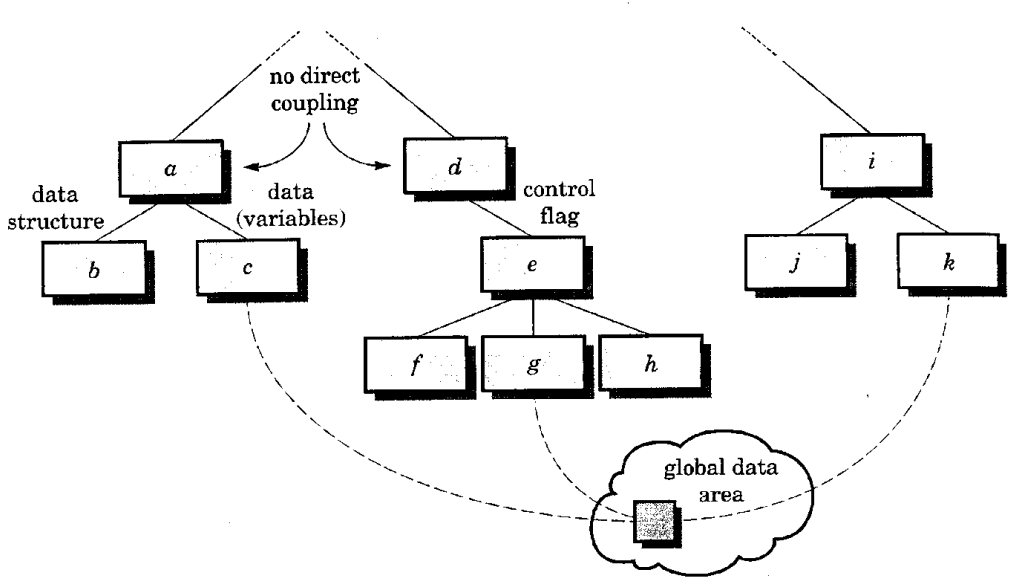


FIGURE 13.8. Types of coupling

Modules c, g and k exhibit common coupling

5. Content coupling (Least desirable)

Coupling

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;
```

- Eg: Part of a program that handles lookup for a customer.

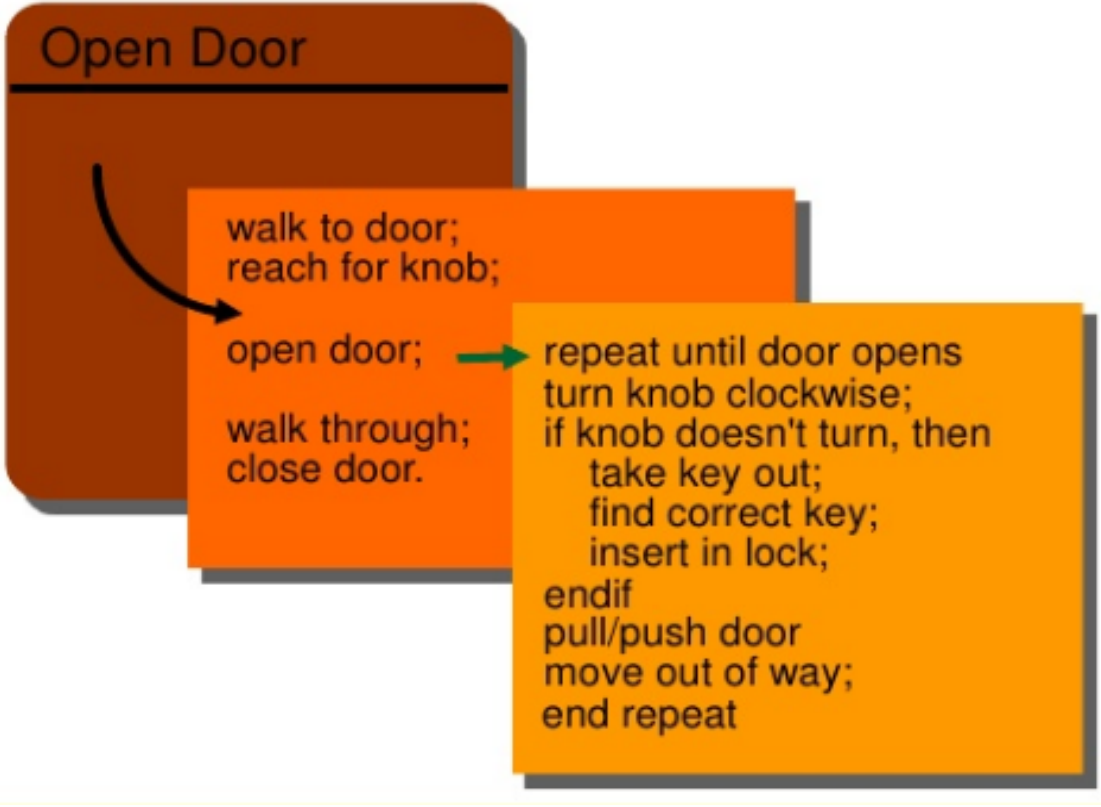
When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data.

Design Concepts

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.

Stepwise refinement



Design Concepts

Refactoring

- Reorganization technique that simplifies the design (or code) of a component without changing its function or behavior

- "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

- When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.
- For example, a first design iteration might yield a component that exhibits low cohesion (i.e., it performs three functions that have only limited relationship to one another). After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion

Design Concepts

Design Classes

- As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented and implement a software infrastructure that supports the business solution.

- *User interface classes*
- define all abstractions that are necessary for human-computer interaction (HCI). In many cases, HCI occurs within the context of a *metaphor* (e.g., a checkbook, an order form) and the design classes for the interface may be visual representations of the elements of the metaphor.

- *Business domain classes*
- are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.
- *Process classes*
- implement lower-level business abstractions required to fully manage the business domain classes.

- *Persistent classes*
- represent data stores (e.g., a database) that will persist beyond the execution of the software.

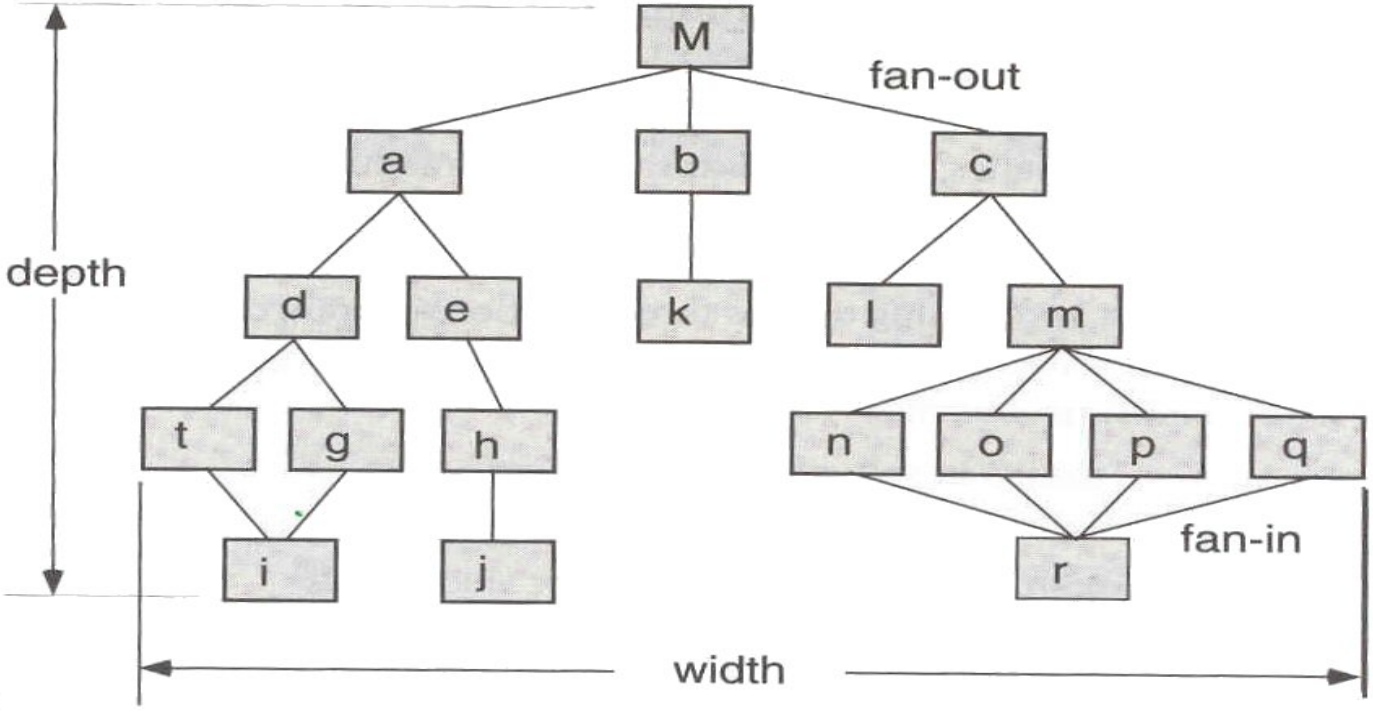
- *System classes*
- implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world

Four characteristics of well-formed design class:

- Complete and sufficient
- Primitiveness (easier to understand, reusable)
- High cohesion
- Low coupling

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)
- **Width**- 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

Design Concepts(Control Hierarchy)

Super ordinate -module that control another module

Subordinate - module controlled by another

Design Concepts

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 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

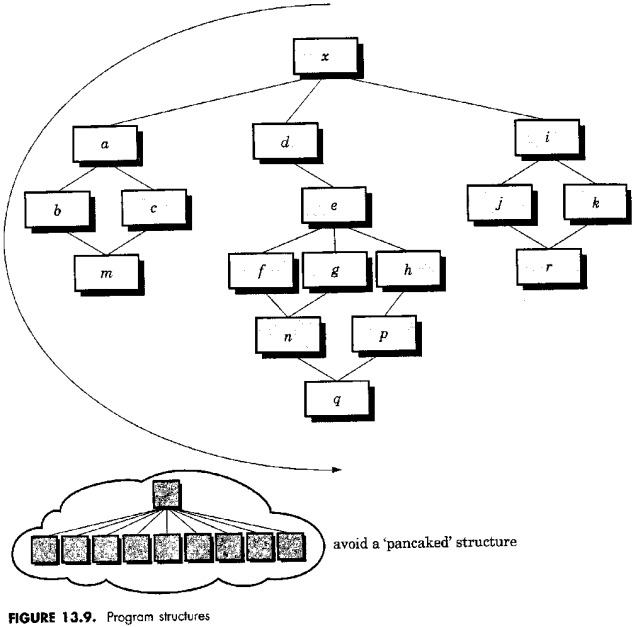


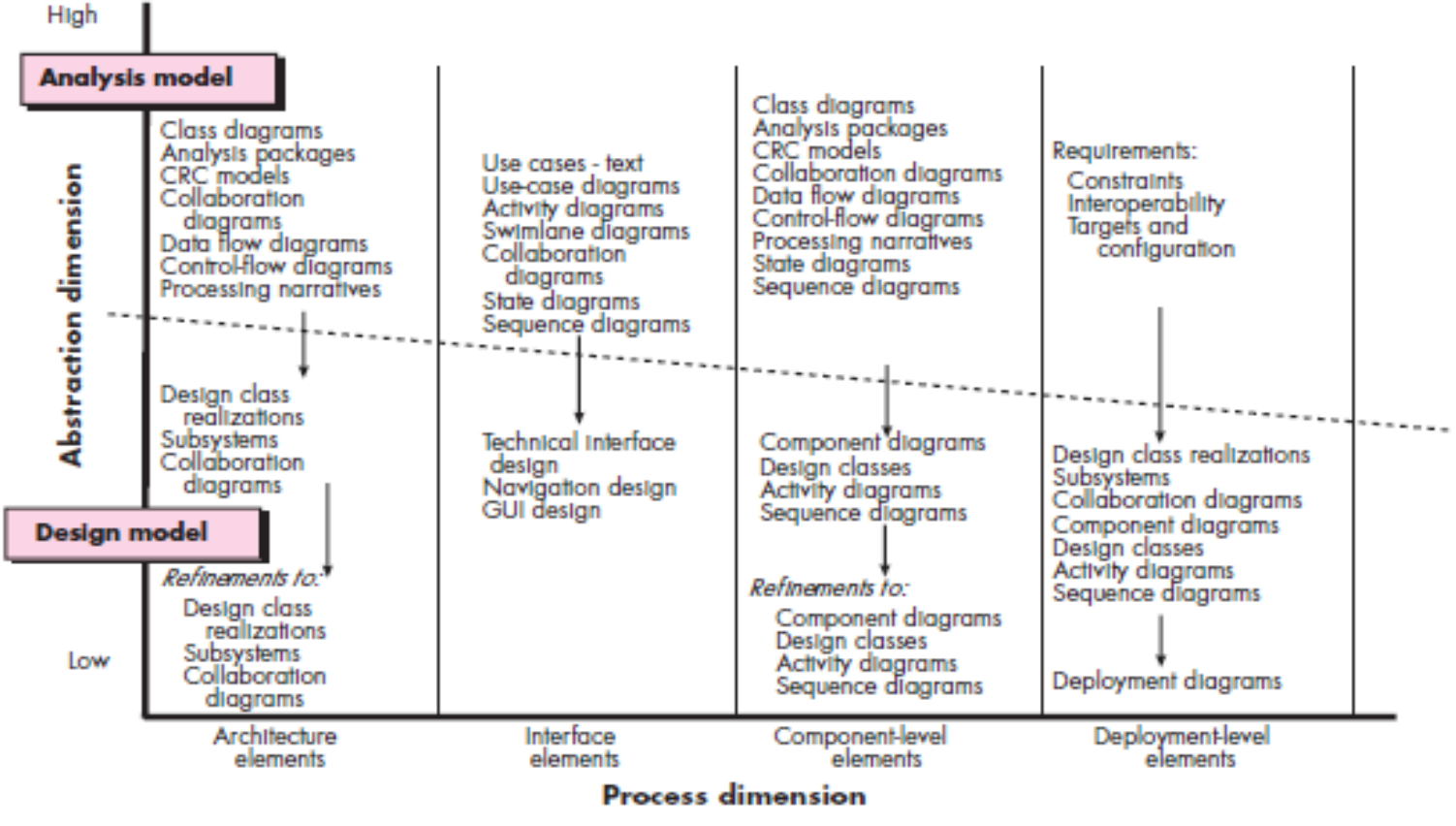
FIGURE 13.9. Program structures

Design heuristics for effective modularity

- Evaluate interfaces to reduce complexity
- Define modules with predictable function
- Strive for controlled entry -- no jumps into the middle

Design Model

FIGURE 8.4 Dimensions of the design model



Data Design Elements

- Like other software engineering activities, data design (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data).
- This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

Architectural Design Elements

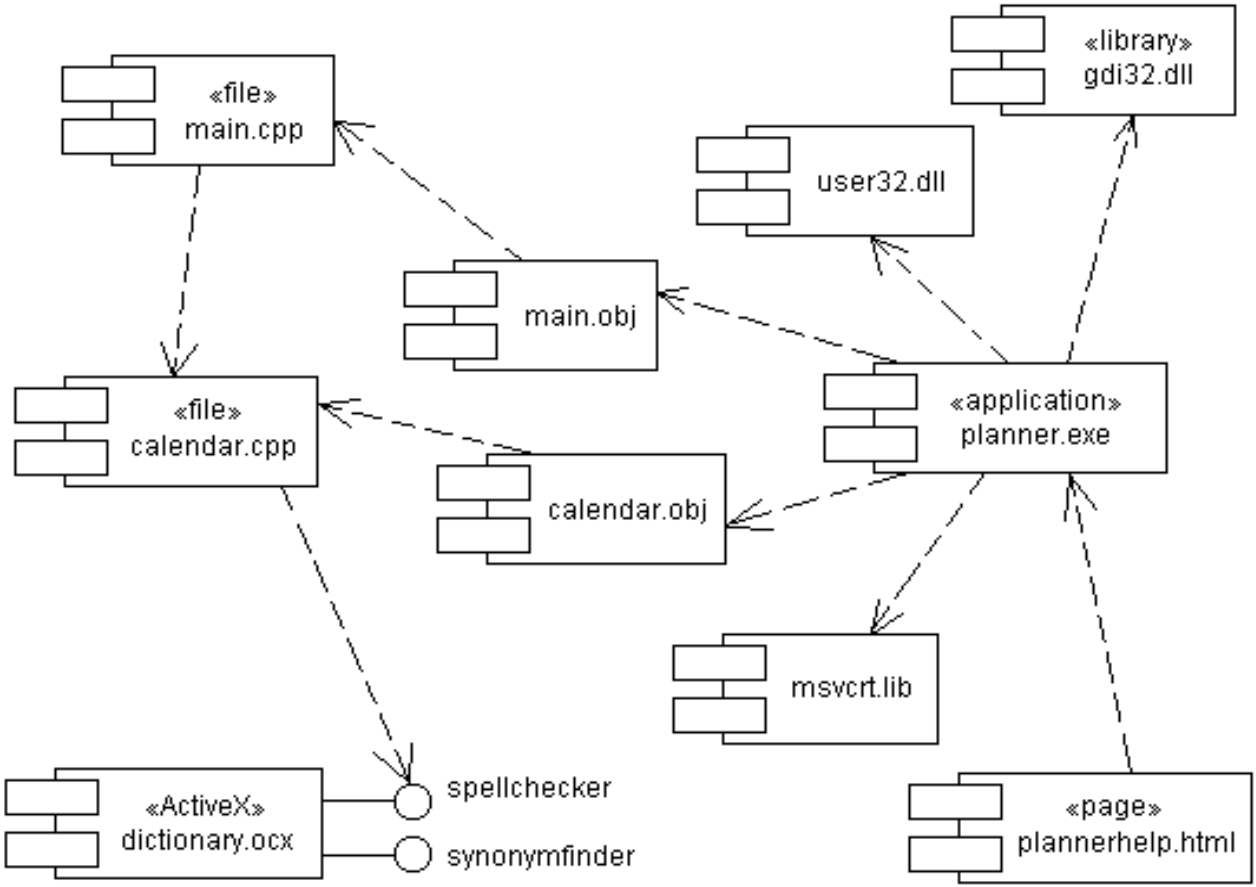
- The *architectural design* for software is the equivalent to the floor plan of a house
- The architectural design element is usually depicted as a set of interconnected subsystems, often derived from analysis packages within the requirements model.
- Each subsystem may have it's own architecture (e.g., a graphical user interface might be structured according to a preexisting architectural style for user interfaces).

Interface Design Elements

- The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.
- There are three important elements of interface design: (1) the user interface (UI); (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and (3) internal interfaces between various design components.
- These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

Component level design elements

- The component-level design for software fully describes the internal detail of each software component.
- To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors).


«file»
main.cpp
«library»
gdi32.dll
user32.dll
main.obj
«file»
calendar.cpp
«application»
planner.exe
calendar.obj
msvcrt.lib
«ActiveX»
dictionary.ocx
spellchecker
synonymfinder
«page»
plannerhelp.html

Deployment level design elements

- Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.