Design Concepts

Design Concerns

- Two issues
  - Where to start?
  - How to represent (design language)?

High-Level Design

- Identification of **modules**
- **Control** relationships between modules
- Definition of **interfaces** between modules

Basic Design Approaches

- **Function oriented** – basic abstractions are functions

Basic Design Approaches

- **Object oriented** – basic abstractions are objects (instantiation of **class**)

How to Represent?

- With a “language”
- DFD (Data Flow Diagram) is one such language to “express” design
  - Used for **functional design approach**

DFD - Basics

DFD

- Represents "flow of data" through a process or a system
  - Focus on data *"movement"* between external entities and processes, and between processes and data stores

Why DFD?

- Provides overview of
  - What data processed by a system
  - Transformations that are performed
  - Which data are stored
  - What results are produced and where they go

Why DFD?

- Provides overview of
  - What data processed by a system
  - Transformations that are performed
  - Which data are stored
  - What results are produced and where they go
- Graphical nature makes it a good communication tool between
  - User and (system) designer
  - Designer and developer

Components of DFD

- Source/Sinks (external entity)
- Processes
- Data stores
- Data flows

Component Representation

Symbol	Symbol 1 (Gane & Sarson)	Symbol 2 (DeMarco & Yourdan)
External entity		
Process		
Data store		
Data flow		

DFD Naming Guidelines

- Process → verb phrase
- Data flow label → noun: name of data
- Data store → noun
- External entity → noun

External Entity

- People or organizations that send data into or receive data from system

External Entity

- People or organizations that send data into or receive data from system
- They either supply or receive data
  - Source – supplies data to system
  - Sink – receives data from system

Process

- Process – series of actions that transform data
- Notation
  - Straight line with incoming arrows → input data flows
  - Straight lines with outgoing arrows → output data flows
  - Labels assigned to data flow

Data Flow

- Data flow - depicts actual flow of data between elements
  - Connects processes, external entities and data stores

Data Flow

- Notation – an arrow with label
  - Generally unidirectional
  - If same data flows in both directions, double-headed arrow can be used

Data Flow Rules

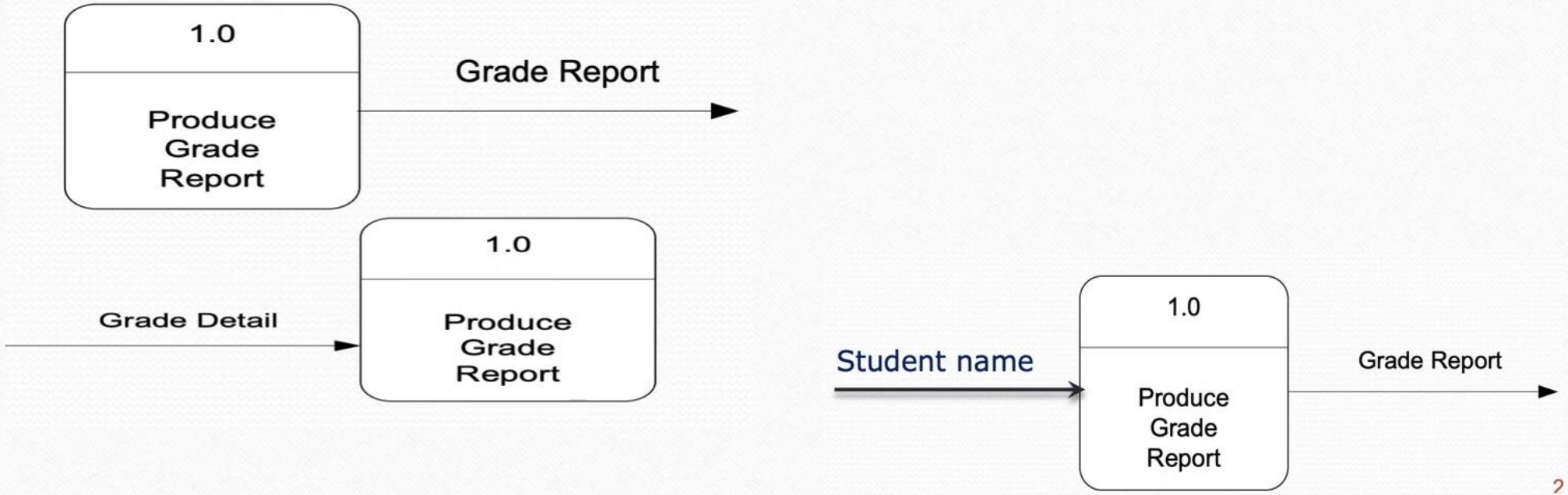
Data can flow from

- ➔External entity to process
- ➔Process to external entity
- ➔Process to store and back
- ➔Process to process

Data can not flow from

- ➔External entity to external entity
- ➔External entity to store
- ➔Store to external entity
- ➔Store to store

3 Incorrect Flow

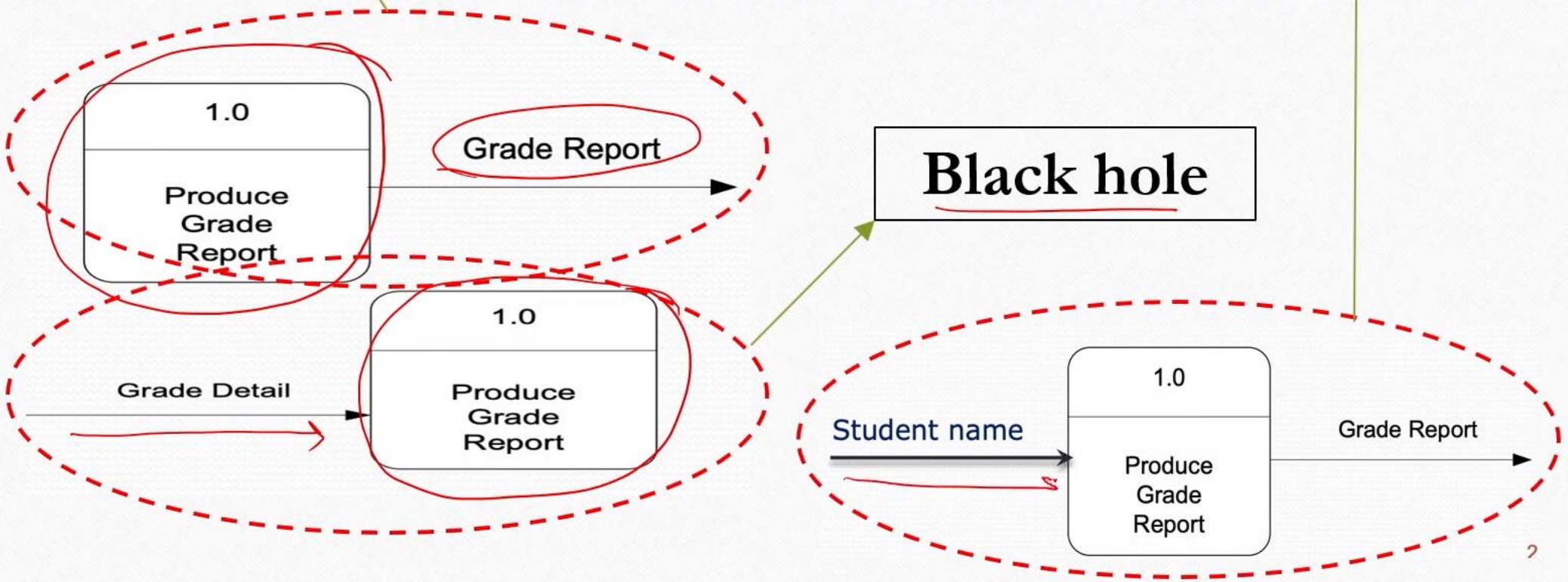


3 Incorrect Flow

Miracle

Gray hole

Black hole



Hierarchy

Representing Complex Designs

- Real-life systems can be very big
  - Involving large number of processes and data stores
- Representing such systems with DFD difficult
  - Very complex diagram
  - Difficult to understand
  - Difficult to modify

Representing Complex Designs

- Way out – decomposition
  - Create hierarchy of “levels”

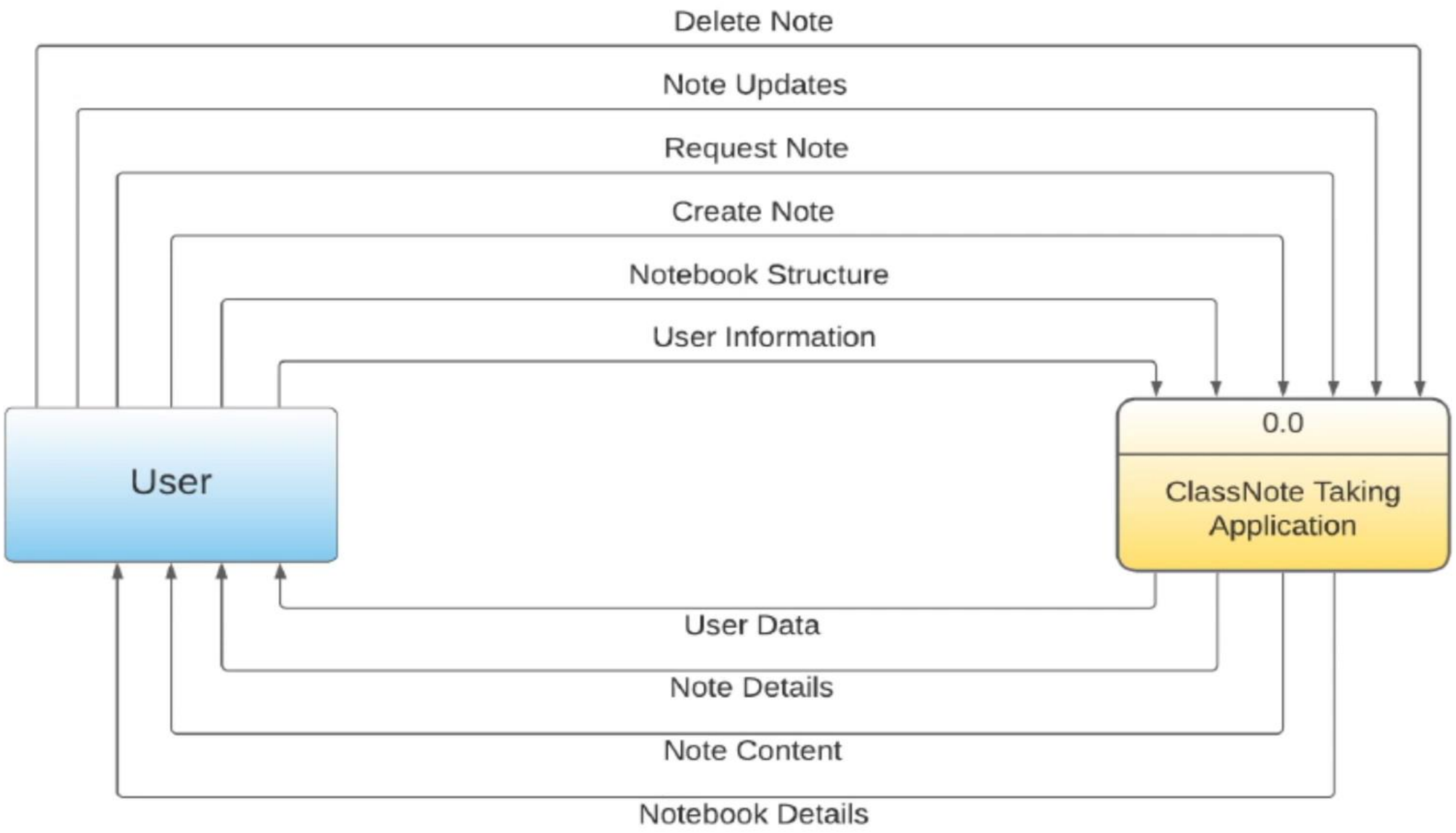
DFD Levels

- At least 3
  - Level 0 – context diagram
  - Level 1 – overview diagram
  - Level 2 – detailed diagram
- There can be further levels, if required

Level 0 (Context Diagram)

- Contains only one process
  - Represents entire system
- Data arrows show input and output (from/to external entities)
- Data stores NOT shown - implicitly contained within system

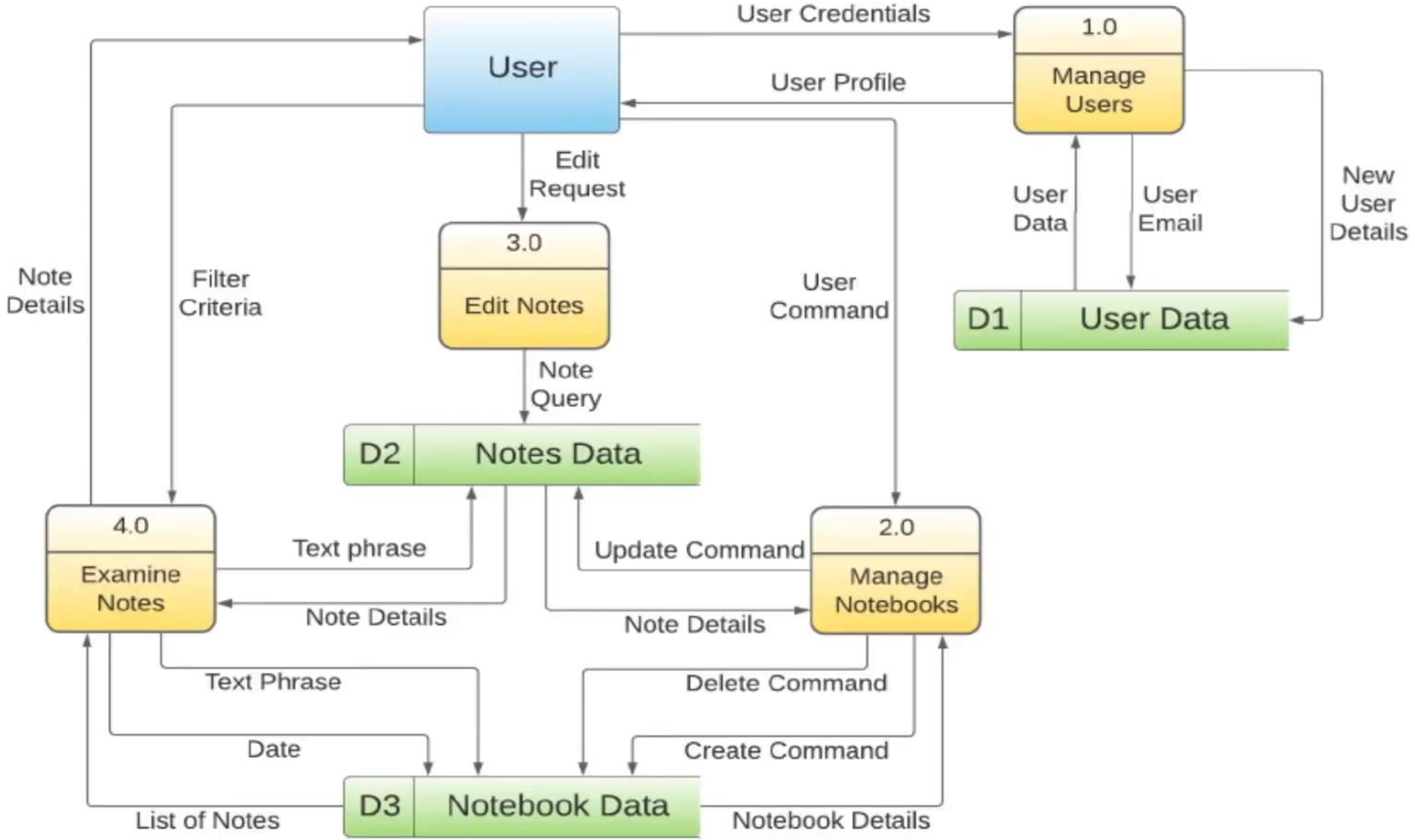
LEVEL 0


Delete Note
Note Updates
Request Note
Create Note
Notebook Structure
User Information
User
0.0
ClassNote Taking Application
User Data
Note Details
Note Content
Notebook Details

Level 1 (Overview Diagram)

- Utilizes all four elements
  - Data stores first shown at this level

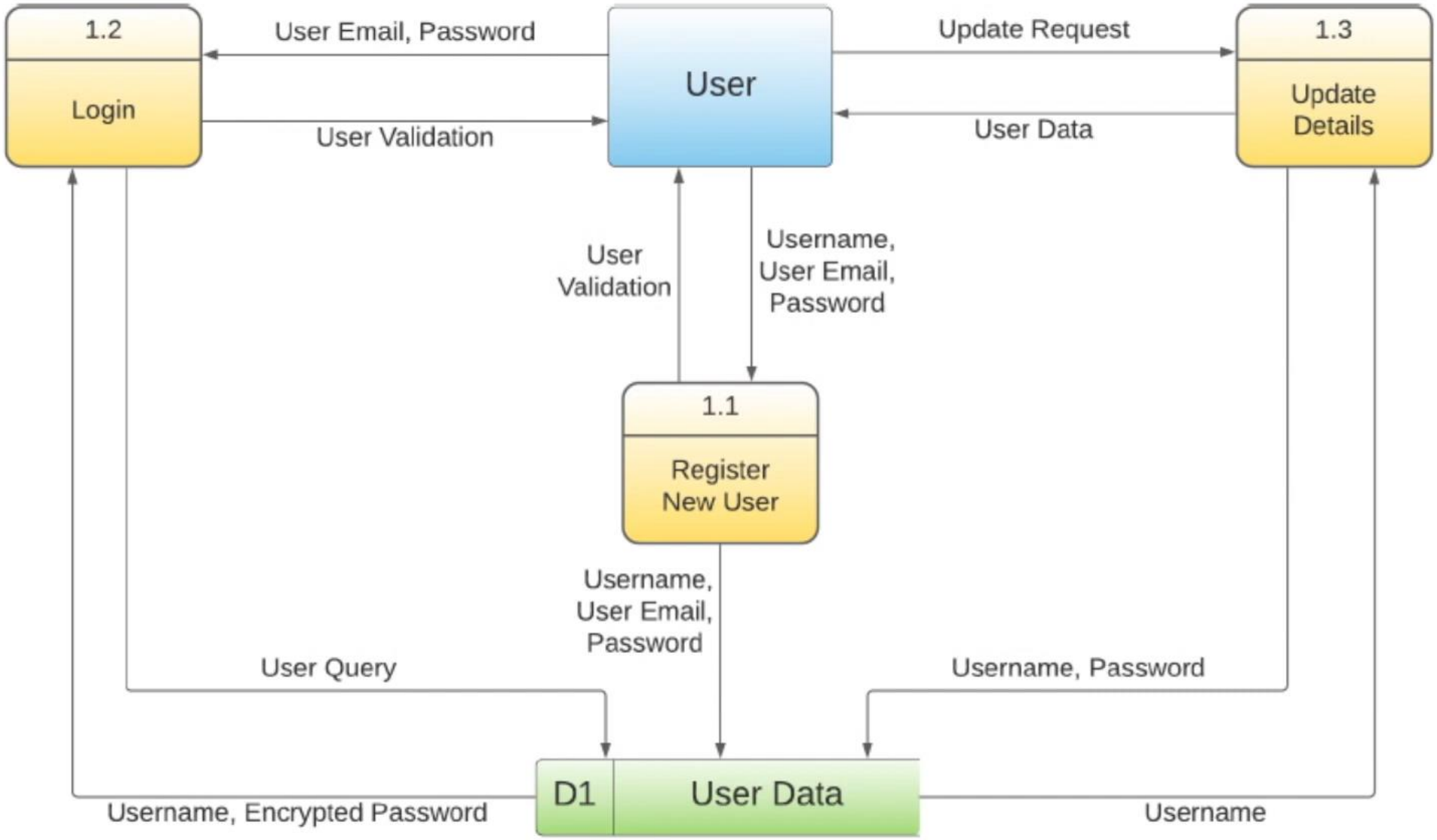
LEVEL 1


User
User Credentials
User Profile
1.0
Manage Users
Edit Request
3.0
Edit Notes
Note Details
Filter Criteria
User Command
User Data
User Email
New User Details
D1
User Data
Note Query
D2
Notes Data
4.0
Examine Notes
Text phrase
Note Details
Update Command
Note Details
2.0
Manage Notebooks
Text Phrase
Date
Delete Command
Create Command
D3
Notebook Data
List of Notes
Notebook Details

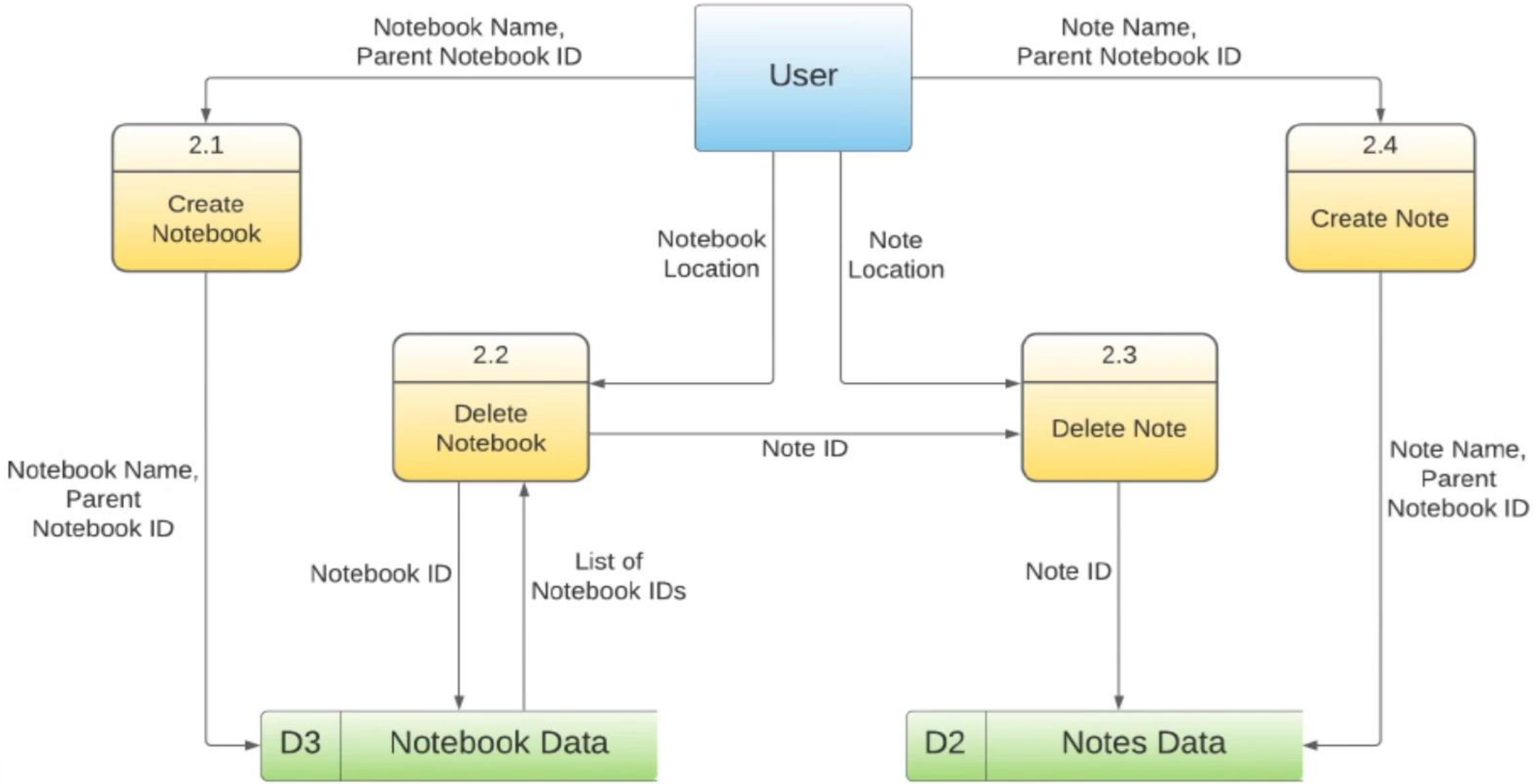
Level 2 (Detailed Diagram)

- Detailed representation of level 1 processes
- Like level 1, utilizes all four elements

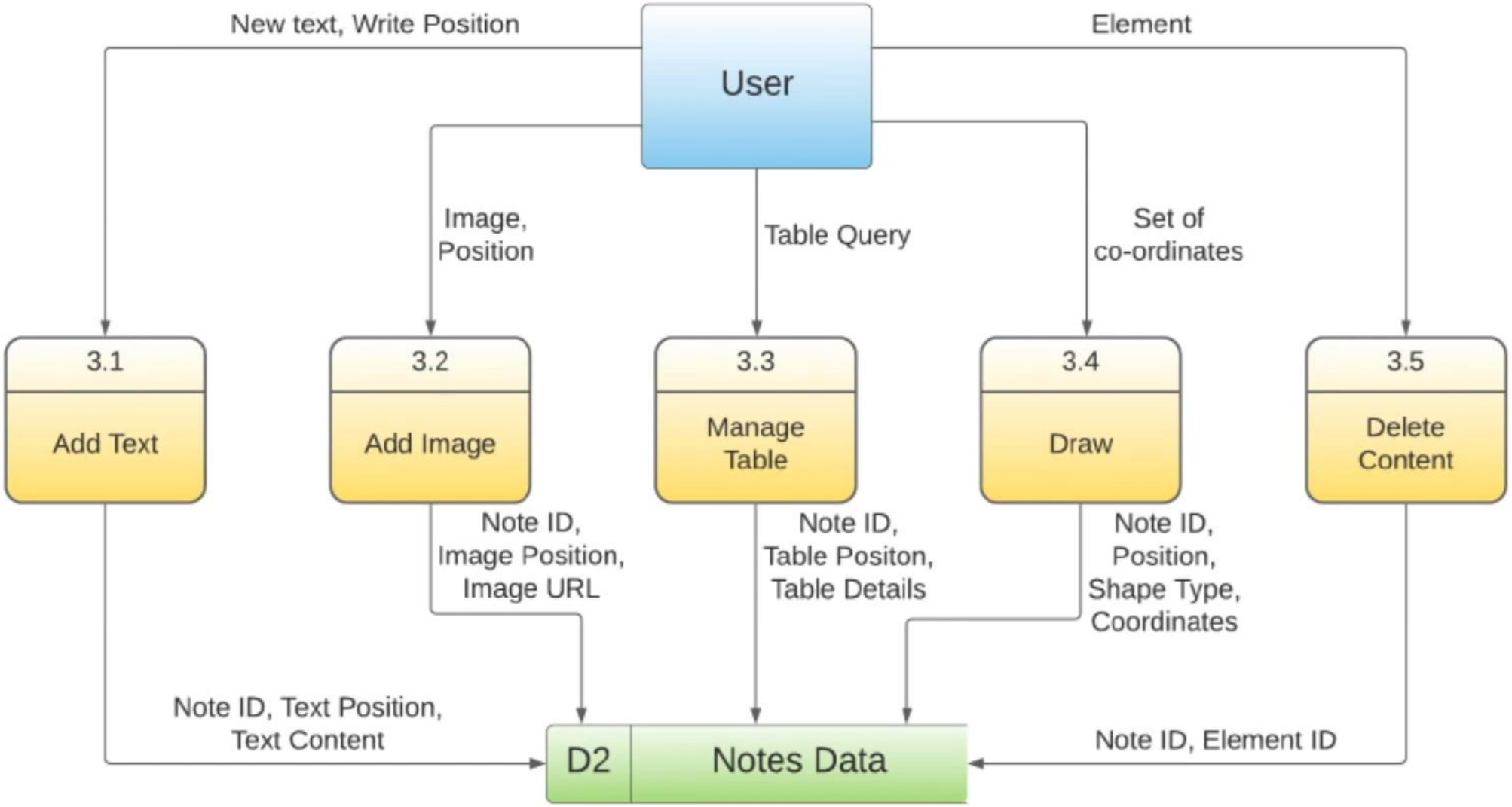
LEVEL 2 , Module 1 : Manage Users



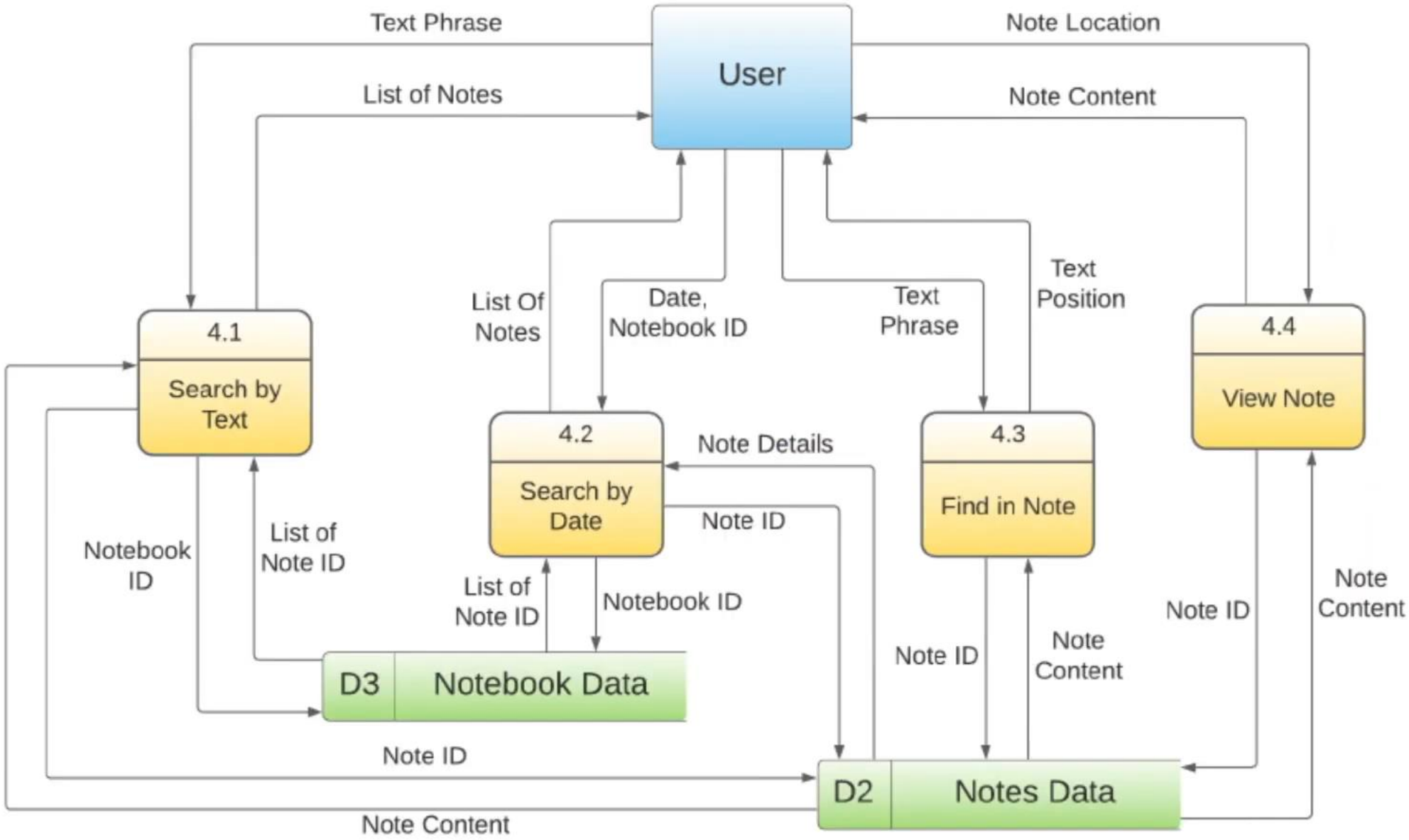
LEVEL 2 , Module 2 : Manage Notebooks



LEVEL 2 , Module 3 : Edit Notes



LEVEL 2 , Module 4 : Examine Notes



Creating Good DFD!

- Use meaningful names for data flows, processes and data stores.
- Use top down development starting from context diagram and successively levelling DFD
- Stop decomposition if levels become trivial (no significant change)
- Remember - only previously stored data can be read
- Remember - data stores cannot create new data

ER Diagram

Representing Data Stores

- In DFD, we have seen data stores
- So far, used only labels to represent those
- Labels don't reveal "structure" and "organization" of data store
- Also there may be relationships between various data elements, not revealed by simple labels

Representing Data Stores

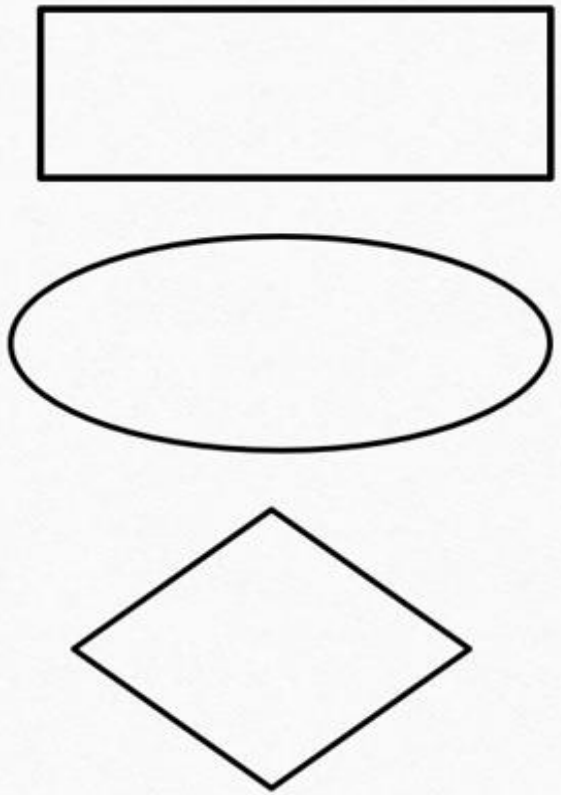
- One way to "express" rich internal structure, organization and relationships in data stores is to use ER (stands for Entity-Relationship) diagrams

E-R Diagram

- Proposed by Peter Chen (1976)
- Includes entities and relationships
- Most common representation for **relational databases**

Basic Components

- **Entity**- an identifiable object or concept of significance
- **Attribute** - property of an entity or relationship
- **Relationship**- an association between entities



Modeling

- A *data store (database)* can be modeled as:
  - A collection of entities
  - Relationship among entities

Entity

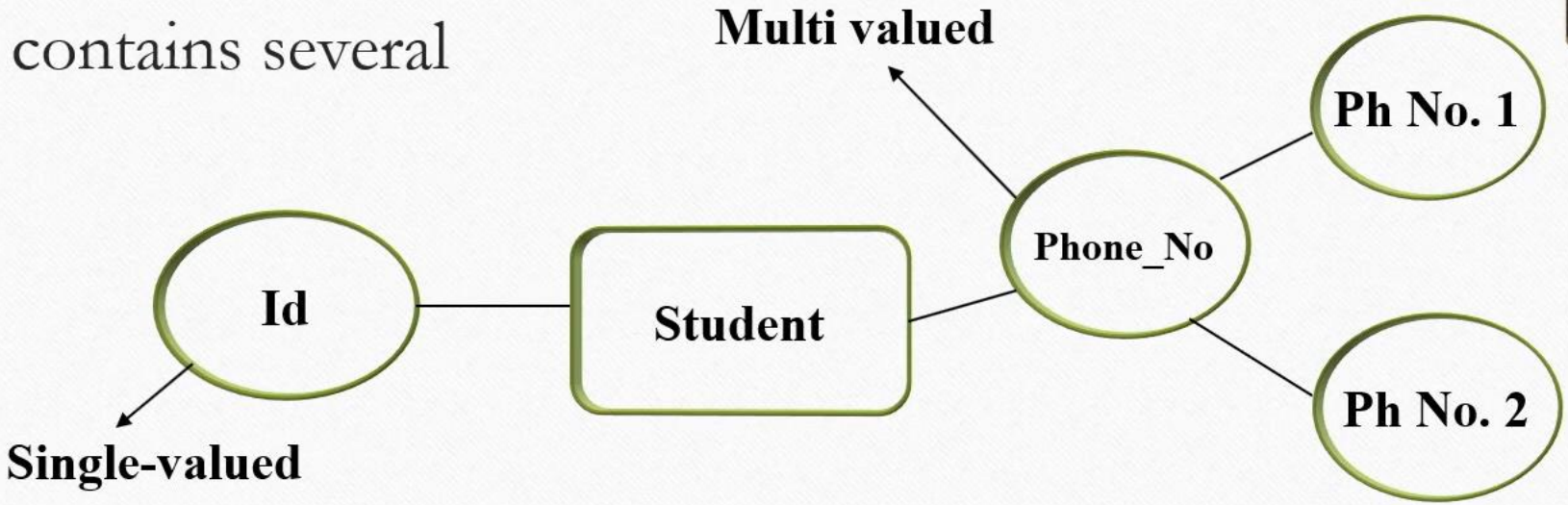
- An object that exists and is distinguishable from other objects (e.g., person, company, student, customer)
- Have *attributes* (e.g., person have *names* and *addresses*)
- **Entity set -** set of same type entities that share same properties
  - E.g. set of all persons
- Each entity must be uniquely identifiable

Attributes

- Descriptive properties of entity
- A particular instance of an attribute is a value
- Domain- set of permitted values of an attribute

Attribute Types

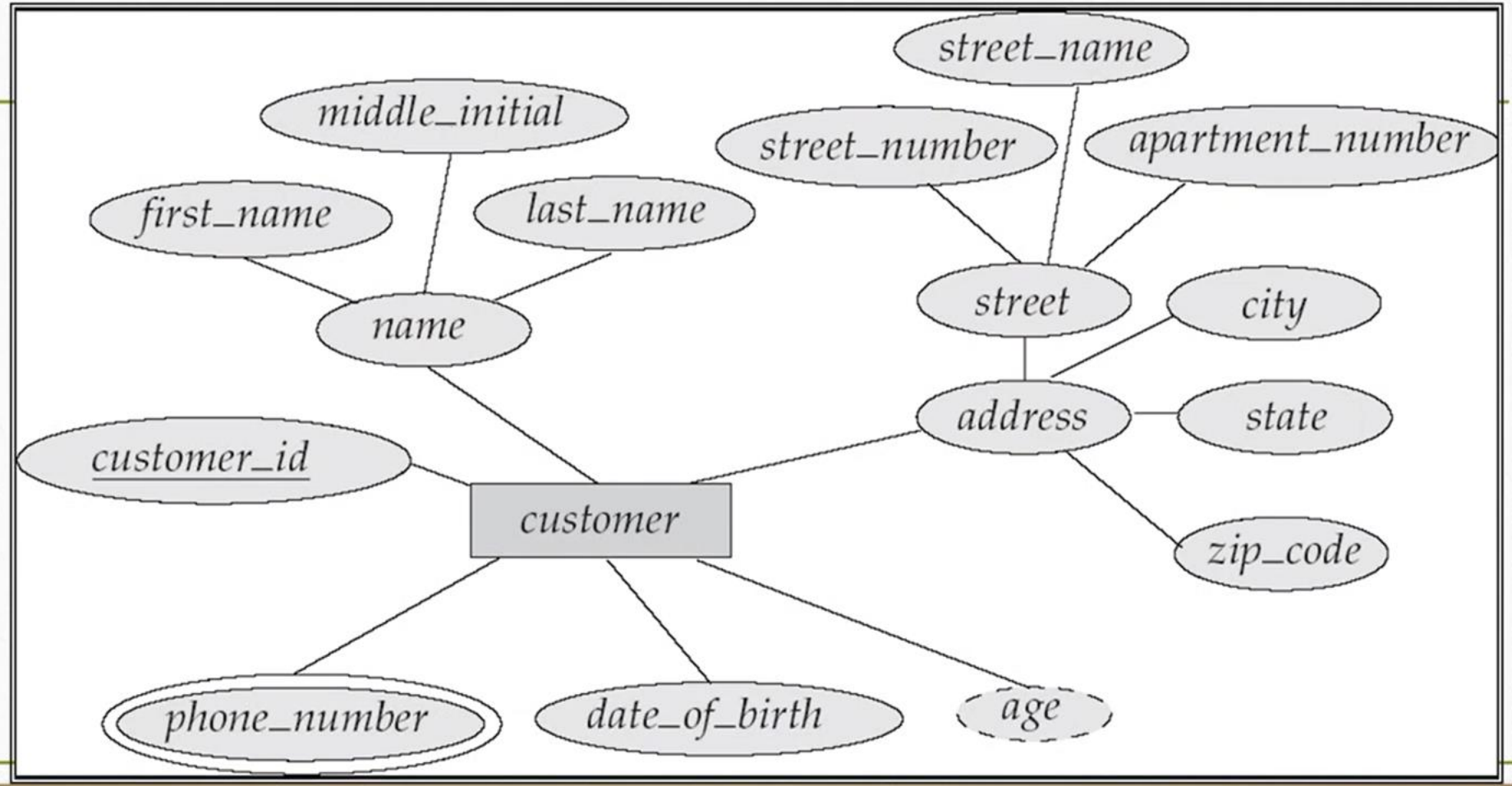
- Simple - contains only atomic values
- Multi-valued - contains several atomic values



Attribute Types

- Null attributes - used when an entity does not have a value for an attribute
- Derived attribute - value derived from other attributes or entities
  - E.g., Age from DOB

ERD with Multivalued and Derived Attributes



Relationship

- An association between entities

  Example:

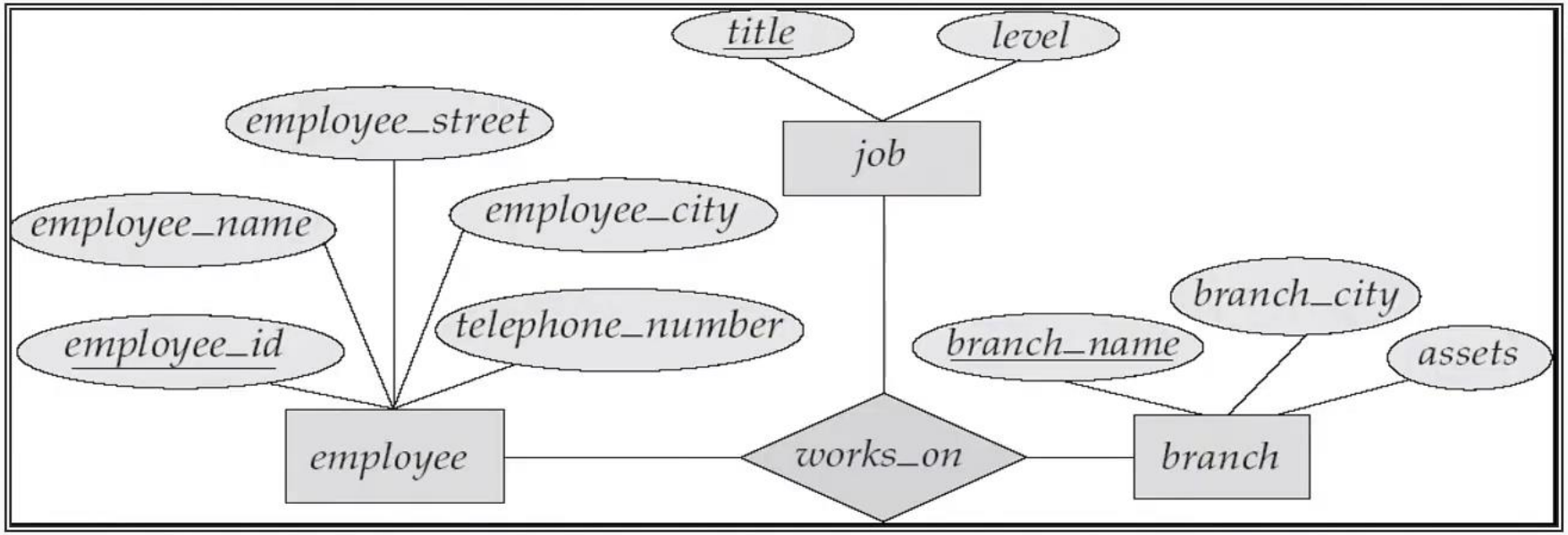
  Sam *depositor* E-100

  customer entity relationship *account* entity

Relationship Degree

- Number of entities in a relationship
  - **Binary** (or degree 2) – two entities (most common)
  - May be more than two (ternary relationship) – although rare

Example – Ternary Relation

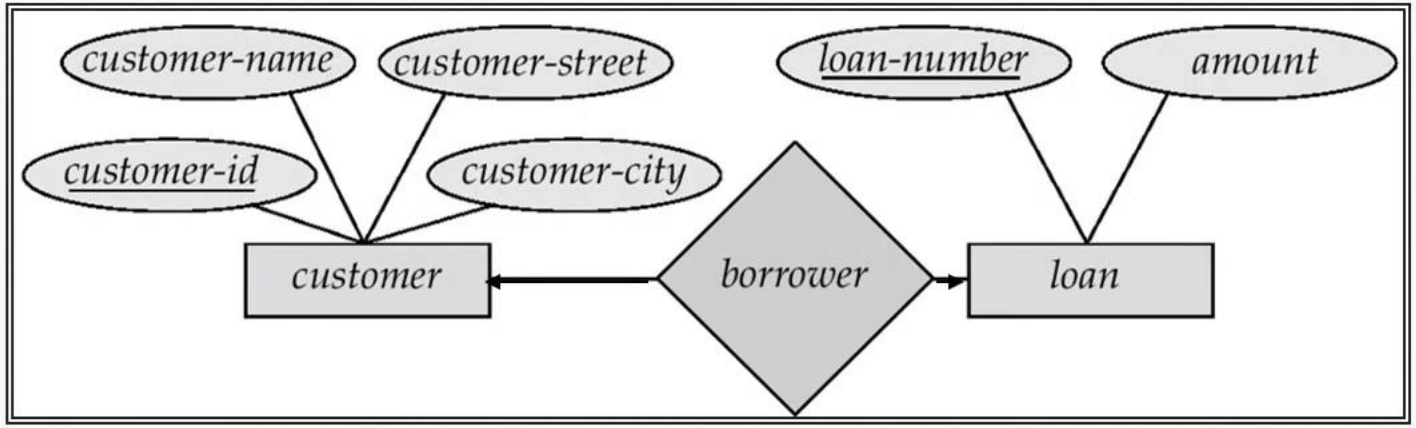


Relationship Types

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

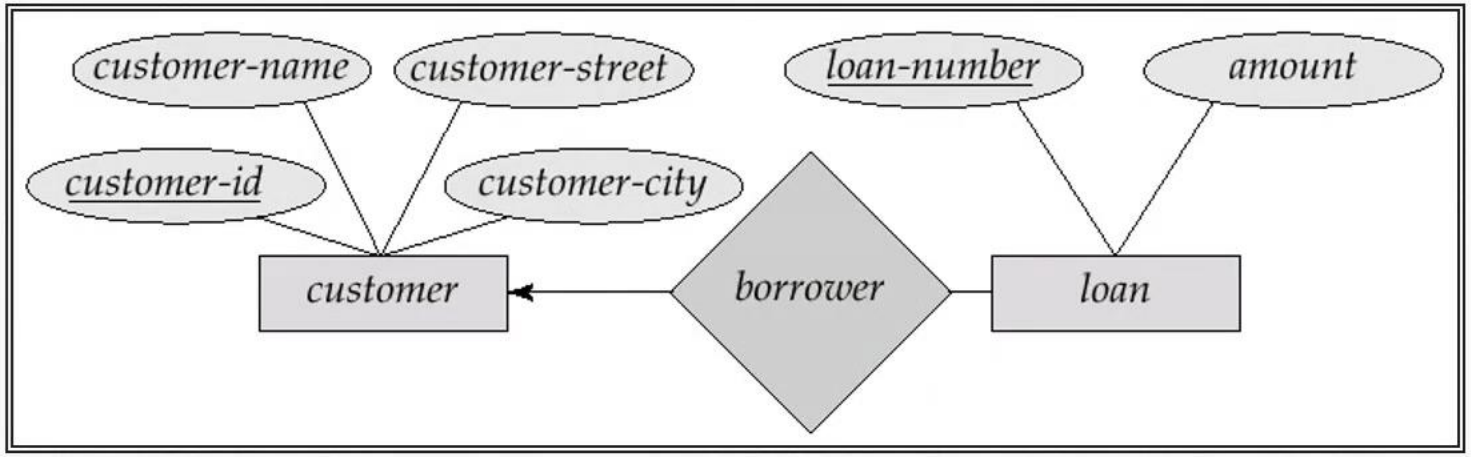
One – To - One Relationship

- A customer entity is associated with at most one (possibly 0) loan via borrower
- A loan entity is associated with at most one (possibly 0) customer via borrower



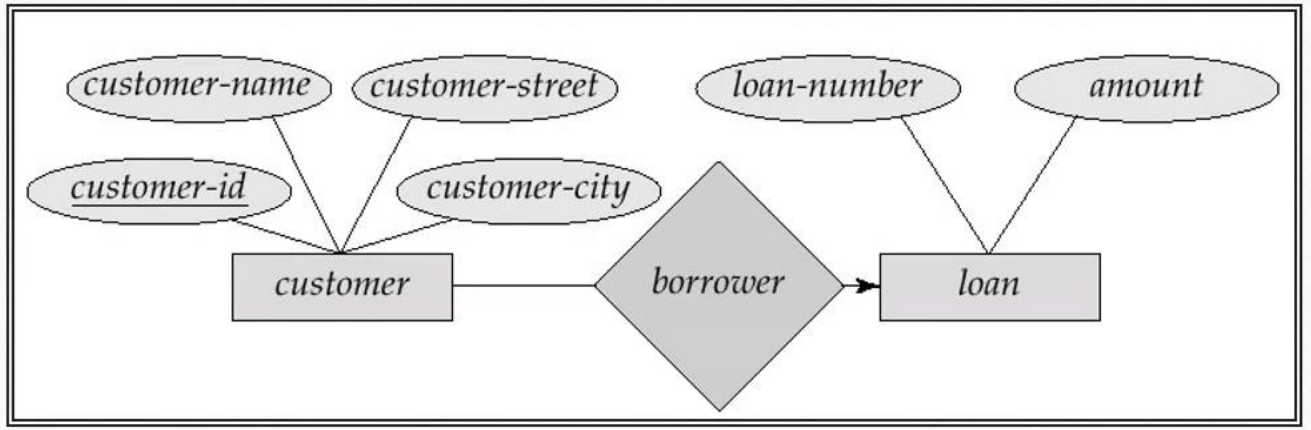
One-To-Many Relationship

- A loan entity is associated with at most one customer via *borrower*
- A customer entity is associated with several (including 0) loans via *borrower*



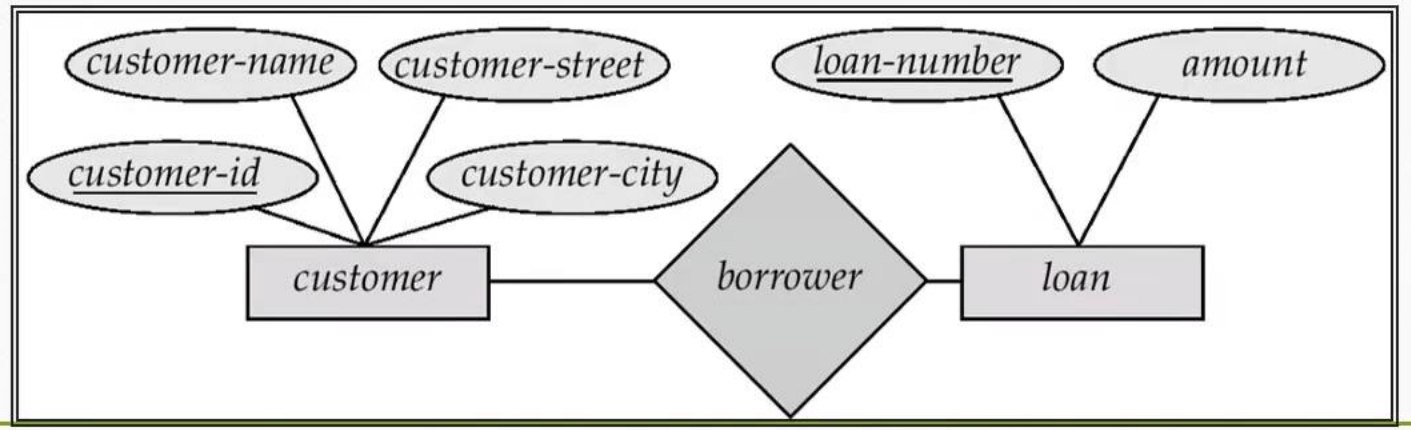
Many-To-One Relationships

- A loan entity is associated with several (including 0) customers via *borrower*
- A customer entity is associated with at most one loan via *borrower*



Many-To-Many Relationship

- A customer entity is associated with several (possibly 0) loans via borrower
- A loan entity is associated with several (possibly 0) customers via borrower



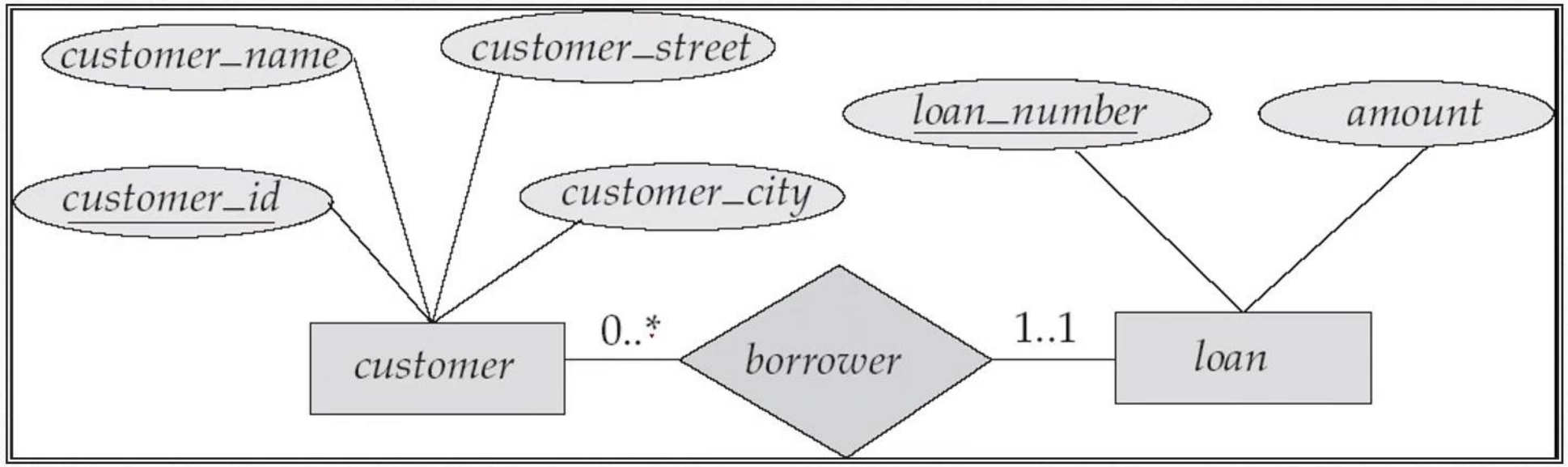
Alternative Notation for Relation

- (min..max) notation
  - Min indicates each entity is in relationship at least min times
  - Max indicates each entity is in relationship at most max times

Alternative Notation for Relation

- (min..max) notation
  - Min indicates each entity is in relationship at least min times
  - Max indicates each entity is in relationship at most max times
- Special cases
  - min=0 - does not have to be in relationship (optional)
  - max=* - may be in arbitrarily many times

E-R Diagram with Alternative Notation



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

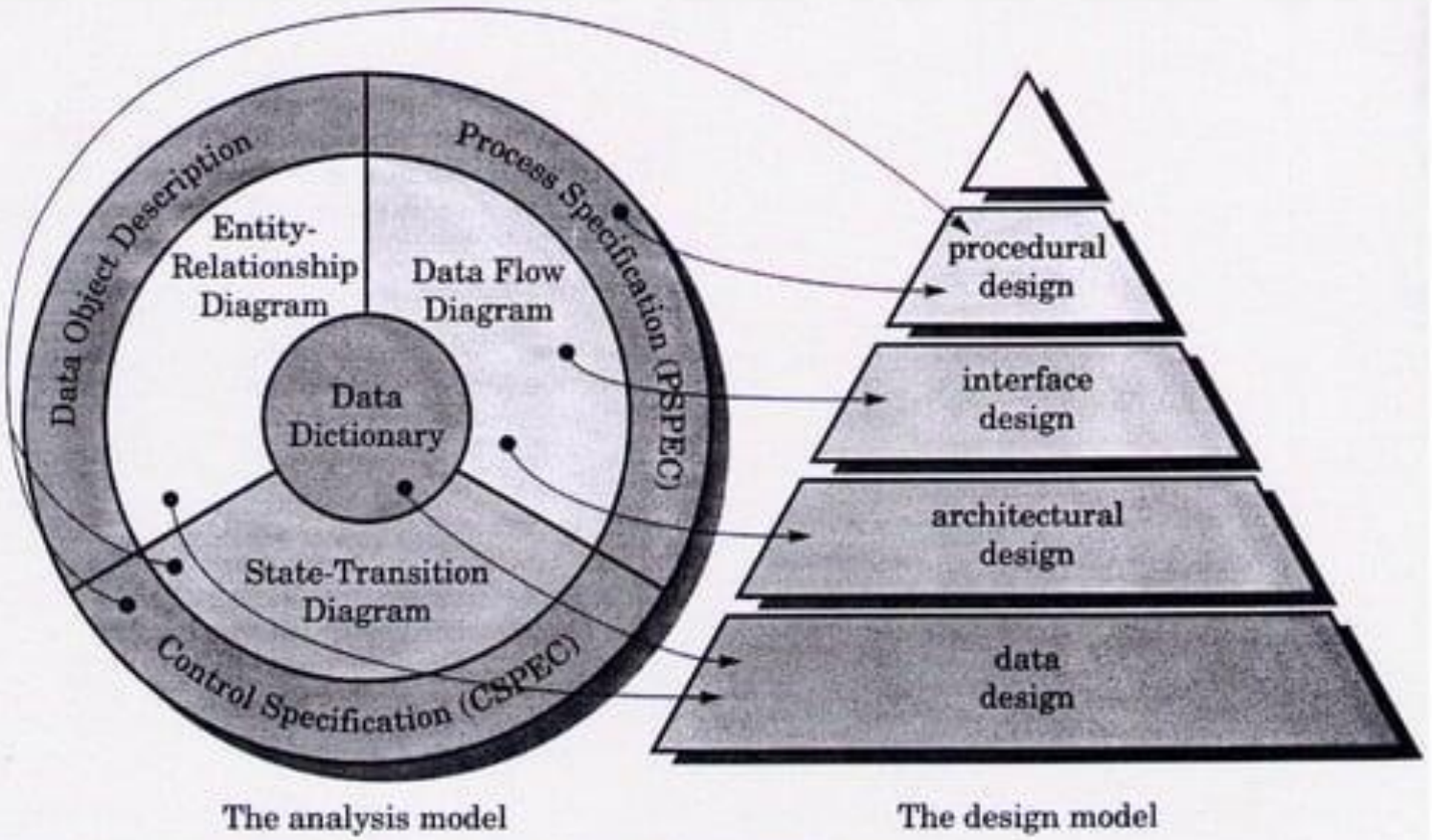
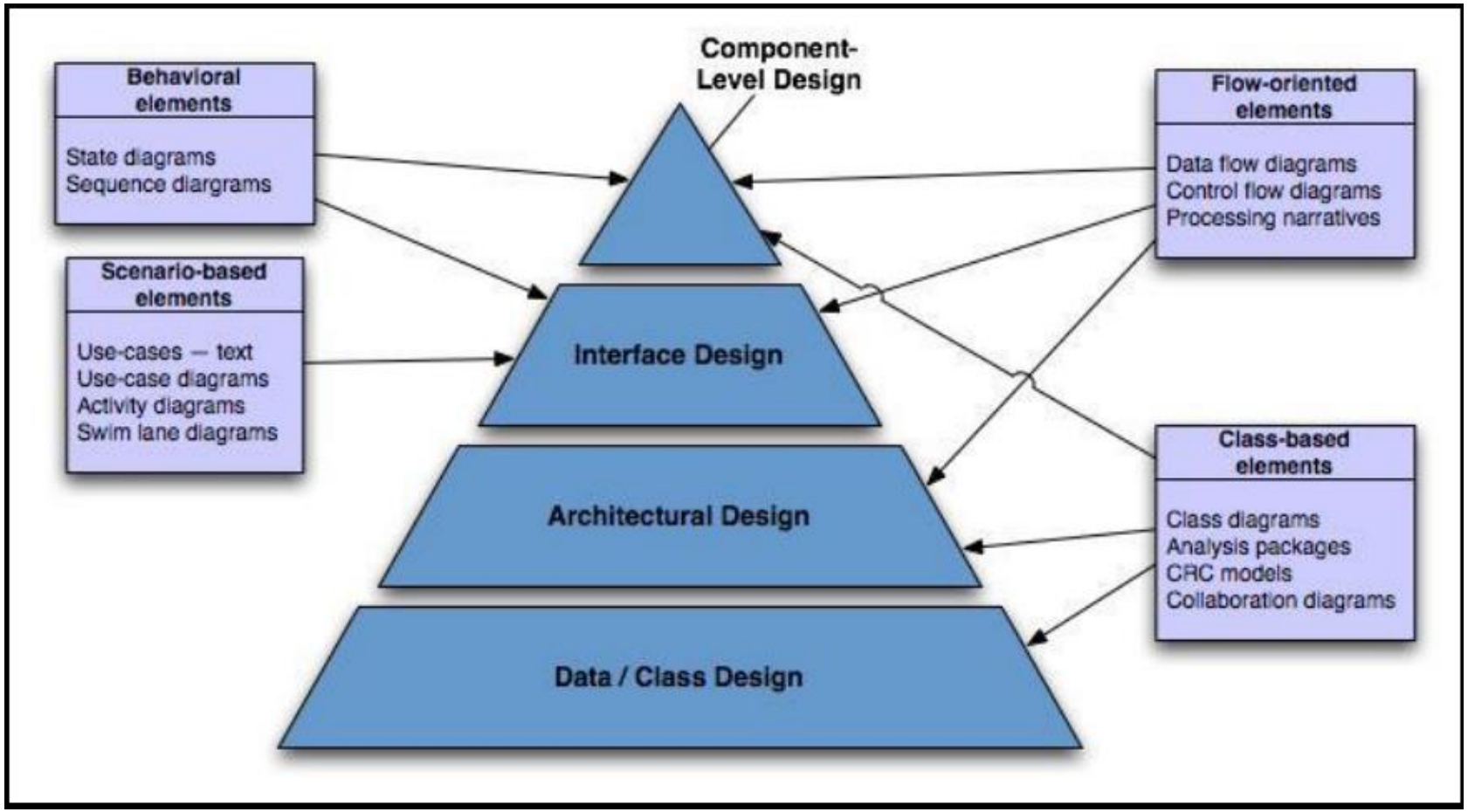


FIGURE 13.1. Translating the analysis model into a software design


Component-Level Design
Behavioral elements
State diagrams
Sequence diargrams
Scenario-based elements
Use-cases — text
Use-case diagrams
Activity diagrams
Swim lane diagrams
Flow-oriented elements
Data flow diagrams
Control flow diagrams
Processing narratives
Class-based elements
Class diagrams
Analysis packages
CRC models
Collaboration diagrams
Interface Design
Architectural Design
Data / Class Design

Design Concepts(Abstraction)

- Wasserman: “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)

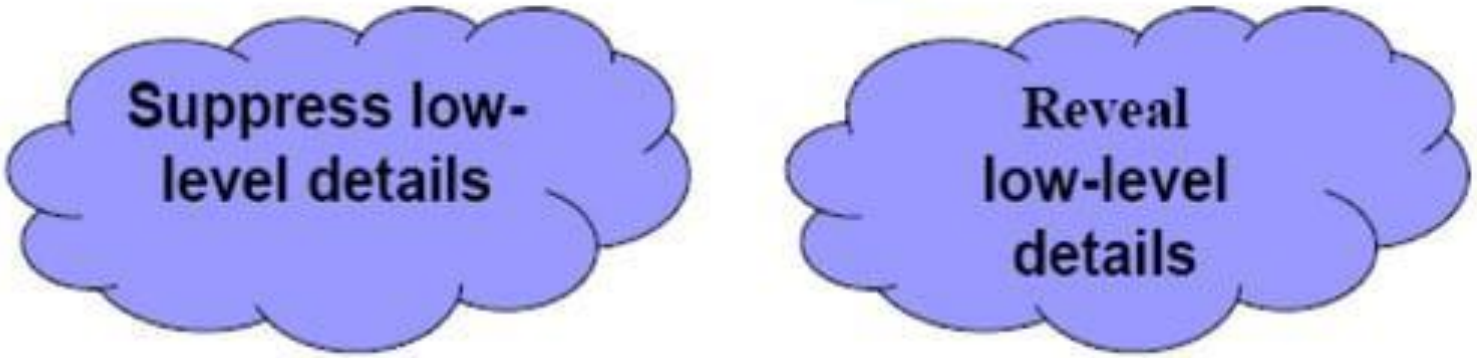
Design Concepts(Refinement)

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)

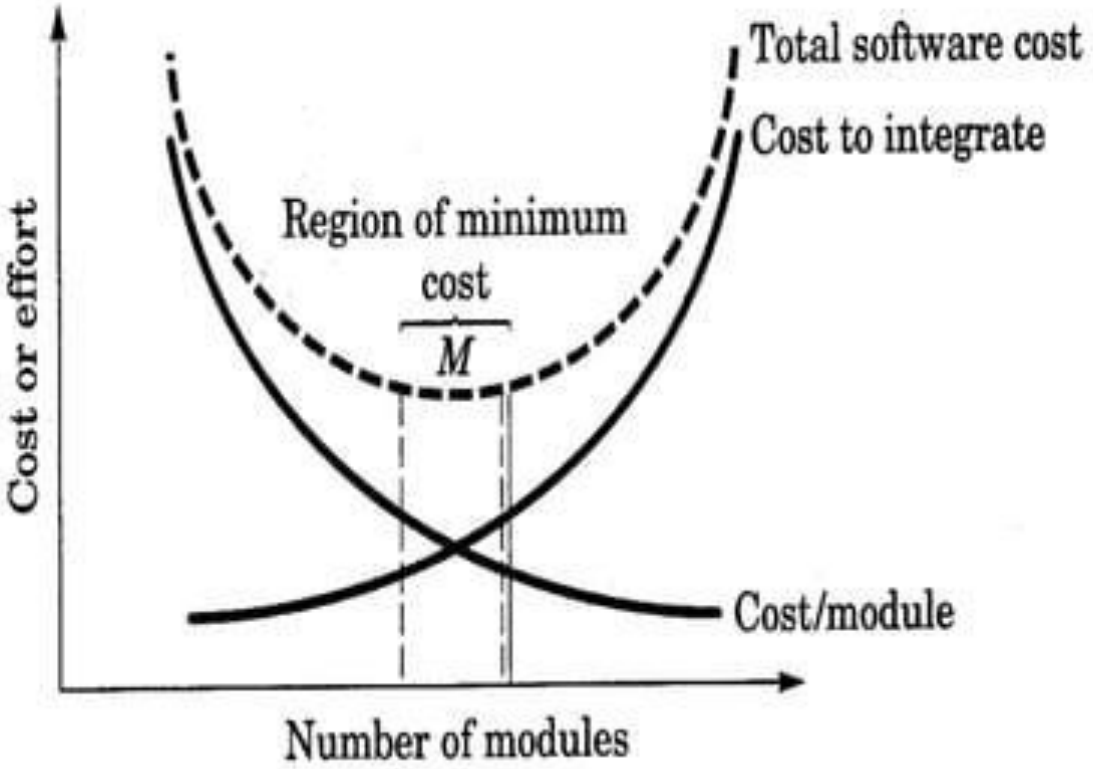


FIGURE 13.2.
Modularity and software cost

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)

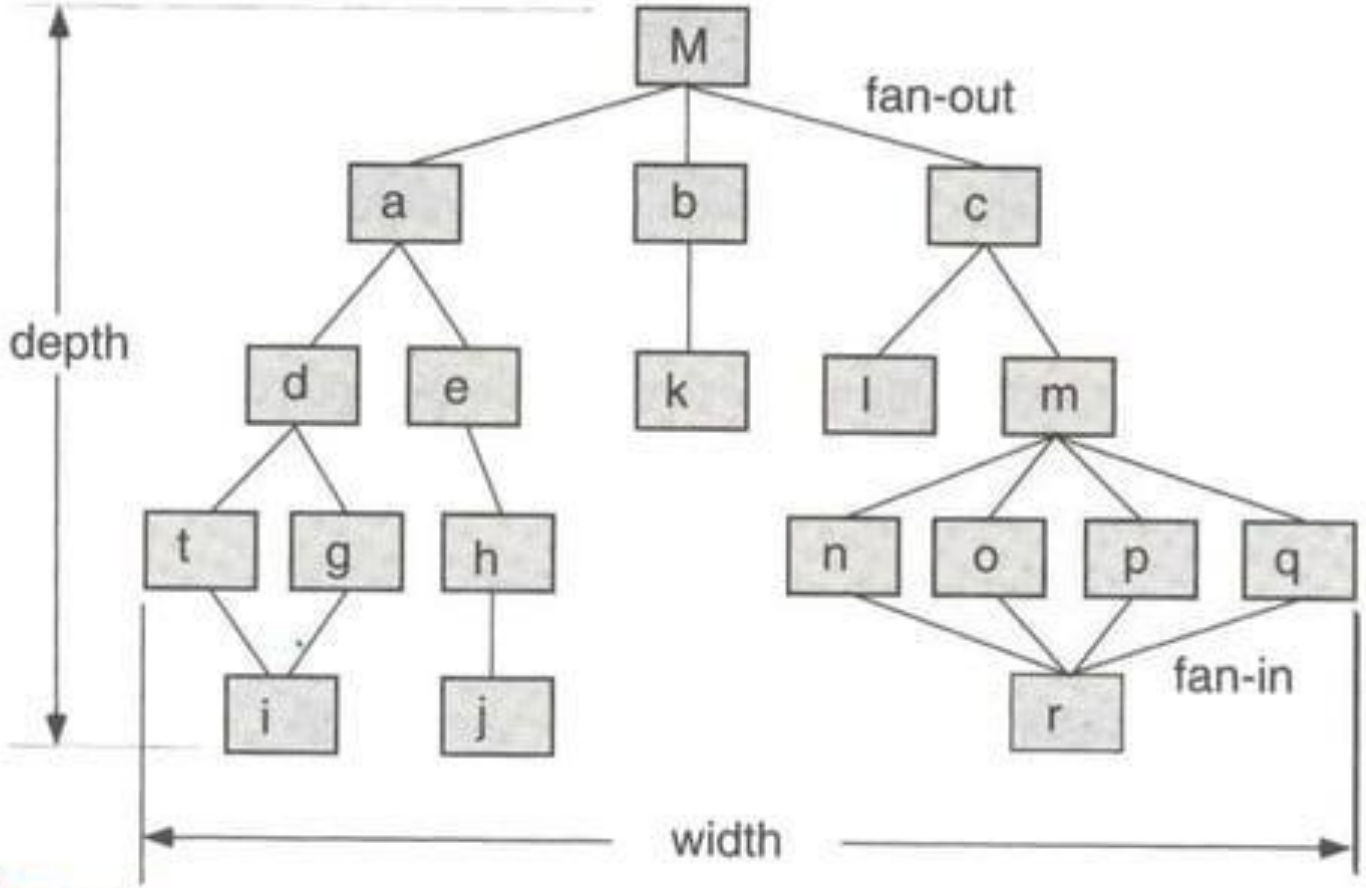
Modular Composability

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(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
- **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.
- **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 (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.

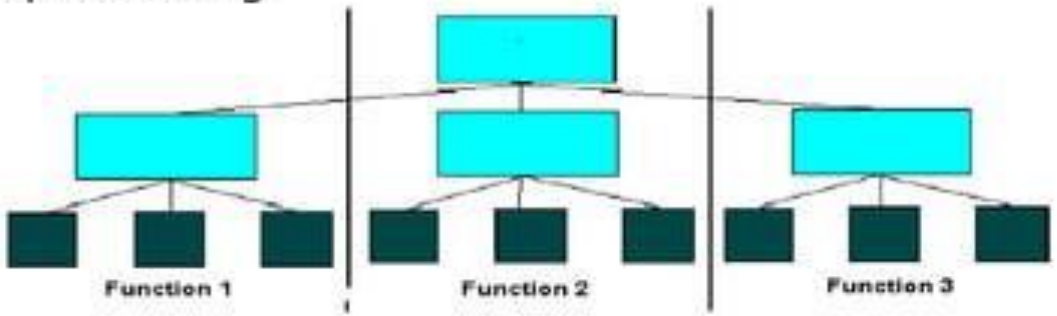
Design Concepts (Structural partitioning)

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

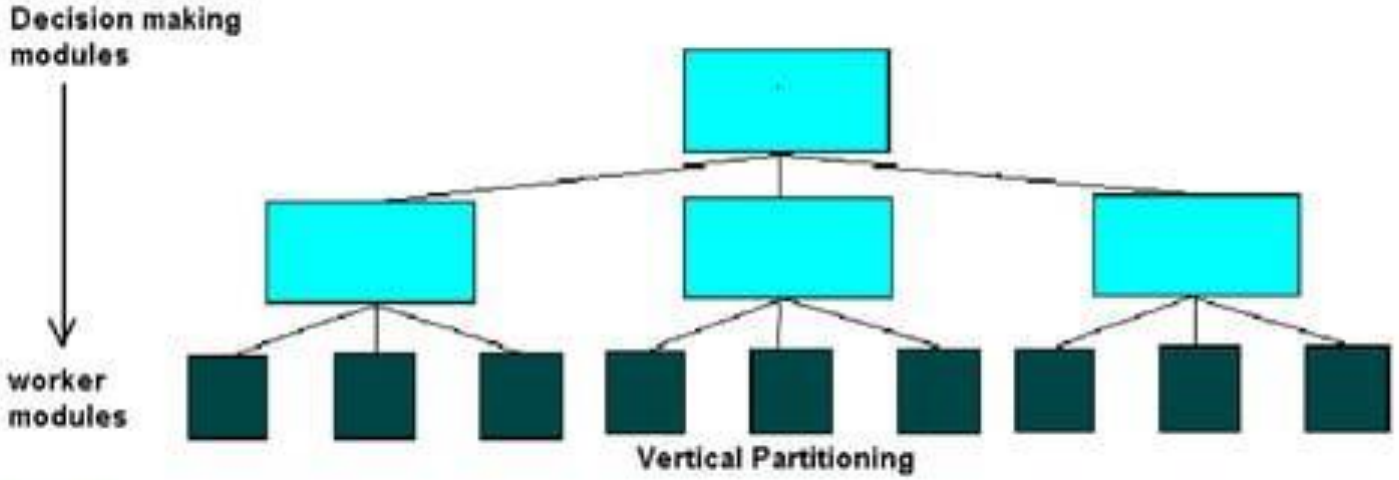
Advantages

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

Design Concepts (Structural partitioning)

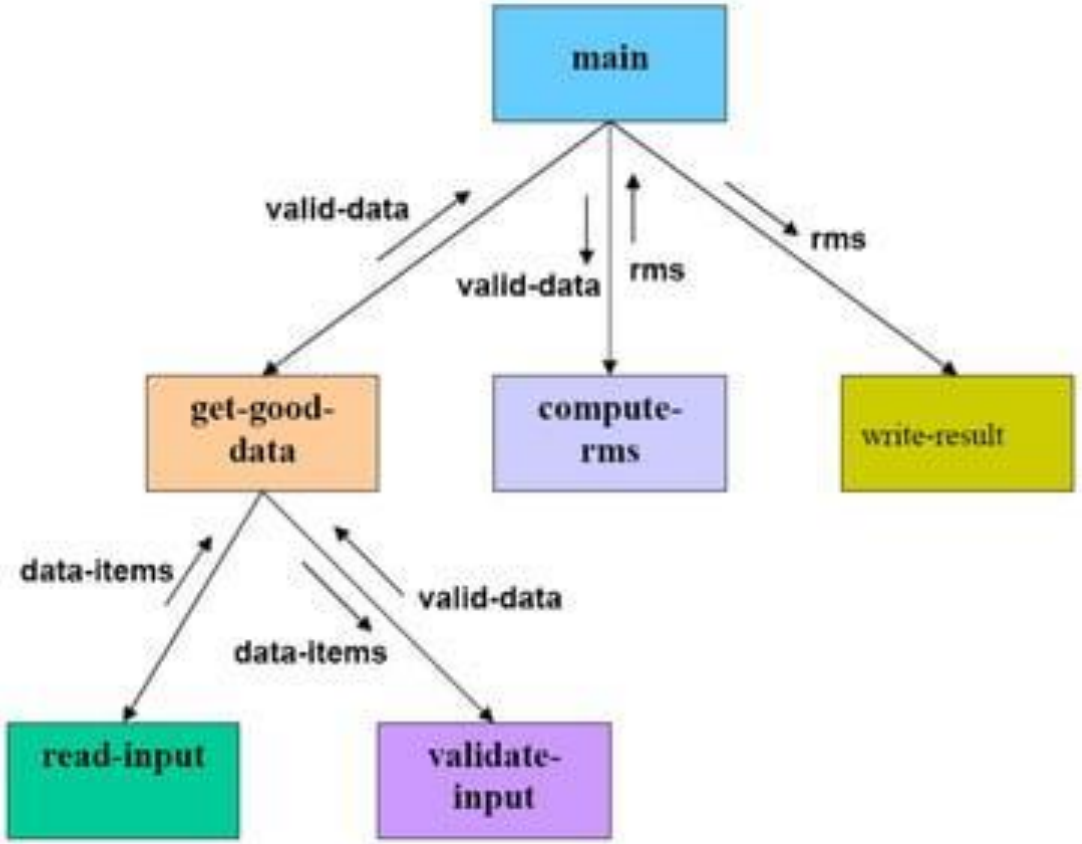
Vertical Partitioning

- 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

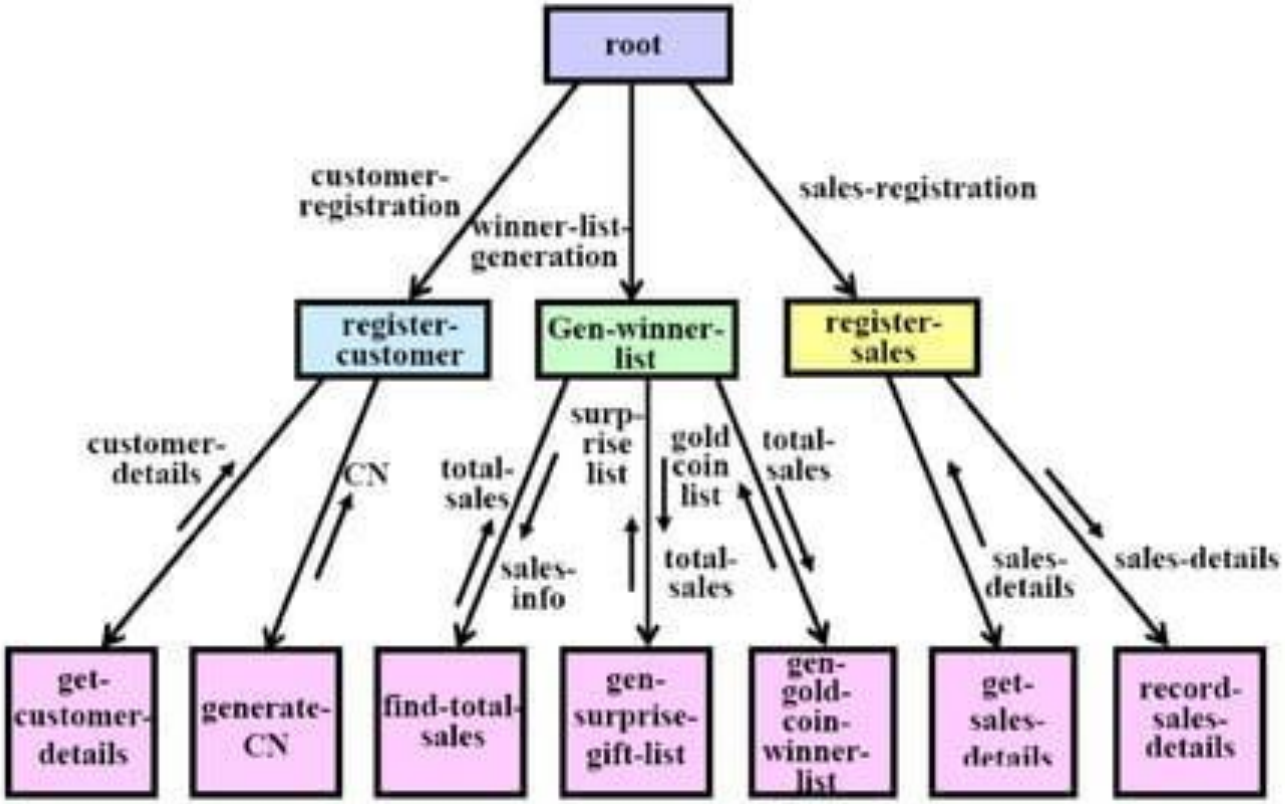


Fig. 36.13 Structure chart for the supermarket prize scheme

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

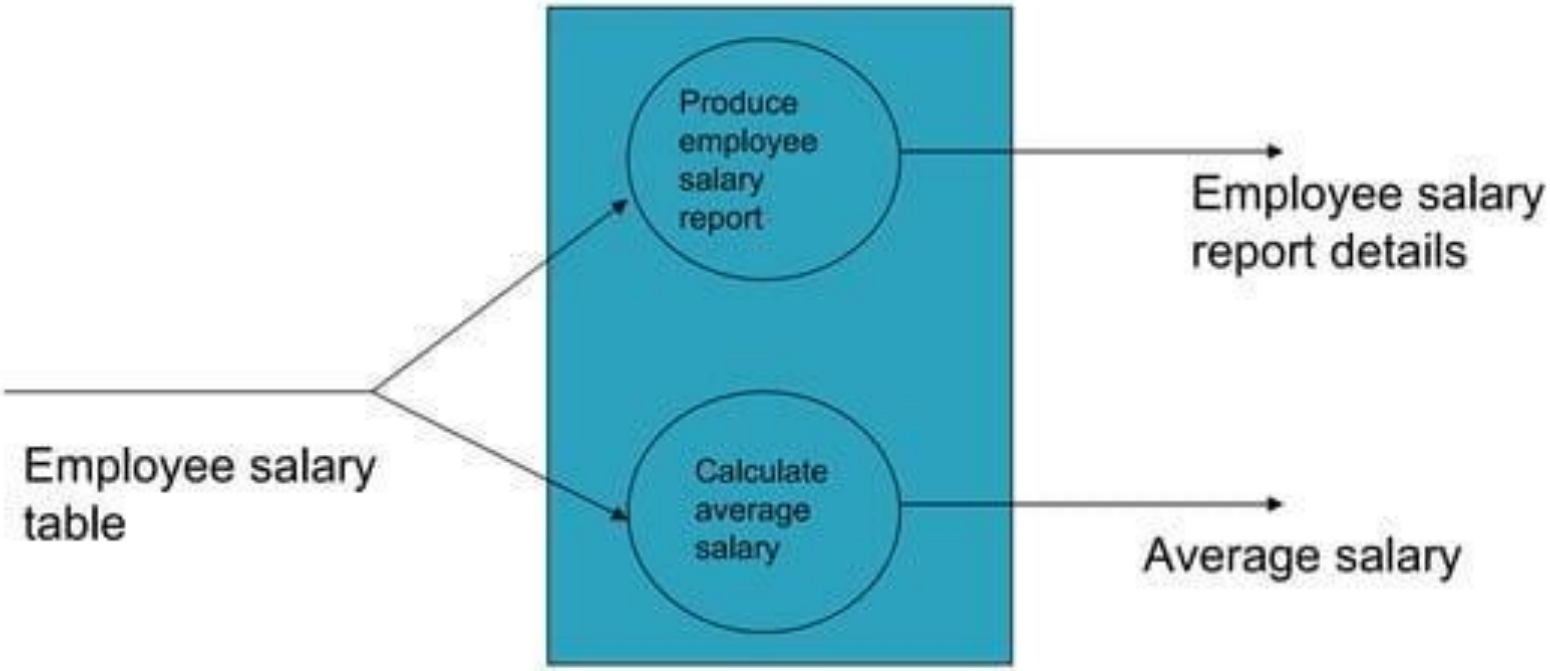
For instance, we may wish to

- 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
        rewind tape-a
        set a-counter to 0
        rewind tape-b
        set b-counter to 0
        clear items table
        clear totals table
        set switch-a to off
        set switch-b to on
    endmodule
```

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

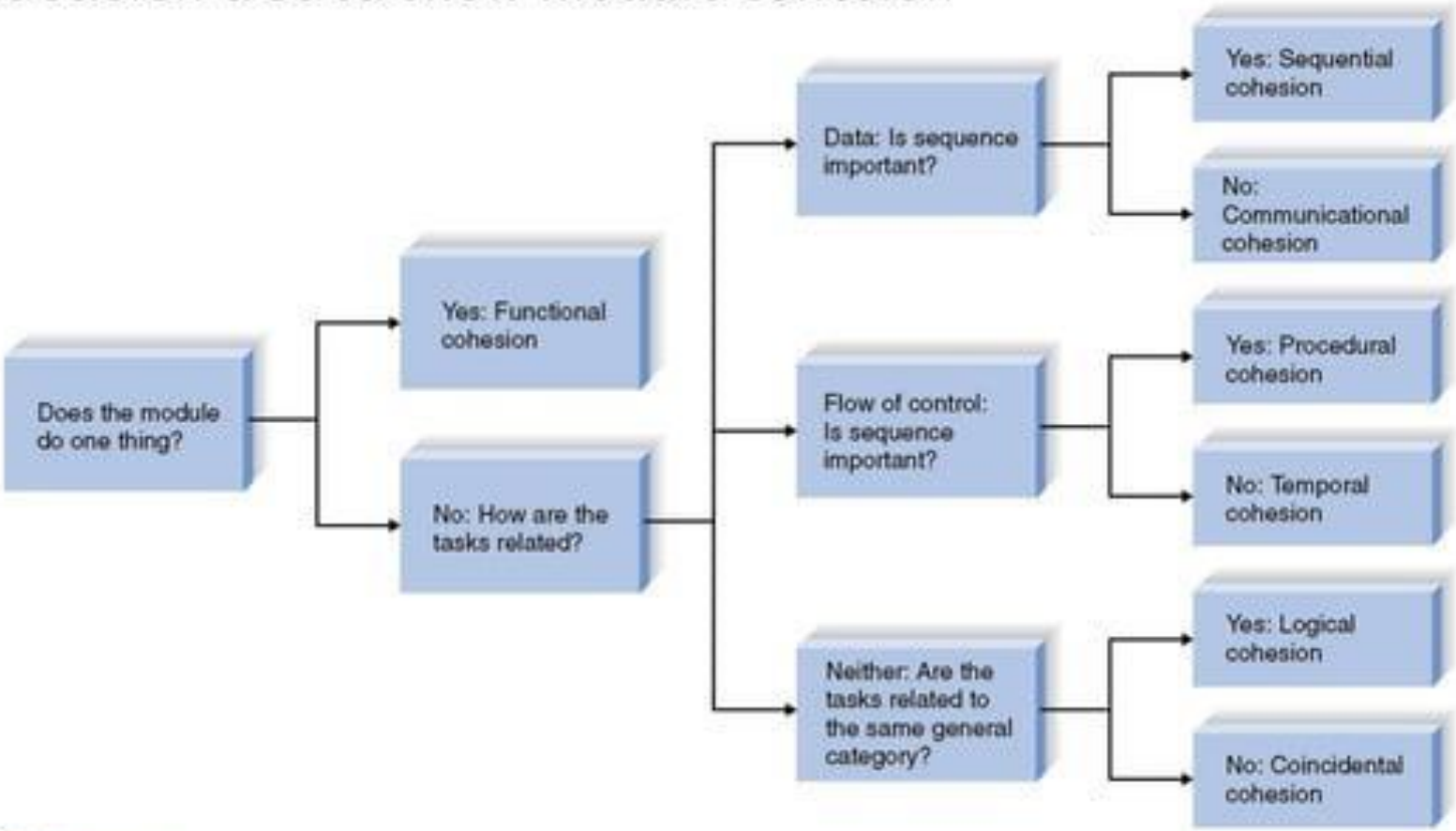
7. Coincidental Cohesion (Lowest)

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 (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.

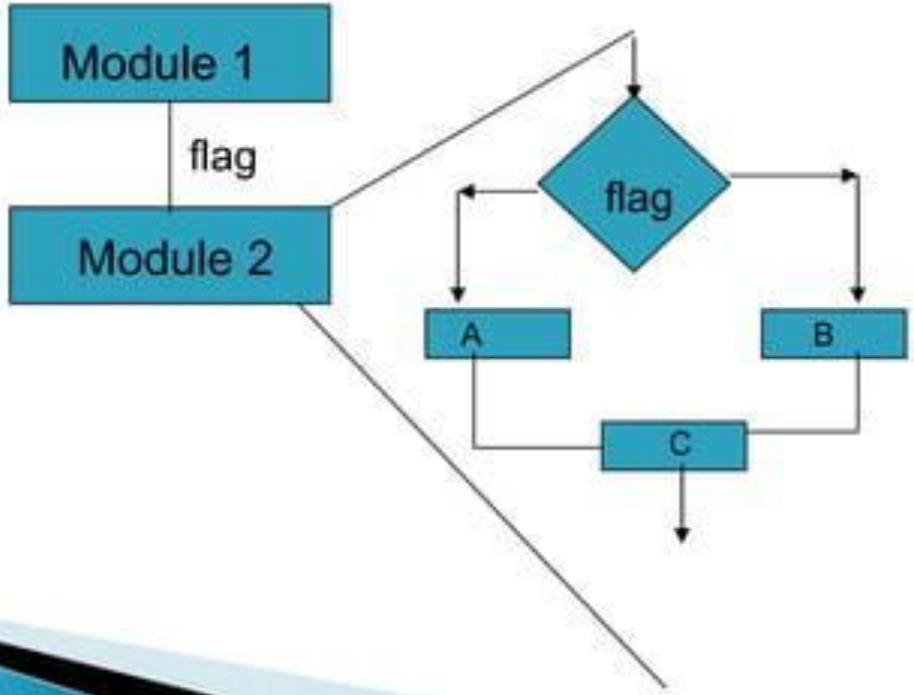
Coupling

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.

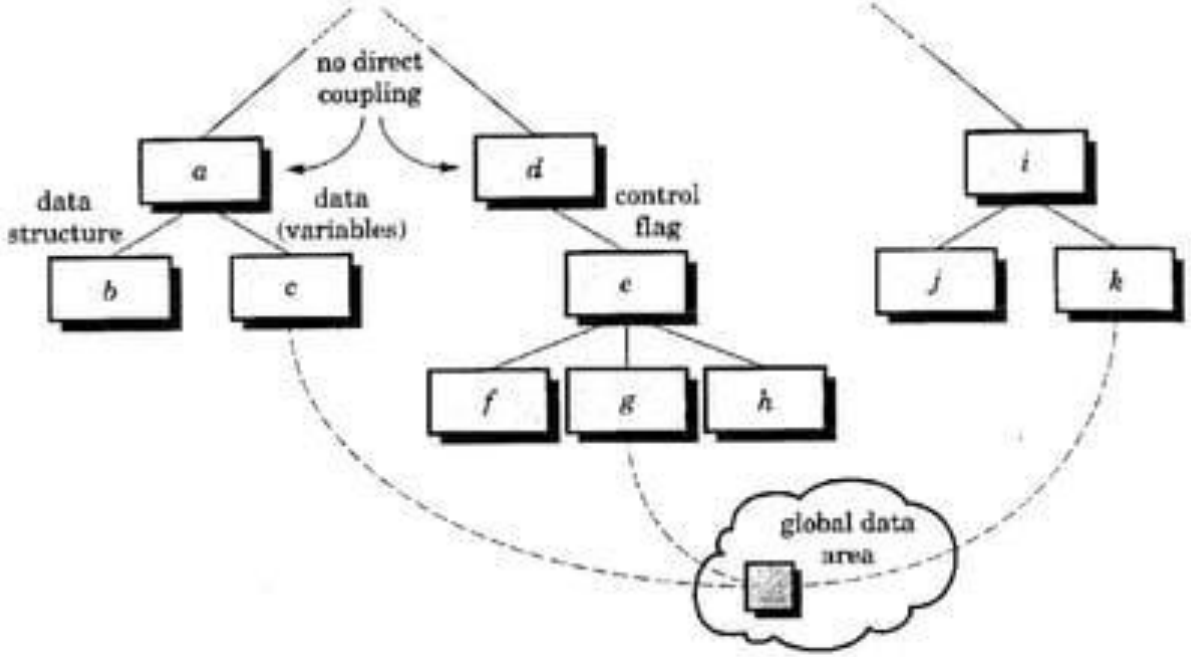


FIGURE 13.8. Types of coupling

Modules c, g and k exhibit common coupling

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

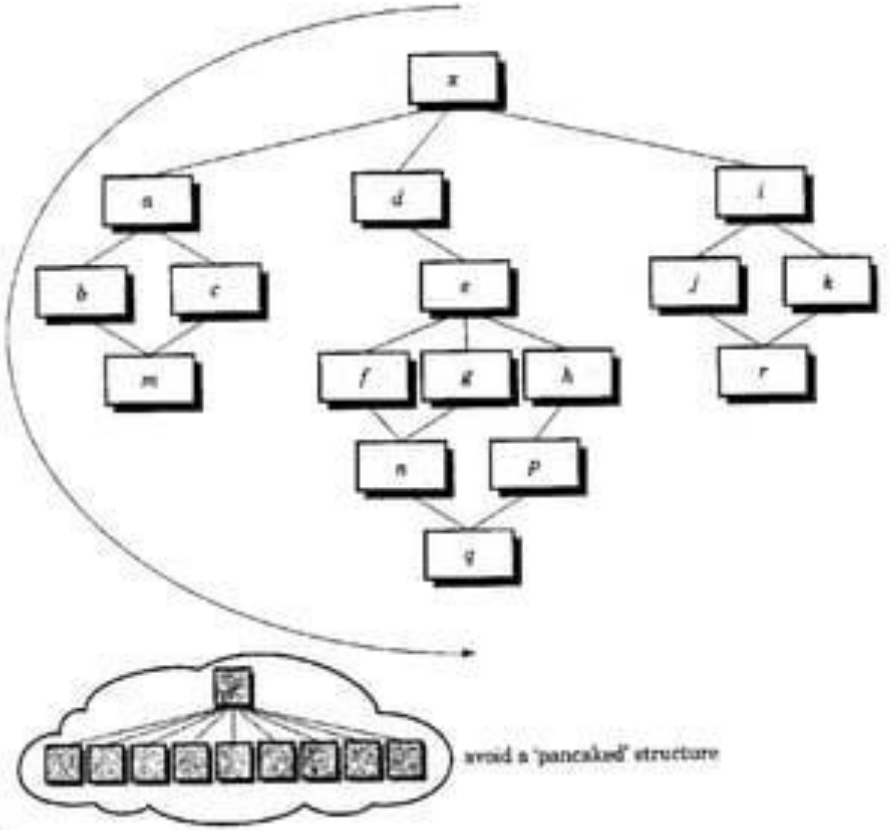


FIGURE 13.9. Program structures

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

Structure Charts

Structure Charts

Structure Chart represents the hierarchical structure of modules. It breaks down the entire system into the lowest functional modules and describes the functions and sub-functions of each module of a system in greater detail. This article focuses on discussing Structure Charts in detail.

What is a Structure Chart?

Structure Chart partitions the system into black boxes (functionality of the system is known to the users, but inner details are unknown).

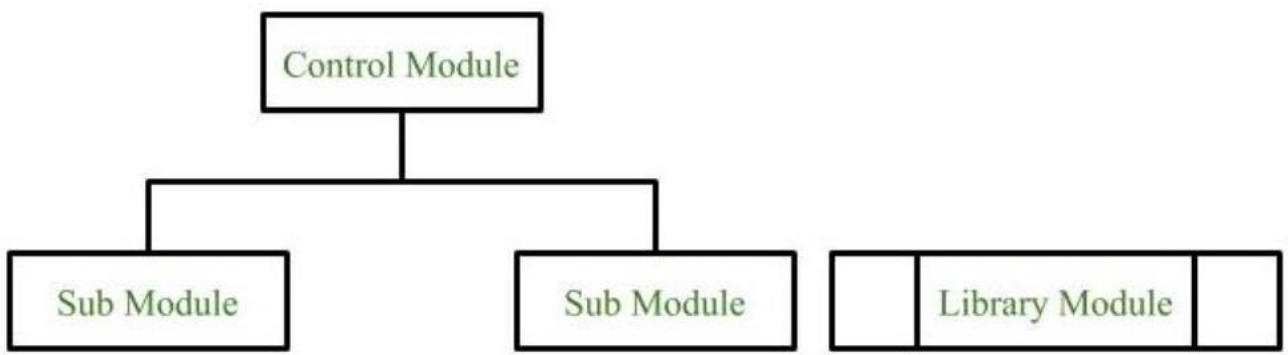
1. Inputs are given to the black boxes and appropriate outputs are generated.
2. Modules at the top level are called modules at low level.
3. Components are read from top to bottom and left to right.
4. When a module calls another, it views the called module as a black box, passing the required parameters and receiving results.

Symbols in Structured Chart

1. Module

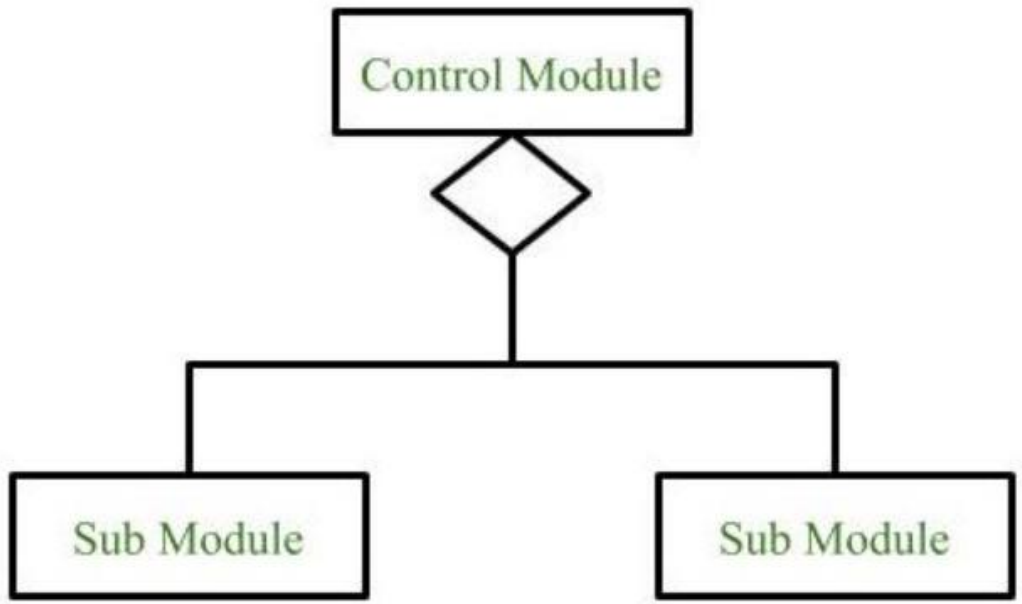
It represents the process or task of the system. It is of three types:

- **Control Module**: A control module branches to more than one submodule.
- **Sub Module:** Sub Module is a module which is the part (Child) of another module.
- **Library Module:** Library Module are reusable and invokable from any module.



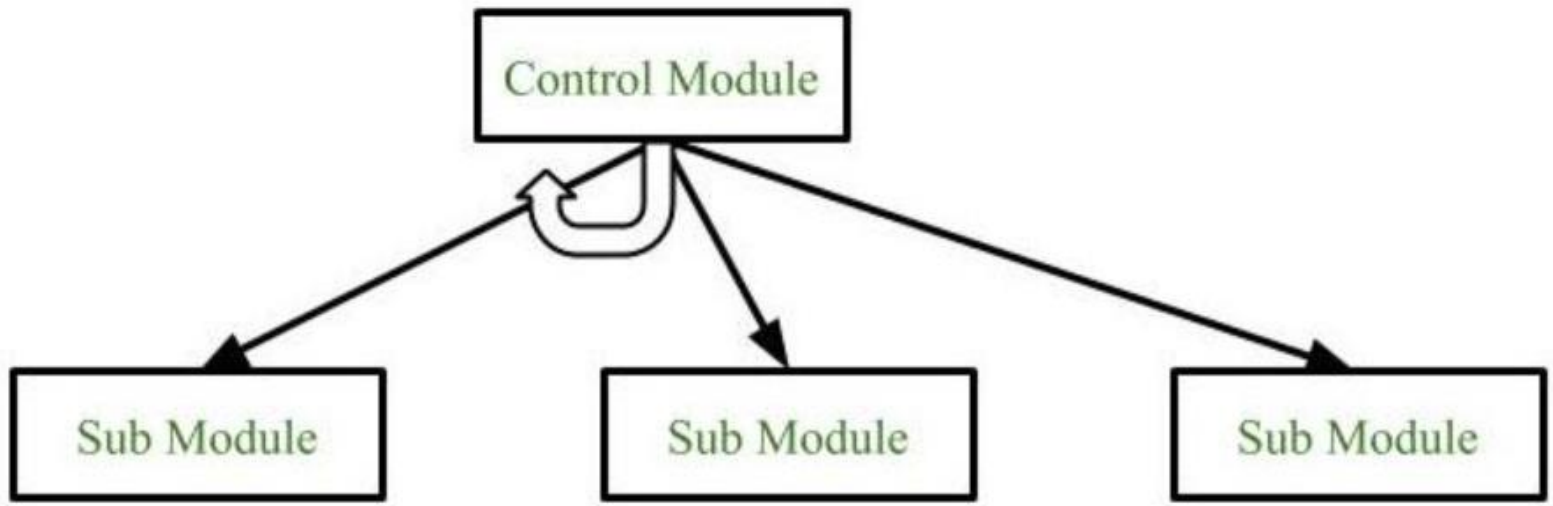
2. Conditional Call

It represents that control module can select any of the sub module on the basis of some condition.



3. Loop (Repetitive call of module)

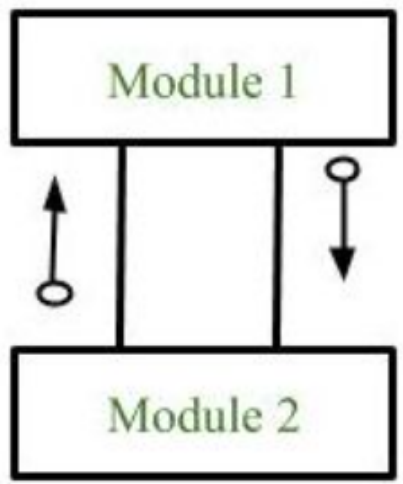
It represents the repetitive execution of module by the sub module. A curved arrow represents a loop in the module.



All the submodules cover by the loop repeat execution of module.

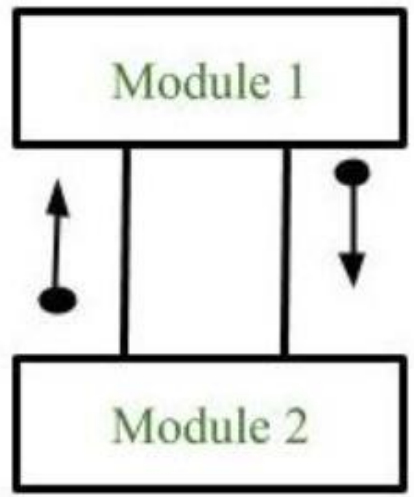
4. Data Flow

It represents the flow of data between the modules. It is represented by a directed arrow with an empty circle at the end.



5. Control Flow

It represents the flow of control between the modules. It is represented by a directed arrow with a filled circle at the end.

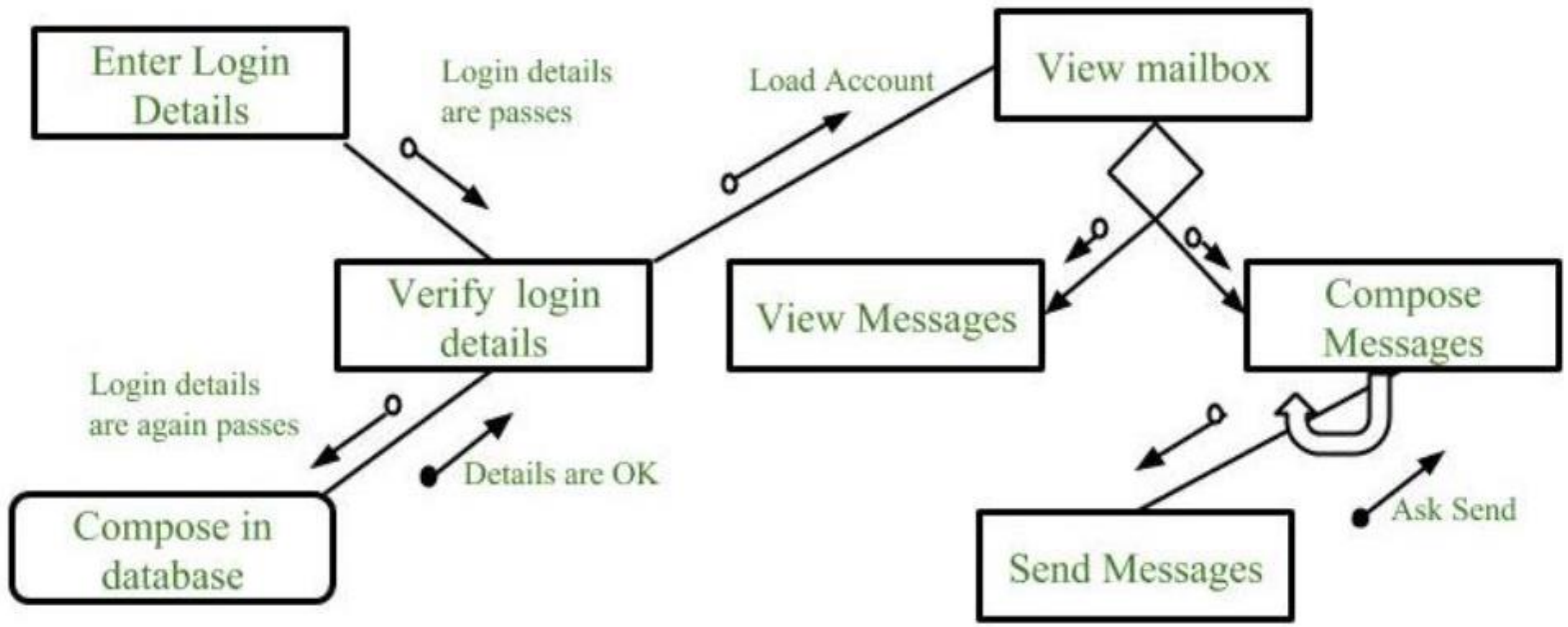


6. Physical Storage

It is that where all the information are to be stored.

Example

Structure chart for an Email server



Architectural Design

- Introduction
- Data design
- Software architectural styles
- Architectural design process
- Assessing alternative architectural designs

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

Introduction

Definitions

- The software architecture of a program or computing system is the structure or structures of the system which comprise
  - The software components
  - The externally visible properties of those components
  - The relationships among the components
- Software architectural design represents the structure of the data and program components that are required to build a computer-based system
- An architectural design model is transferable
  - It can be applied to the design of other systems
  - It represents a set of abstractions that enable software engineers to describe architecture in predictable ways

Architectural Design Process

- Basic Steps
  - Creation of the data design
  - Derivation of one or more representations of the architectural structure of the system
  - Analysis of alternative architectural styles to choose the one best suited to customer requirements and quality attributes
  - Elaboration of the architecture based on the selected architectural style
- A database designer creates the data architecture for a system to represent the data components
- A system architect selects an appropriate architectural style derived during system engineering and software requirements analysis

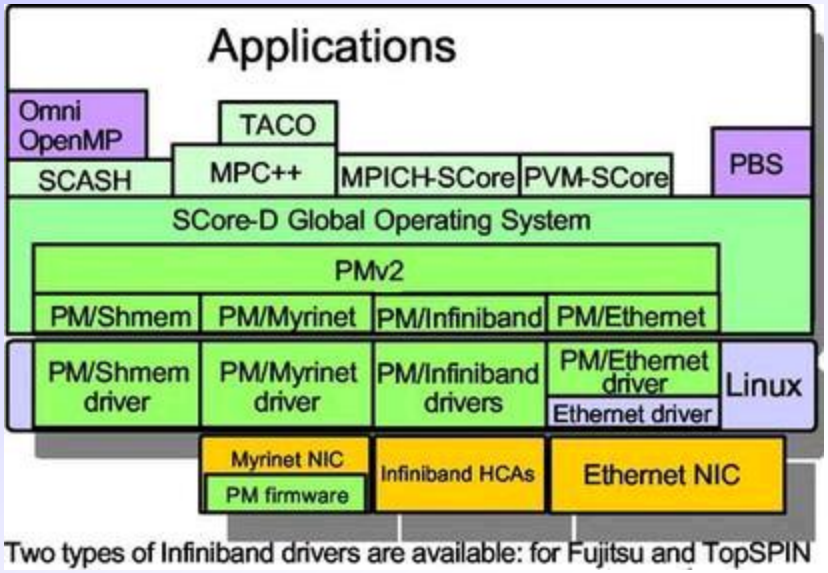
Emphasis on Software Components

- A software architecture enables a software engineer to
  - Analyze the effectiveness of the design in meeting its stated requirements
  - Consider architectural alternatives at a stage when making design changes is still relatively easy
  - Reduce the risks associated with the construction of the software
- Focus is placed on the software component
  - A program module
  - An object-oriented class
  - A database
  - Middleware

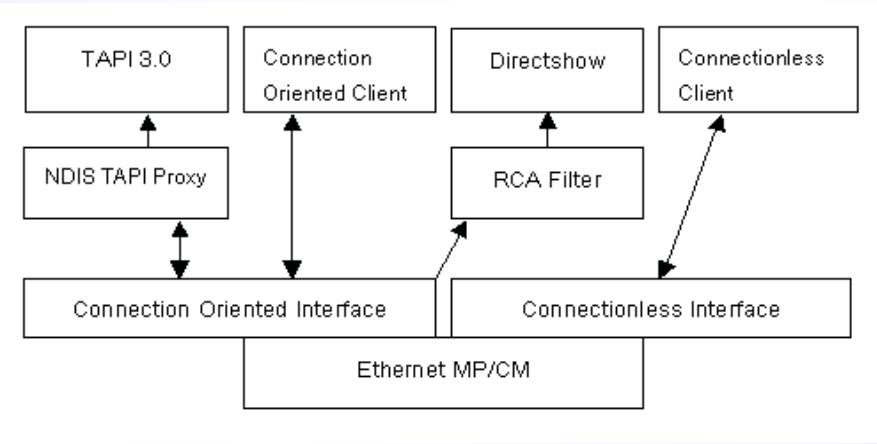
Importance of Software Architecture

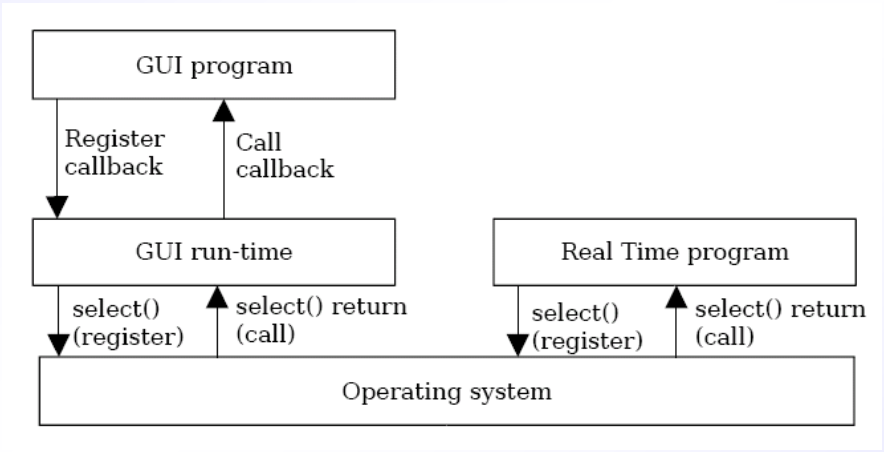
- Representations of software architecture are an enabler for communication between all stakeholders interested in the development of a computer-based system
- The software architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity
- The software architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together

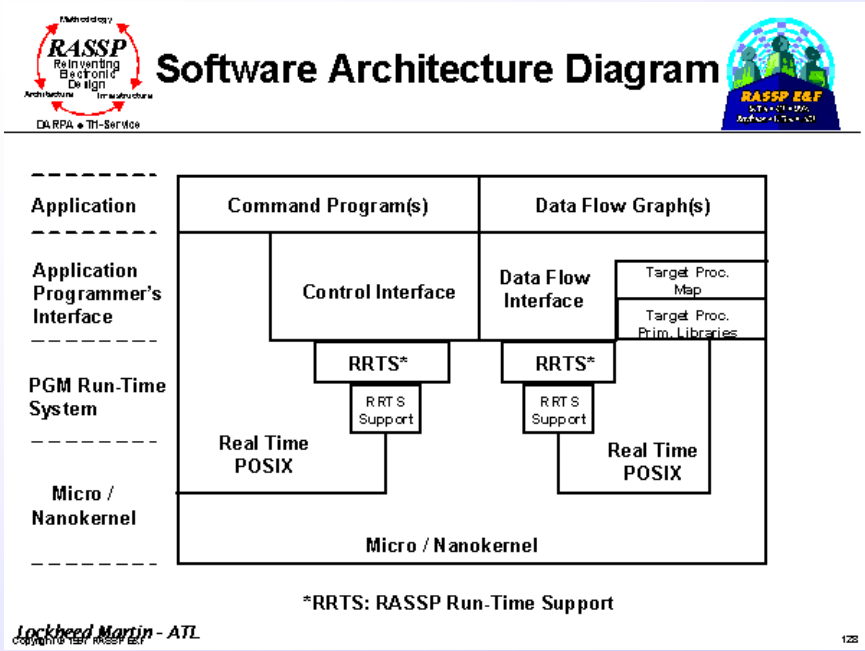
Example Software Architecture Diagrams



Two types of Infiniband drivers are available: for Fujitsu and TopSPIN







Data Design

Purpose of Data Design

- Data design translates data objects defined as part of the analysis model into
  - Data structures at the software component level
  - A possible database architecture at the application level
- It focuses on the representation of data structures that are directly accessed by one or more software components
- The challenge is to store and retrieve the data in such way that useful information can be extracted from the data environment
- "Data quality is the difference between a data warehouse and a data garbage dump"

Data Design Principles

- The systematic analysis principles that are applied to function and behavior should also be applied to data
- All data structures and the operations to be performed on each one should be identified
- A mechanism for defining the content of each data object should be established and used to define both data and the operations applied to it
- Low-level data design decisions should be deferred until late in the design process
- The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure
- A library of useful data structures and the operations that may be applied to them should be developed
- A software programming language should support the specification and realization of abstract data types

Software Architectural Styles

Common Architectural Styles of American Homes

Common Architectural Styles of American Homes

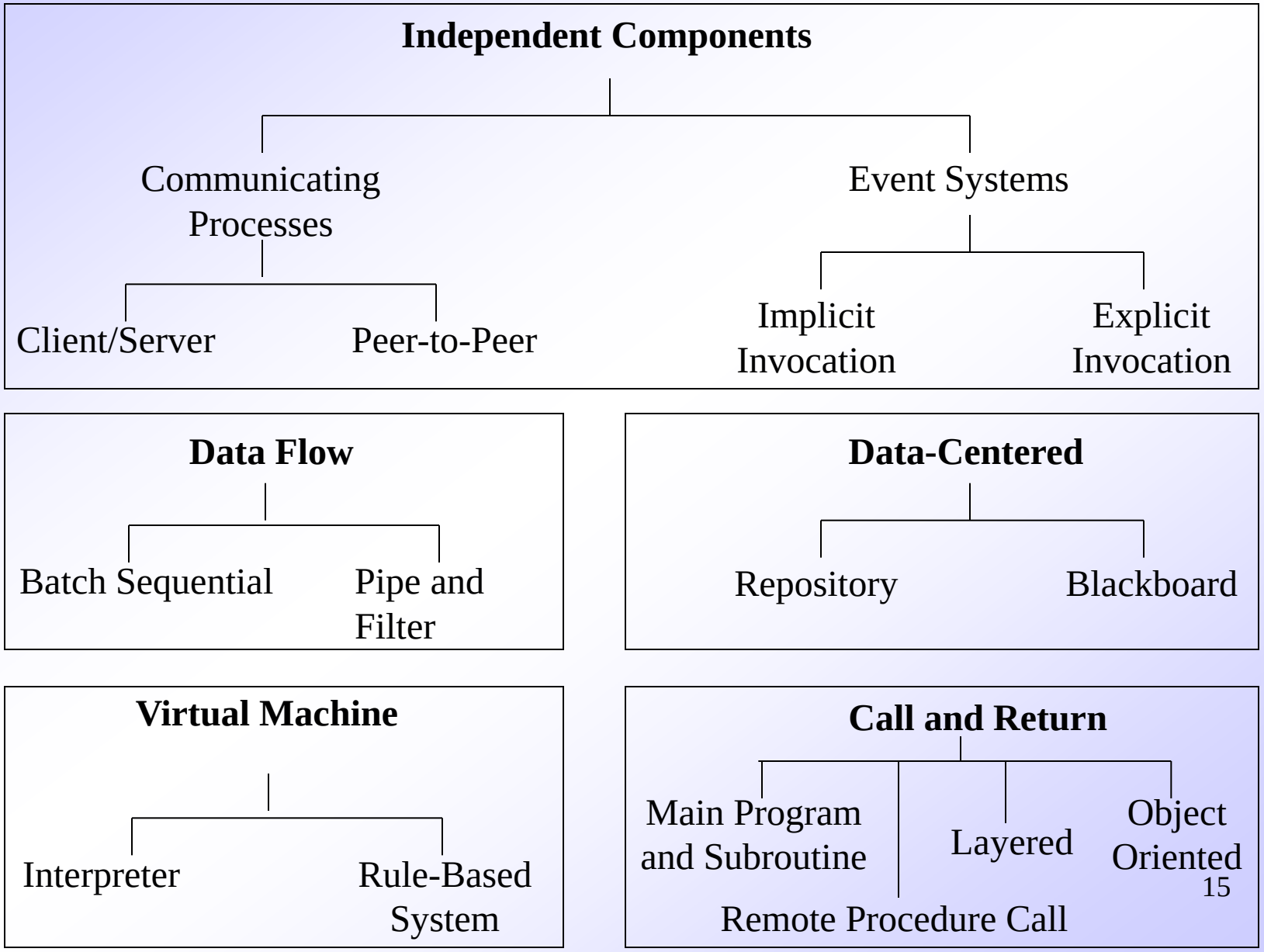
- A-Frame
- Bungalow
- Cape Cod
- Colonial
- Federal
- Four square
- Georgian
- Greek Revival
- Prairie Style
- Pueblo
- Ranch
- Split level
- Tidewater
- Tudor
- Victorian

Software Architectural Style

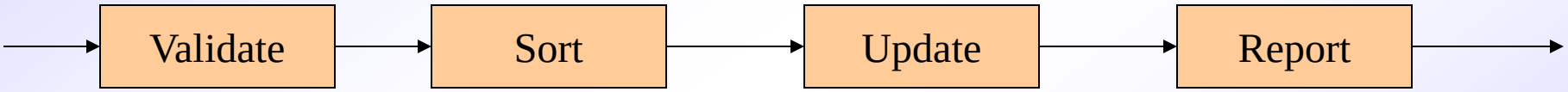
- The software that is built for computer-based systems exhibit one of many architectural styles
- Each style describes a system category that encompasses
  - A set of component types that perform a function required by the system
  - A set of connectors (subroutine call, remote procedure call, data stream, socket) that enable communication, coordination, and cooperation among components
  - Semantic constraints that define how components can be integrated to form the system
  - A topological layout of the components indicating their runtime interrelationships

(Source: Bass, Clements, and Kazman. *Software Architecture in Practice.* Addison-Wesley, 2003)

A Taxonomy of Architectural Styles



Data Flow Style



Data Flow Style

- Has the goal of modifiability
- Characterized by viewing the system as a series of transformations on successive pieces of input data
- Data enters the system and then flows through the components one at a time until they are assigned to output or a data store
- Batch sequential style
  - The processing steps are independent components
  - Each step runs to completion before the next step begins
- Pipe-and-filter style
  - Emphasizes the incremental transformation of data by successive components
  - The filters incrementally transform the data (entering and exiting via streams)
  - The filters use little contextual information and retain no state between instantiations
  - The pipes are stateless and simply exist to move data between filters

(More on next slide)

Data Flow Style (continued)

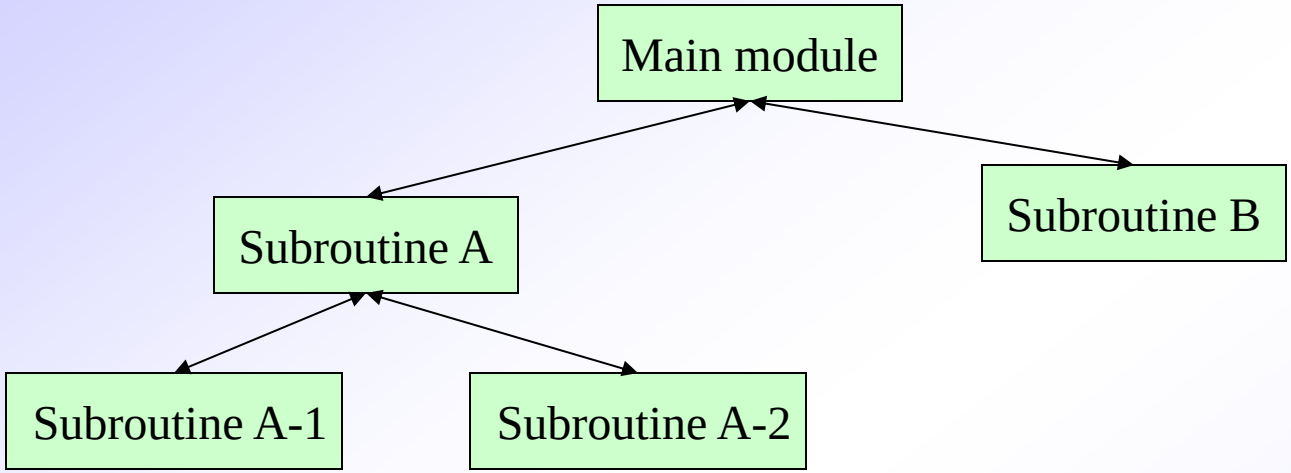
- Advantages
  - Has a simplistic design in the limited ways in which the components interact with the environment
  - Consists of no more and no less than the construction of its parts
  - Simplifies reuse and maintenance
  - Is easily made into a parallel or distributed execution in order to enhance system performance
- Disadvantages
  - Implicitly encourages a batch mentality so interactive applications are difficult to create in this style
  - Ordering of filters can be difficult to maintain so the filters cannot cooperatively interact to solve a problem
  - Exhibits poor performance
    - Filters typically force the least common denominator of data representation (usually ASCII stream)
    - Filter may need unlimited buffers if they cannot start producing output until they receive all of the input
    - Each filter operates as a separate process or procedure call, thus incurring overhead in set-up and take-down time

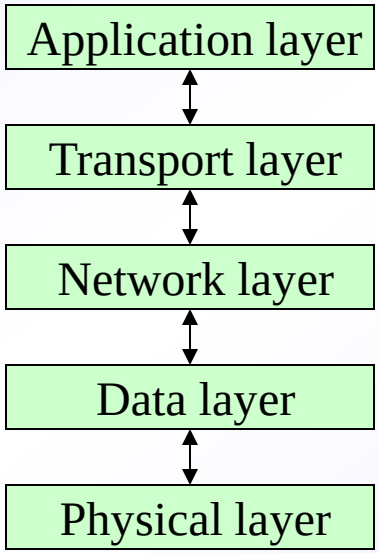
(More on next slide)

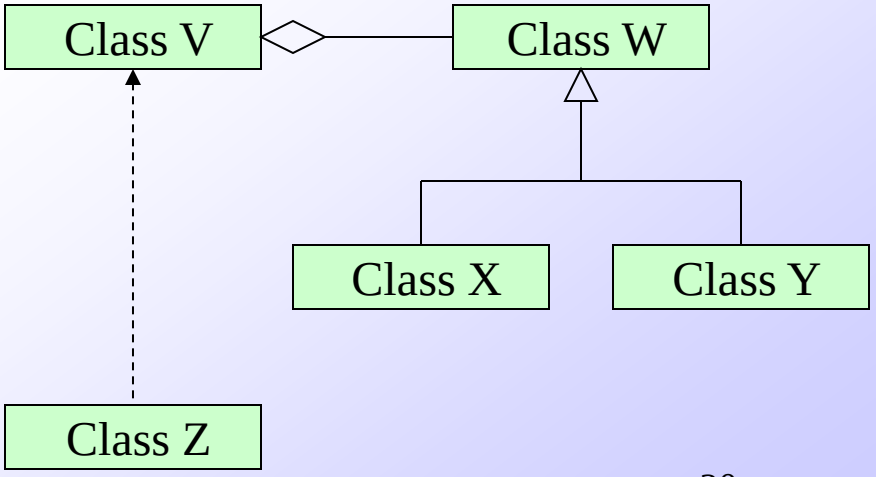
Data Flow Style (continued)

- Use this style when it makes sense to view your system as one that produces a well-defined easily identified output
  - The output should be a direct result of sequentially transforming a well-defined easily identified input in a time-independent fashion

Call-and-Return Style







Call-and-Return Style

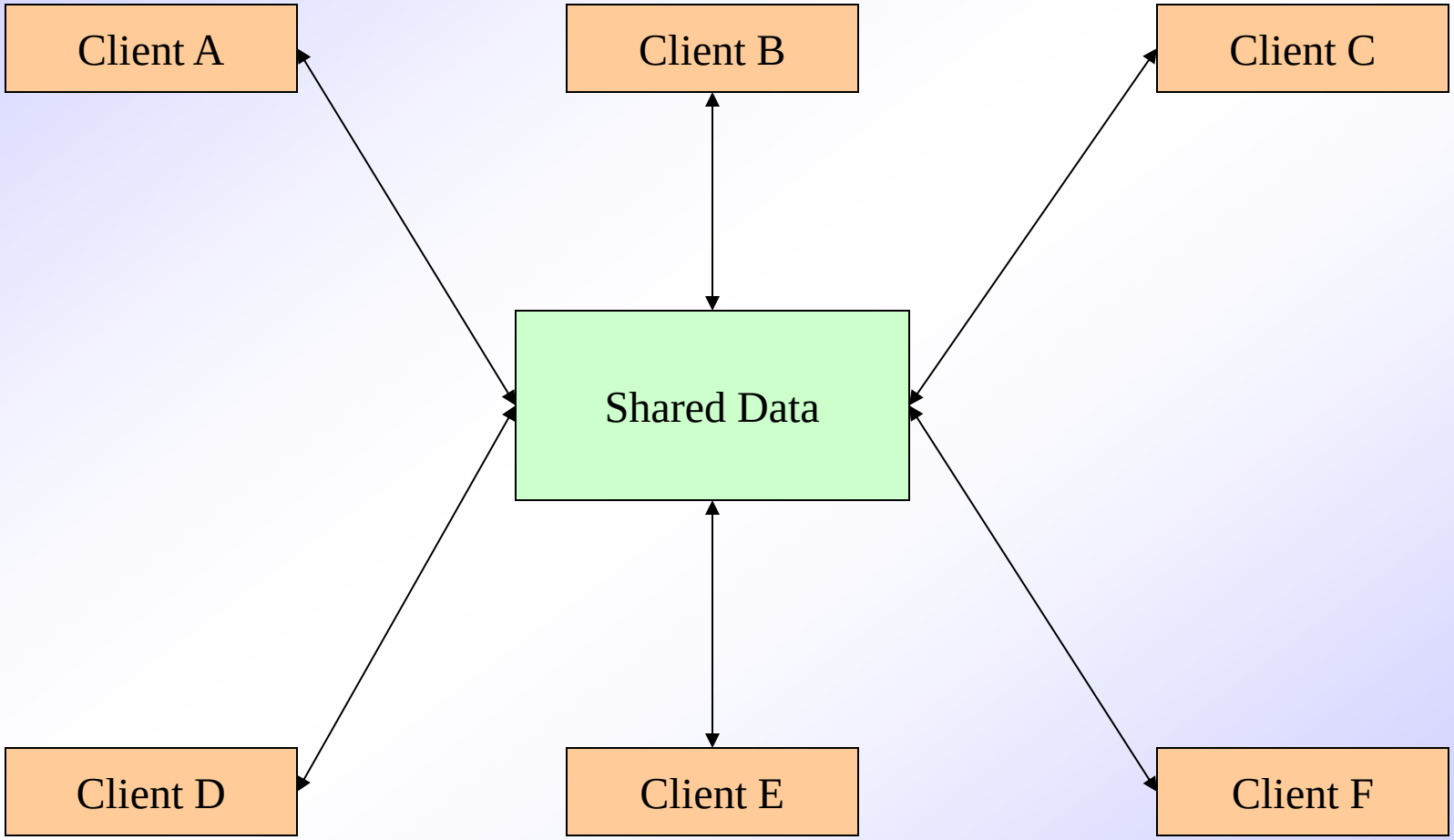
- Has the goal of modifiability and scalability
- Has been the dominant architecture since the start of software development
- Main program and subroutine style
  - Decomposes a program hierarchically into small pieces (i.e., modules)
  - Typically has a single thread of control that travels through various components in the hierarchy
- Remote procedure call style
  - Consists of main program and subroutine style of system that is decomposed into parts that are resident on computers connected via a network
  - Strives to increase performance by distributing the computations and taking advantage of multiple processors
  - Incurs a finite communication time between subroutine call and response

(More on next slide)

Call-and-Return Style (continued)

- Object-oriented or abstract data type system
  - Emphasizes the bundling of data and how to manipulate and access data
  - Keeps the internal data representation hidden and allows access to the object only through provided operations
  - Permits inheritance and polymorphism
- Layered system
  - Assigns components to layers in order to control inter-component interaction
  - Only allows a layer to communicate with its immediate neighbor
  - Assigns core functionality such as hardware interfacing or system kernel operations to the lowest layer
  - Builds each successive layer on its predecessor, hiding the lower layer and providing services for the upper layer
  - Is compromised by layer bridging that skips one or more layers to improve runtime performance
- Use this style when the order of computation is fixed, when interfaces are specific, and when components can make no useful progress while awaiting the results of request to other components

Data-Centered Style



Data-Centered Style (continued)

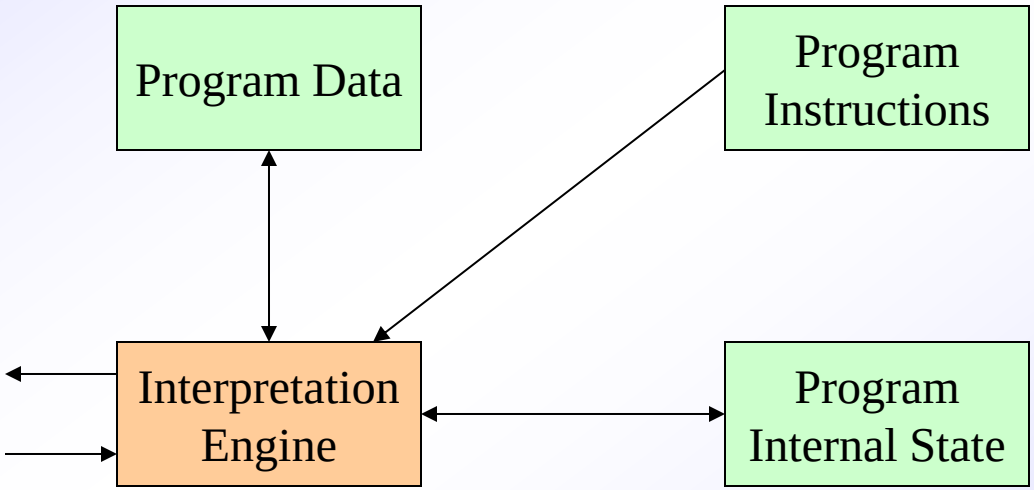
- Has the goal of integrating the data
- Refers to systems in which the access and update of a widely accessed data store occur
- A client runs on an independent thread of control
- The shared data may be a passive repository or an active blackboard
  - A blackboard notifies subscriber clients when changes occur in data of interest
- At its heart is a centralized data store that communicates with a number of clients
- Clients are relatively independent of each other so they can be added, removed, or changed in functionality
- The data store is independent of the clients

(More on next slide)

Data-Centered Style (continued)

- Use this style when a central issue is the storage, representation, management, and retrieval of a large amount of related persistent data
- Note that this style becomes client/server if the clients are modeled as independent processes

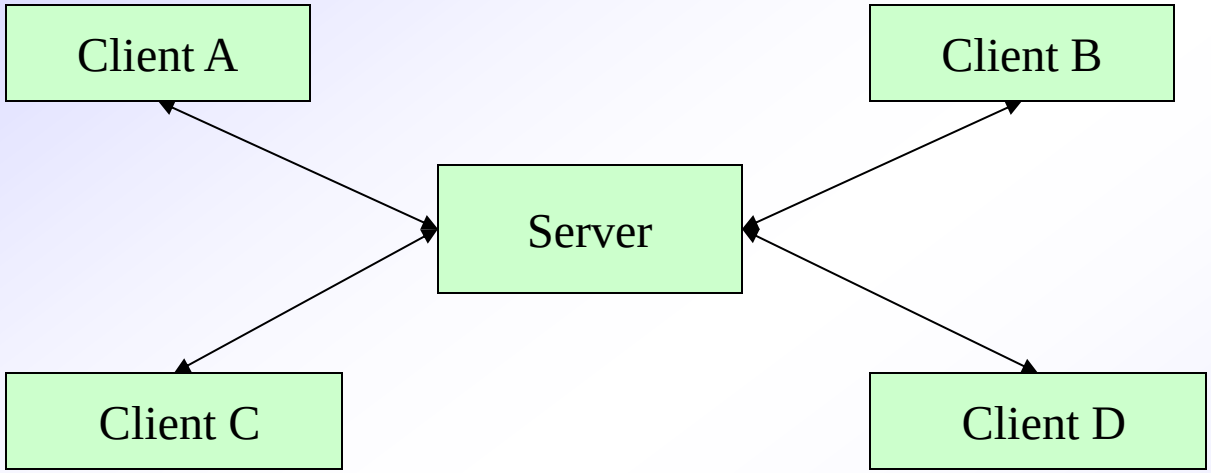
Virtual Machine Style

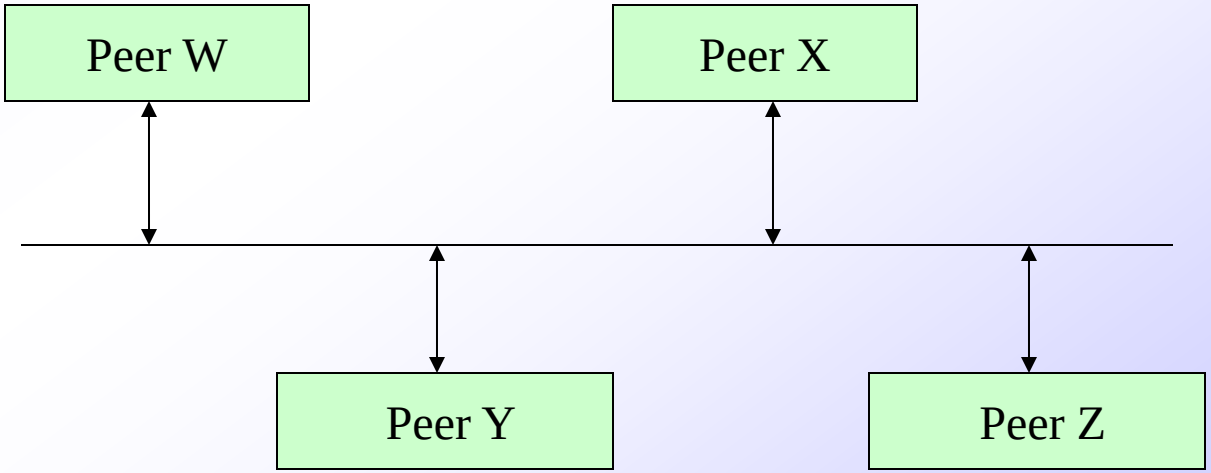


Virtual Machine Style

- Has the goal of portability
- Software systems in this style simulate some functionality that is not native to the hardware and/or software on which it is implemented
  - Can simulate and test hardware platforms that have not yet been built
  - Can simulate "disaster modes" as in flight simulators or safety-critical systems that would be too complex, costly, or dangerous to test with the real system
- Examples include interpreters, rule-based systems, and command language processors
- Interpreters
  - Add flexibility through the ability to interrupt and query the program and introduce modifications at runtime
  - Incur a performance cost because of the additional computation involved in execution
- Use this style when you have developed a program or some form of computation but have no make of machine to directly run it on

Independent Component Style





Independent Component Style

- Consists of a number of independent processes that communicate through messages
- Has the goal of modifiability by decoupling various portions of the computation
- Sends data between processes but the processes do not directly control each other
- Event systems style
  - Individual components announce data that they wish to share (publish) with their environment
  - The other components may register an interest in this class of data (subscribe)
  - Makes use of a message component that manages communication among the other components
  - Components publish information by sending it to the message manager
  - When the data appears, the subscriber is invoked and receives the data
  - Decouples component implementation from knowing the names and locations of other components

(More on next slide)

Independent Component Style (continued)

- Communicating processes style
  - These are classic multi-processing systems
  - Well-know subtypes are client/server and peer-to-peer
  - The goal is to achieve scalability
  - A server exists to provide data and/or services to one or more clients
  - The client originates a call to the server which services the request
- Use this style when
  - Your system has a graphical user interface
  - Your system runs on a multiprocessor platform
  - Your system can be structured as a set of loosely coupled components
  - Performance tuning by reallocating work among processes is important
  - Message passing is sufficient as an interaction mechanism among components

Heterogeneous Styles

- Systems are seldom built from a single architectural style
- Three kinds of heterogeneity
  - Locationally heterogeneous
    - The drawing of the architecture reveals different styles in different areas (e.g., a branch of a call-and-return system may have a shared repository)
  - Hierarchically heterogeneous
    - A component of one style, when decomposed, is structured according to the rules of a different style
  - Simultaneously heterogeneous
    - Two or more architectural styles may both be appropriate descriptions for the style used by a computer-based system

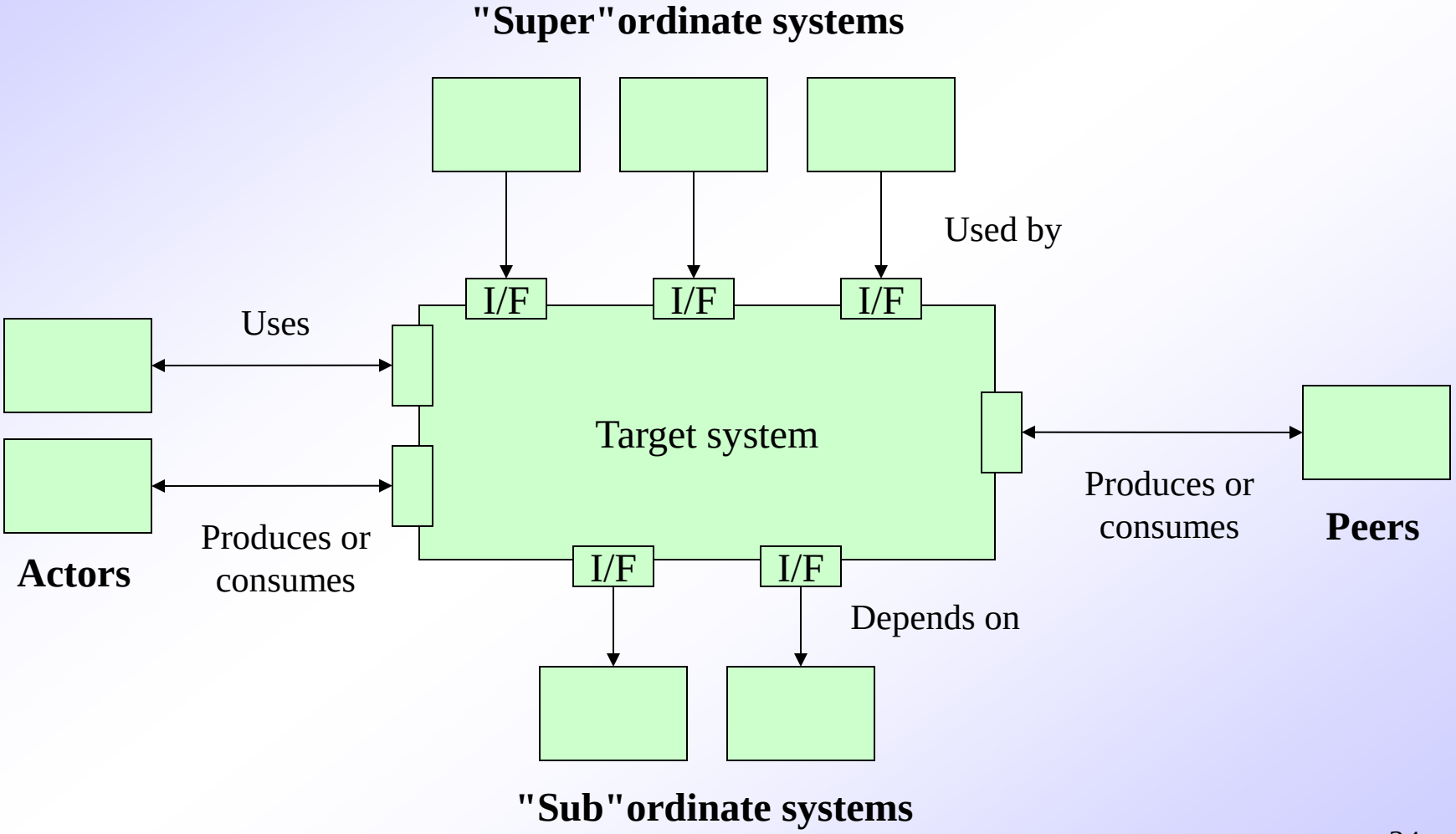
Architectural Design Process

Architectural Design Steps

1) Represent the system in context
2) Define archetypes
3) Refine the architecture into components
4) Describe instantiations of the system

"A doctor can bury his mistakes, but an architect can only advise his client to plant vines." Frank Lloyd Wright

1. Represent the System in Context



(More on next slide)

1. Represent the System in Context (continued)

- Use an architectural context diagram (ACD) that shows
  - The identification and flow of all information into and out of a system
  - The specification of all interfaces
  - Any relevant support processing from/by other systems
- An ACD models the manner in which software interacts with entities external to its boundaries
- An ACD identifies systems that interoperate with the target system
  - Super-ordinate systems
    - Use target system as part of some higher level processing scheme
  - Sub-ordinate systems
    - Used by target system and provide necessary data or processing
  - Peer-level systems
    - Interact on a peer-to-peer basis with target system to produce or consume data
  - Actors
    - People or devices that interact with target system to produce or consume data

2. Define Archetypes

- Archetypes indicate the important abstractions within the problem domain (i.e., they model information)
- An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system
- It is also an abstraction from a class of programs with a common structure and includes class-specific design strategies and a collection of example program designs and implementations
- Only a relatively small set of archetypes is required in order to design even relatively complex systems
- The target system architecture is composed of these archetypes
  - They represent stable elements of the architecture
  - They may be instantiated in different ways based on the behavior of the system
  - They can be derived from the analysis class model
- The archetypes and their relationships can be illustrated in a UML class diagram

Example Archetypes in Humanity

- Addict/Gambler
- Amateur
- Beggar
- Clown
- Companion
- Damsel in distress
- Destroyer
- Detective
- Don Juan
- Drunk
- Engineer
- Father
- Gossip
- Guide
- Healer
- Hero
- Judge
- King
- Knight
- Liberator/Rescuer
- Lover/Devotee
- Martyr
- Mediator
- Mentor/Teacher
- Messiah/Savior
- Monk/Nun
- Mother
- Mystic/Hermit
- Networker
- Pioneer
- Poet
- Priest/Minister
- Prince
- Prostitute
- Queen
- Rebel/Pirate
- Saboteur
- Samaritan
- Scribe/Journalist
- Seeker/Wanderer
- Servant/Slave
- Storyteller
- Student
- Trickster/Thief
- Vampire
- Victim
- Virgin
- Visionary/Prophet
- Warrior/Soldier

(Source: http://www.myss.com/ThreeArchs.asp)

Example Archetypes in Software Architecture

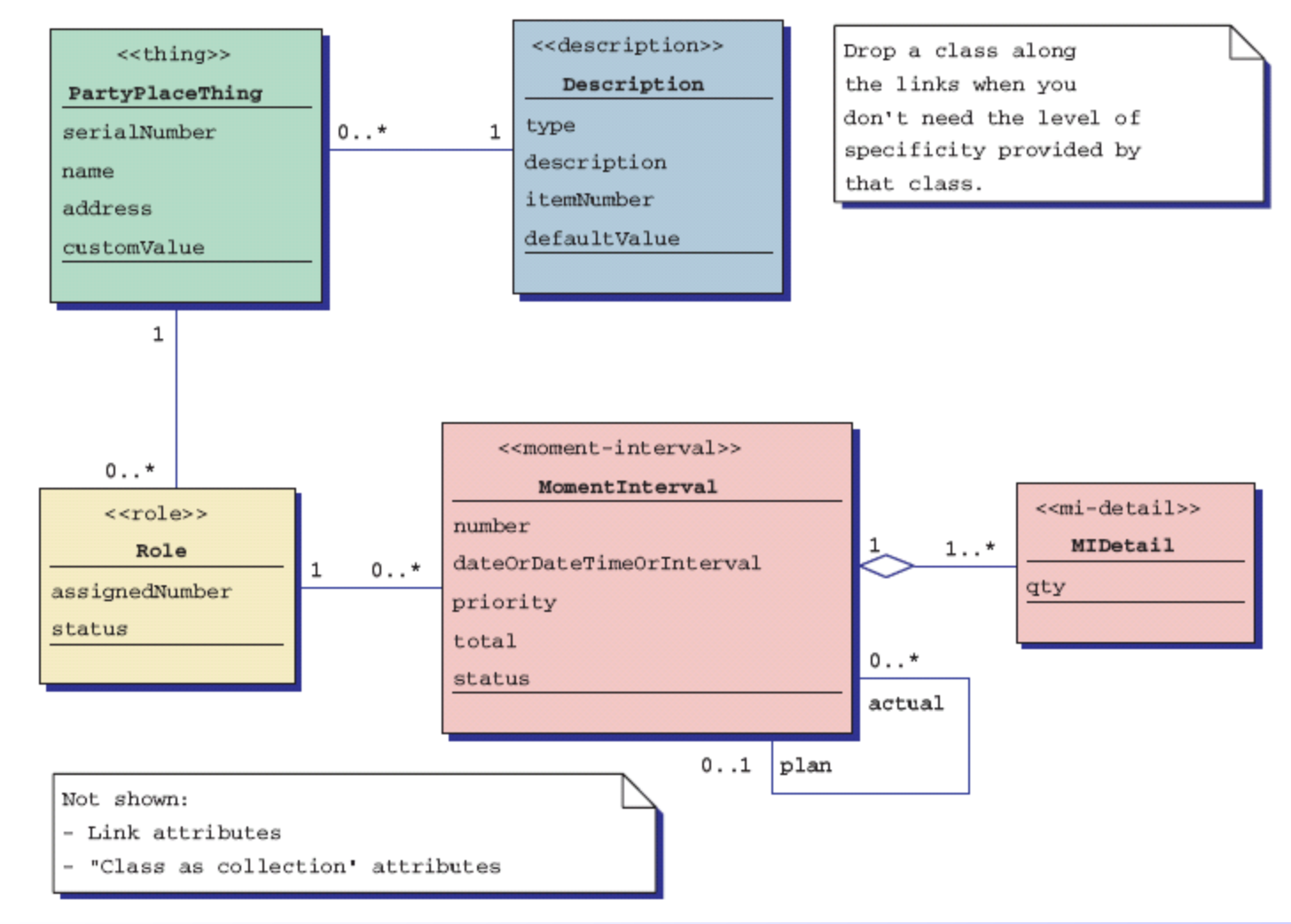
- Node
- Detector/Sensor
- Indicator
- Controller
- Manager

(Source: Pressman)

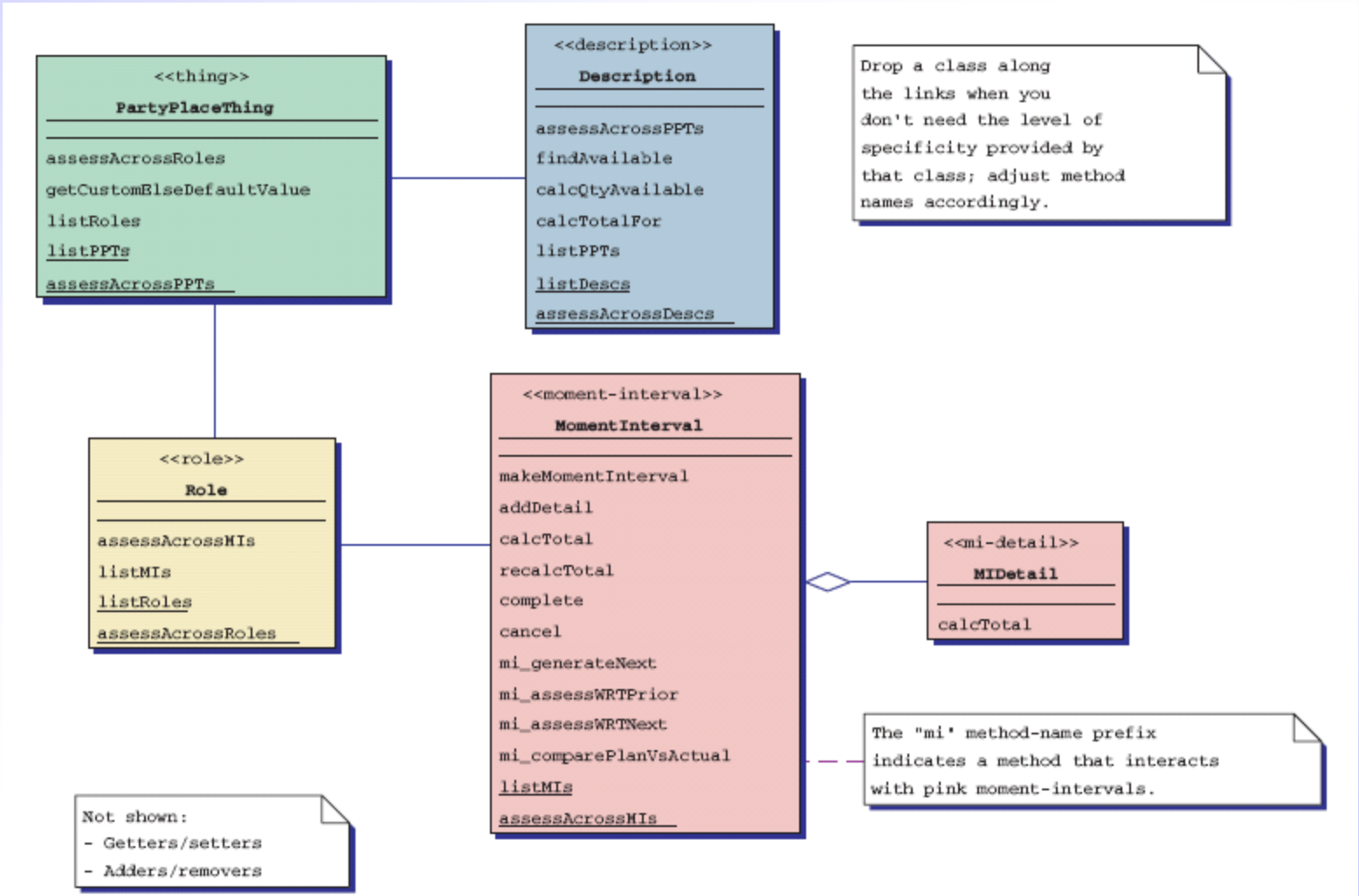
- Moment-Interval
- Role
- Description
- Party, Place, or Thing

(Source: Archetypes, Color, and the Domain Neutral Component)

Archetypes – their attributes



Archetypes – their methods



3. Refine the Architecture into Components

- Based on the archetypes, the architectural designer refines the software architecture into components to illustrate the overall structure and architectural style of the system
- These components are derived from various sources
  - The application domain provides application components, which are the domain classes in the analysis model that represent entities in the real world
  - The infrastructure domain provides design components (i.e., design classes) that enable application components but have no business connection
    - Examples: memory management, communication, database, and task management
  - The interfaces in the ACD imply one or more specialized components that process the data that flow across the interface
- A UML class diagram can represent the classes of the refined architecture and their relationships

4. Describe Instantiations of the System

- An actual instantiation of the architecture is developed by applying it to a specific problem
- This demonstrates that the architectural structure, style and components are appropriate
- A UML component diagram can be used to represent this instantiation

Assessing Alternative Architectural Designs

Various Assessment Approaches

A. Ask a set of questions that provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture
   - Assess the control in an architectural design (see next slide)
   - Assess the data in an architectural design (see upcoming slide)

B. Apply the architecture trade-off analysis method

C. Assess the architectural complexity

Approach A: Questions -- Assessing Control in an Architectural Design

- How is control managed within the architecture?
- Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?
- How do components transfer control within the system?
- How is control shared among components?
- What is the control topology (i.e., the geometric form that the control takes)?
- Is control synchronized or do components operate asynchronously?

Approach A: Questions -- Assessing Data in an Architectural Design

- How are data communicated between components?
- Is the flow of data continuous, or are data objects passed to the system sporadically?
- What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)
- Do data components exist (e.g., a repository or blackboard), and if so, what is their role?
- How do functional components interact with data components?
- Are data components passive or active (i.e., does the data component actively interact with other components in the system)?
- How do data and control interact within the system?

Approach B: Architecture Trade-off Analysis Method

1) Collect scenarios representing the system from the user's point of view
2) Elicit requirements, constraints, and environment description to be certain all stakeholder concerns have been addressed
3) Describe the candidate architectural styles that have been chosen to address the scenarios and requirements
4) Evaluate quality attributes by considering each attribute in isolation (reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability)
5) Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style by making small changes in the architecture
6) Critique the application of the candidate architectural styles (from step #3) using the sensitivity analysis conducted in step #5

Based on the results of steps 5 and 6, some architecture alternatives may be eliminated. Others will be modified and represented in more detail until a target architecture is selected

Approach C: Assessing Architectural Complexity

- The overall complexity of a software architecture can be assessed by considering the dependencies between components within the architecture
- These dependencies are driven by the information and control flow within a system
- Three types of dependencies
  - Sharing dependency **U ← → □ ← → V**
    - Represents a dependency relationship among consumers who use the same source or producer
  - Flow dependency **→ U → V →**
    - Represents a dependency relationship between producers and consumers of resources
  - Constrained dependency **U "XOR" V**
    - Represents constraints on the relative flow of control among a set of activities such as mutual exclusion between two components

Summary

- A software architecture provides a uniform, high-level view of the system to be built
- It depicts
  - The structure and organization of the software components
  - The properties of the components
  - The relationships (i.e., connections) among the components
- Software components include program modules and the various data representations that are manipulated by the program
- The choice of a software architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative architectures
- Data design translates the data objects defined in the analysis model into data structures that reside in the software

(More on next slide)

Summary (continued)

- A number of different architectural styles are available that encompass a set of component types, a set of connectors, semantic constraints, and a topological layout
- The architectural design process contains four distinct steps
  1) Represent the system in context
  2) Identify the component archetypes (the top-level abstractions)
  3) Identify and refine components within the context of various architectural styles
  4) Formulate a specific instantiation of the architecture
- Once a software architecture has been derived, it is elaborated and then analyzed against quality criteria

☺

Background

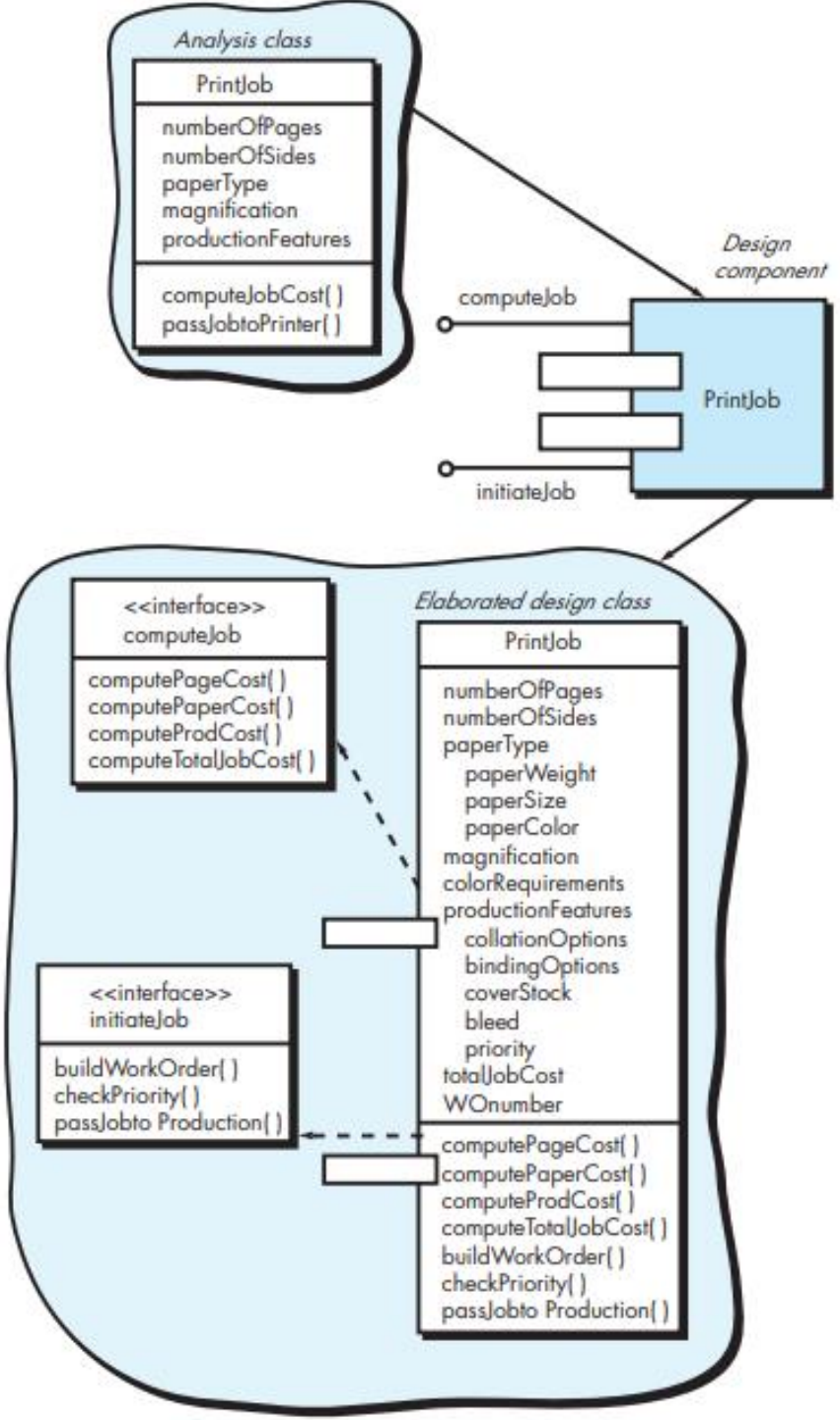
- Component-level design occurs after the first iteration of the architectural design
- It strives to create a design model from the analysis and architectural models
  - The translation can open the door to subtle errors that are difficult to find and correct later
  - "Effective programmers should not waste their time debugging – they should not introduce bugs to start with." Edsgar Dijkstra
- A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code
- The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors

The Software Component Defined

- A software component is a modular building block for computer software
  - It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- A component communicates and collaborates with
  - Other components
  - Entities outside the boundaries of the system
- Three different views of a component
  - An object-oriented view
  - A conventional view
  - A process-related view

Object-oriented View

- A component is viewed as a set of one or more collaborating classes
- Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation
  - This also involves defining the interfaces that enable classes to communicate and collaborate
- This elaboration activity is applied to every component defined as part of the architectural design
- Once this is completed, the following steps are performed
  1) Provide further elaboration of each attribute, operation, and interface
  2) Specify the data structure appropriate for each attribute
  3) Design the algorithmic detail required to implement the processing logic associated with each operation
  4) Design the mechanisms required to implement the interface to include the messaging that occurs between objects


Analysis class
PrintJob
numberOfPages
numberOfSides
paperType
magnification
productionFeatures
computeJobCost( )
passJobtoPrinter( )
Design component
computeJob
PrintJob
initiateJob
<<interface>>
computeJob
computePageCost( )
computePaperCost( )
computeProdCost( )
computeTotalJobCost( )
Elaborated design class
PrintJob
numberOfPages
numberOfSides
paperType
paperWeight
paperSize
paperColor
magnification
colorRequirements
productionFeatures
collationOptions
bindingOptions
coverStock
bleed
priority
totalJobCost
WOnumber
computePageCost( )
computePaperCost( )
computeProdCost( )
computeTotalJobCost( )
buildWorkOrder( )
checkPriority( )
passJobto Production( )
<<interface>>
initiateJob
buildWorkOrder( )
checkPriority( )
passJobto Production( )

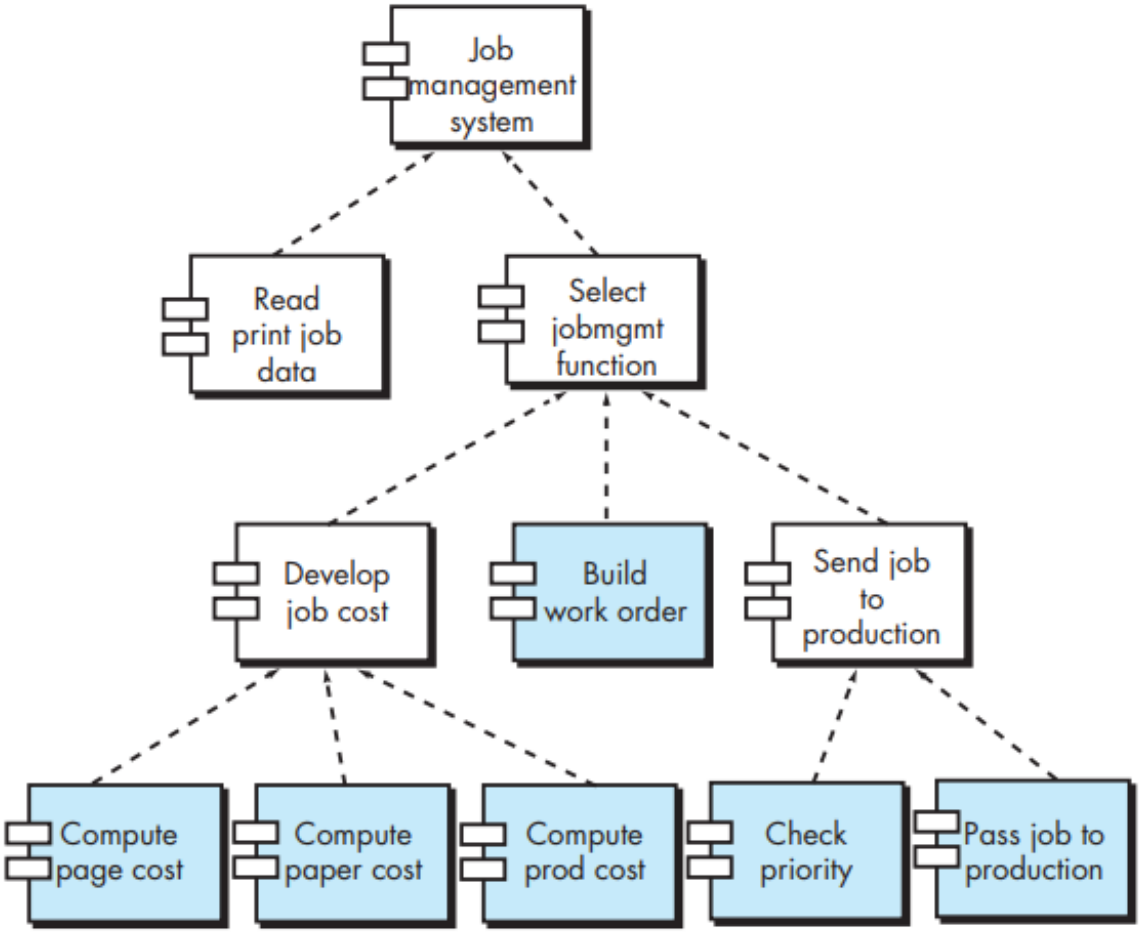
Conventional View

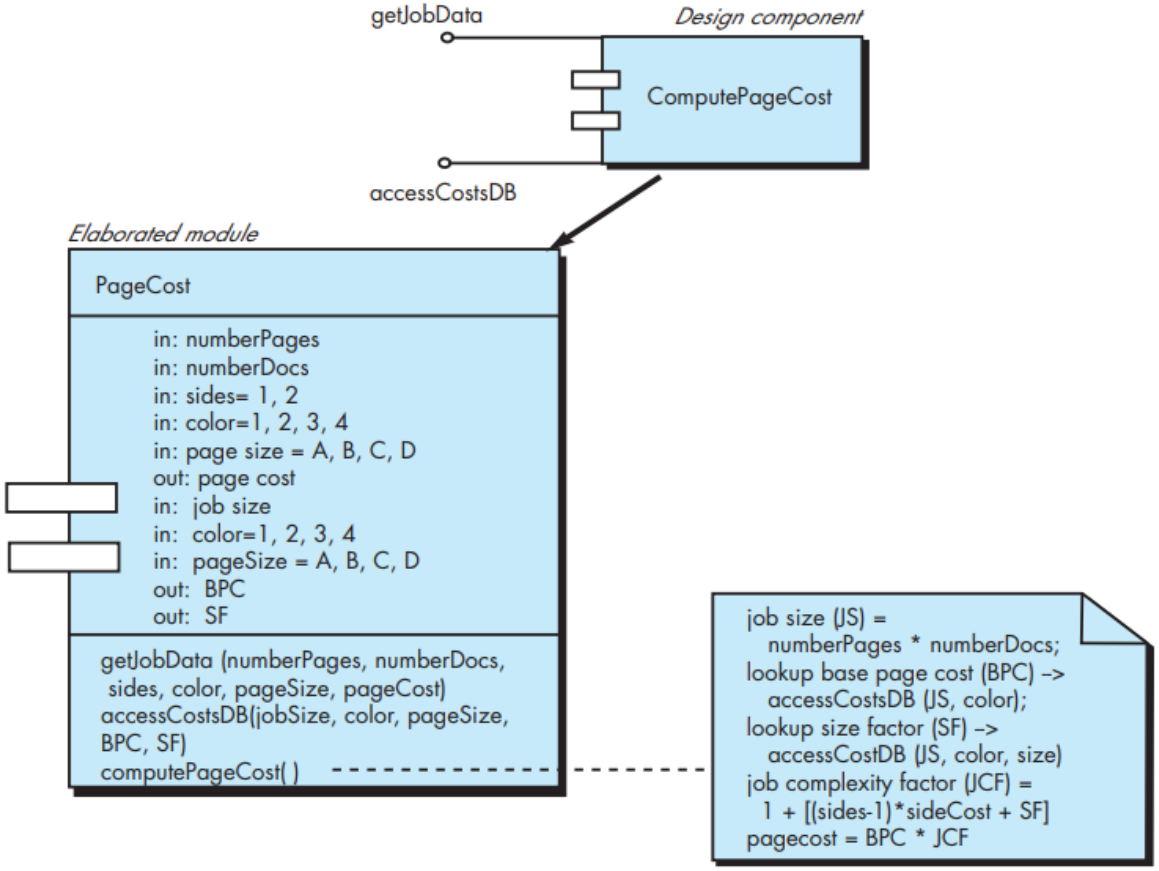
- A component is viewed as a functional element (i.e., a module) of a program that incorporates
  - The processing logic
  - The internal data structures that are required to implement the processing logic
  - An interface that enables the component to be invoked and data to be passed to it
- A component serves one of the following roles
  - A control component that coordinates the invocation of all other problem domain components
  - A problem domain component that implements a complete or partial function that is required by the customer
  - An infrastructure component that is responsible for functions that support the processing required in the problem domain

(More on next slide)

Conventional View (continued)

- Conventional software components are derived from the data flow diagrams (DFDs) in the analysis model
  - Each transform bubble (i.e., module) represented at the lowest levels of the DFD is mapped into a module hierarchy
  - Control components reside near the top
  - Problem domain components and infrastructure components migrate toward the bottom
  - Functional independence is strived for between the transforms
- Once this is completed, the following steps are performed for each transform
  1) Define the interface for the transform (the order, number and types of the parameters)
  2) Define the data structures used internally by the transform
  3) Design the algorithm used by the transform (using a stepwise refinement approach)


Job management system
Read print job data
Select jobmgmt function
Develop job cost
Build work order
Send job to production
Compute page cost
Compute paper cost
Compute prod cost
Check priority
Pass job to production



Component-level design for ComputePageCost

Process-related View

- Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
- As the software architecture is formulated, components are selected from the library and used to populate the architecture
- Because the components in the library have been created with reuse in mind, each contains the following:
  - A complete description of their interface
  - The functions they perform
  - The communication and collaboration they require

Process-related View

- A user interface (UI) component includes grids, buttons referred as controls, and utility components expose a specific subset of functions used in other components.
- Other common types of components are those that are resource intensive, not frequently accessed, and must be activated using the just-in-time (JIT) approach.
- Many components are invisible which are distributed in enterprise business applications and internet web applications such as Enterprise JavaBean (EJB), .NET components, and CORBA components.

Designing Class-Based Components
Component-level Design Principles

- **Open-closed principle**
  - A module or component should be open for extension but closed for modification
  - The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component
- **Liskov substitution principle**
  - Subclasses should be substitutable for their base classes
  - A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead
- **Dependency inversion principle**
  - Depend on abstractions (i.e., interfaces); do not depend on concretions
  - The more a component depends on other concrete components (rather than on the interfaces) the more difficult it will be to extend
- **Interface segregation principle**
  - Many client-specific interfaces are better than one general purpose interface
  - For a server class, specialized interfaces should be created to serve major categories of clients
  - Only those operations that are relevant to a particular category of clients should be specified in the interface

Component Packaging Principles

- Release reuse equivalency principle
  - The granularity of reuse is the granularity of release
  - Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
- Common closure principle
  - Classes that change together belong together
  - Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
- Common reuse principle
  - Classes that aren't reused together should not be grouped together
  - Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

Component-Level Design Guidelines

- Components
  - Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
  - Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
  - Use infrastructure component names that reflect their implementation-specific meaning (e.g., Stack)
- Dependencies and inheritance in UML
  - Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes)
  - Consider modeling any component dependencies as interfaces rather than representing them as a direct component-to-component dependency

Cohesion

- Cohesion is the "single-mindedness' of a component
- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- The objective is to keep cohesion as high as possible
- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)
  - Functional
    - A module performs one and only one computation and then returns a result
  - Layer
    - A higher layer component accesses the services of a lower layer component
  - Communicational
    - All operations that access the same data are defined within one class

(More on next slide)

Cohesion (continued)

- Kinds of cohesion (continued)
  - Sequential
    - Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations
  - Procedural
    - Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them
  - Temporal
    - Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected
  - Utility
    - Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

Coupling

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible

(More on next slide)

Coupling (continued)

- The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
  - Data coupling
    - Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
  - Stamp coupling
    - A whole data structure or class instantiation is passed as a parameter to an operation
  - Control coupling
    - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
    - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result
  - Common coupling
    - A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
  - Content coupling
    - One component secretly modifies data that is stored internally in another component

(More on next slide)

Coupling (continued)

- Other kinds of coupling (unranked)
  - Subroutine call coupling
    - When one operation is invoked it invokes another operation within side of it
  - Type use coupling
    - Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration
    - If/when the type definition changes, every component that declares a variable of that data type must also change
  - Inclusion or import coupling
    - Component A imports or includes the contents of component B
  - External coupling
    - A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

Conducting Component-Level Design

1) Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
2) Identify all design classes that correspond to the infrastructure domain
   - These classes are usually not present in the analysis or architectural models
   - These classes include GUI components, operating system components, data management components, networking components, etc.
3) Elaborate all design classes that are not acquired as reusable components
   a) Specify message details (i.e., structure) when classes or components collaborate
   b) Identify appropriate interfaces (e.g., abstract classes) for each component
   c) Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
   d) Describe processing flow within each operation in detail by means of pseudocode or UML activity diagrams

(More on next slide)

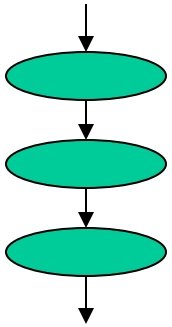
Conducting Component-Level Design (continued)

4) Describe persistent data sources (databases and files) and identify the classes required to manage them
5) Develop and elaborate behavioral representations for a class or component
   - This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class
6) Elaborate deployment diagrams to provide additional implementation detail
   - Illustrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environments
7) Factor every component-level design representation and always consider alternatives
   - Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model
   - The final decision can be made by using established design principles and guidelines

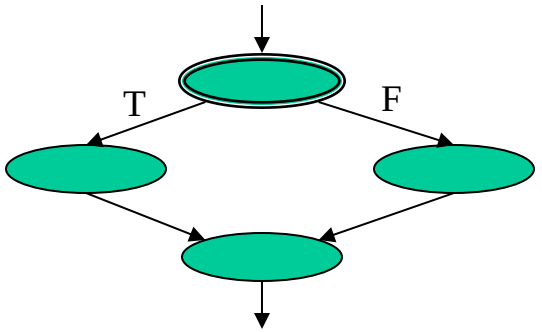
Designing Conventional Components Introduction

- Conventional design constructs emphasize the maintainability of a functional/procedural program
  - Sequence, condition, and repetition
- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow
- Various notations depict the use of these constructs
  - Graphical design notation
    - Sequence, if-then-else, selection, repetition (see next slide)
  - Tabular design notation (see upcoming slide)
  - Program design language
    - Similar to a programming language; however, it uses narrative text embedded directly within the program statements

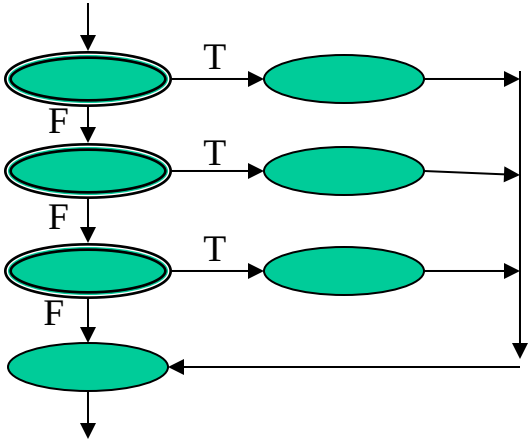
Graphical Design Notation



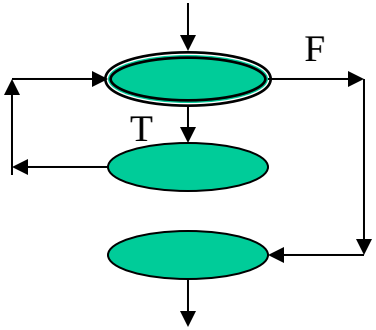
Sequence



If-then-else



Selection



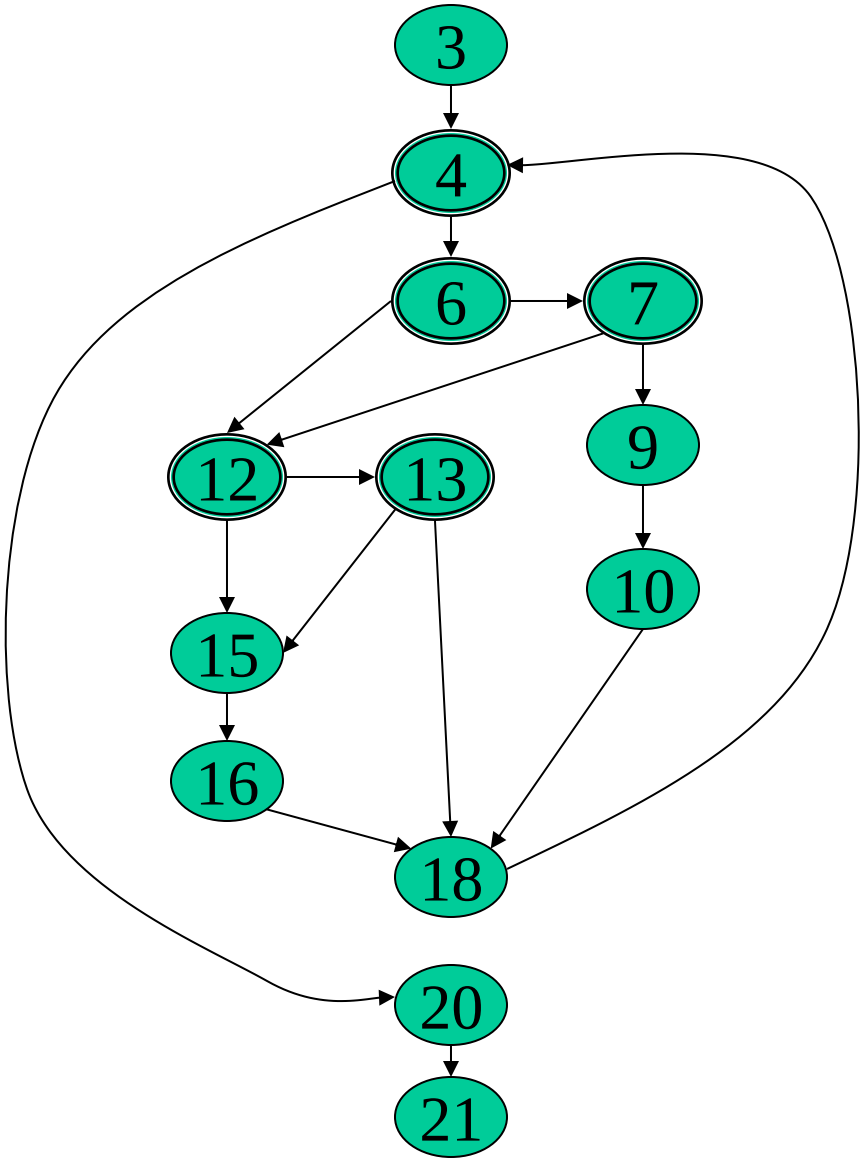
Repetition

Graphical Example used for Algorithm Analysis

```
 1  int functionZ(int y)
 2  {
 3  int x = 0;

 4  while (x <= (y * y))
 5     {
 6     if ((x % 11 == 0) &&
 7          (x % y == 0))
 8        {
 9        printf("%d", x);
10        x++;
11        } // End if
12     else if ((x % 7 == 0) ||
13               (x % y == 1))
14        {
15        printf("%d", y);
16        x = x + 2;
17        } // End else
18     printf("\n");
19     } // End while

20  printf("End of list\n");
21  return 0;
22  } // End functionZ
```



Tabular Design Notation

1) List all actions that can be associated with a specific procedure (or module)
2) List all conditions (or decisions made) during execution of the procedure
3) Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions
4) Define rules by indicating what action(s) occurs for a set of conditions

Conditions	1	2	3	4
Condition A	T	T		F
Condition B		F	T	
Condition C	T			T
Actions				
Action X	✓		✓	
Action Y				✓
Action Z	✓	✓		✓

(More on next slide)

Coding Principles

Coding

The coding is the process of transforming the design of a system into a computer language format. This coding phase of software development is concerned with software translating design specification into the source code. It is necessary to write source code & internal documentation so that conformance of the code to its specification can be easily verified.

Coding is done by the coder or programmers who are independent people than the designer. The goal is not to reduce the effort and cost of the coding phase, but to cut to the cost of a later stage. The cost of testing and maintenance can be significantly reduced with efficient coding.

Goals of Coding

1. **To translate the design of system into a computer language format:** The coding is the process of transforming the design of a system into a computer language format, which can be executed by a computer and that perform tasks as specified by the design of operation during the design phase.
2. **To reduce the cost of later phases:** The cost of testing and maintenance can be significantly reduced with efficient coding.
3. **Making the program more readable:** Program should be easy to read and understand. It increases code understanding having readability and understandability as a clear objective of the coding activity can itself help in producing more maintainable software.

- For implementing our design into code, we require a high-level functional language. A programming language should have the following characteristics:
- Characteristics of Programming Language
  - Following are the characteristics of Programming Language:

Characteristics of Programming Language

Readability: A good high-level language will allow programs to be written in some methods that resemble a quite-English description of the underlying functions. The coding may be done in an essentially self-documenting way.

Portability: High-level languages, being virtually machine-independent, should be easy to develop portable software.

Generality: Most high-level languages allow the writing of a vast collection of programs, thus relieving the programmer of the need to develop into an expert in many diverse languages.

Brevity: Language should have the ability to implement the algorithm with less amount of code. Programs mean in high-level languages are often significantly shorter than their low-level equivalents.

Error checking: A programmer is likely to make many errors in the development of a computer program. Many high-level languages invoke a lot of bugs checking both at compile-time and run-time.

Cost: The ultimate cost of a programming language is a task of many of its characteristics.

Quick translation: It should permit quick translation.

Efficiency: It should authorize the creation of an efficient object code.

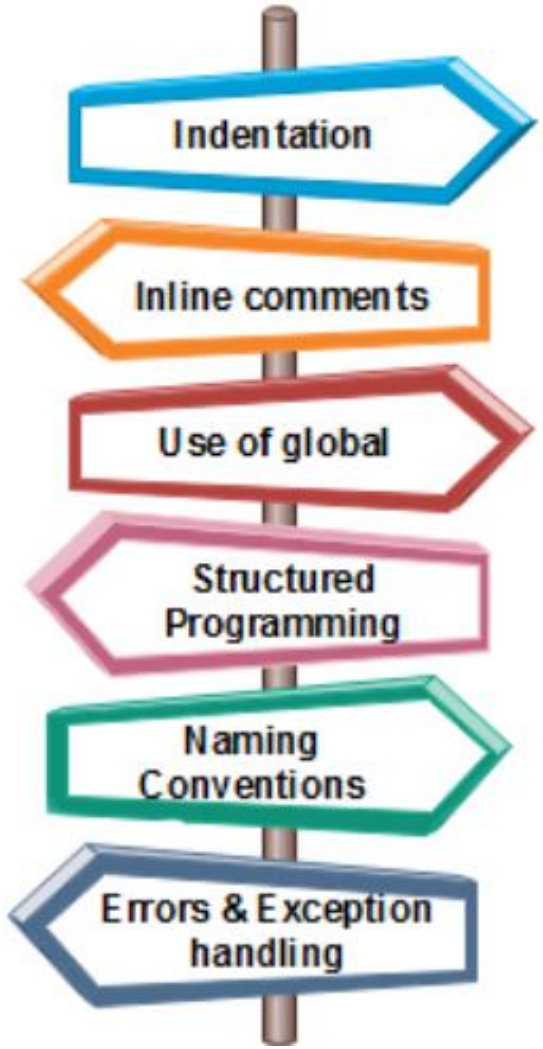
Modularity: It is desirable that programs can be developed in the language as several separately compiled modules, with the appropriate structure for ensuring self-consistency among these modules.

Widely available: Language should be widely available, and it should be feasible to provide translators for all the major machines and all the primary operating systems.

Coding Standards

- General coding standards refers to how the developer writes code, so here we will discuss some essential standards regardless of the programming language being used.
- **The following are some representative coding standards:**

Coding Standards



1. **Indentation:** Proper and consistent indentation is essential in producing easy to read and maintainable programs.
   Indentation should be used to:
   - Emphasize the body of a control structure such as a loop or a select statement.
   - Emphasize the body of a conditional statement
   - Emphasize a new scope block
2. **Inline comments:** Inline comments analyze the functioning of the subroutine, or key aspects of the algorithm shall be frequently used.
3. **Rules for limiting the use of global:** These rules file what types of data can be declared global and what cannot.

4. **Structured Programming:** Structured (or Modular) Programming methods shall be used. "GOTO" statements shall not be used as they lead to "spaghetti" code, which is hard to read and maintain, except as outlined line in the FORTRAN Standards and Guidelines.
5. **Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always begin with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
6. **Error return conventions and exception handling system:** Different functions in a program report the way error conditions are handled should be standard within an organization. For example, different tasks while encountering an error condition should either return a 0 or 1 consistently.

Coding Guidelines

General coding guidelines provide the programmer with a set of the best methods which can be used to make programs more comfortable to read and maintain. Most of the examples use the C language syntax, but the guidelines can be tested to all languages.

The following are some representative coding guidelines recommended by many software development organizations

1. Line Length: It is considered a good practice to keep the length of source code lines at or below 80 characters. Lines longer than this may not be visible properly on some terminals and tools. Some printers will truncate lines longer than 80 columns.

2. Spacing: The appropriate use of spaces within a line of code can improve readability.

Example:

Bad:
```
cost=price+(price*sales_tax)
fprintf(stdout ,"The total cost is %5.2f\n",cost);
```

Better:
```
cost = price + ( price * sales_tax )
fprintf (stdout,"The total cost is %5.2f\n",cost);
```

3. The code should be well-documented: As a rule of thumb, there must be at least one comment line on the average for every three-source line.

4. The length of any function should not exceed 10 source lines: A very lengthy function is generally very difficult to understand as it possibly carries out many various functions. For the same reason, lengthy functions are possible to have a disproportionately larger number of bugs.

5. Do not use goto statements: Use of goto statements makes a program unstructured and very tough to understand.

6. Inline Comments: Inline comments promote readability.

7. Error Messages: Error handling is an essential aspect of computer programming. This does not only include adding the necessary logic to test for and handle errors but also involves making error messages meaningful.