# UNIT 4 : Requirement Analysis

4.1 UML (Class Diagram, Use Case)

4.2 DFD, Data Dictionary and Process Specification.

4.3 Design model.

4.4 Principal and Concepts.

4.5 Functional Independence.

4.6 Effectiveness of Modular Design.

# 4.1 UML(Class Diagram,Use Case)

UML Diagrams :

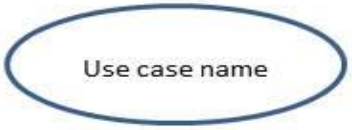
- Use Case Diagram
- Activity Diagram
- Class Diagram

# UML Diagrams :

- Stands for Unified Modelling Language
- The Unified Modeling Language is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system.

# Use Case Diagram


Actor



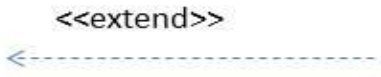

Use case

Generalization symbol used between actors and between use cases



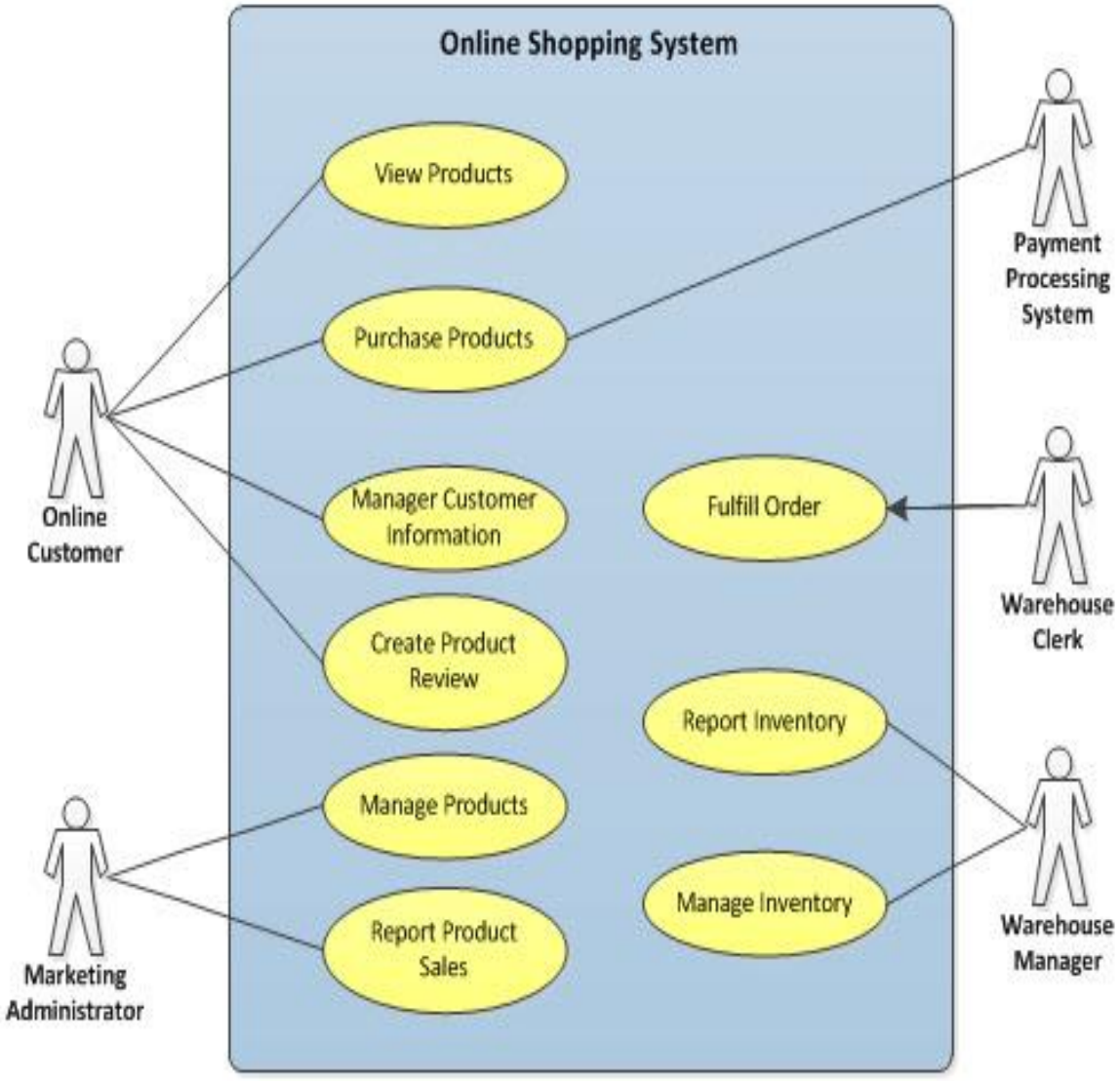
Association between actor and use case


Include relationship between use cases

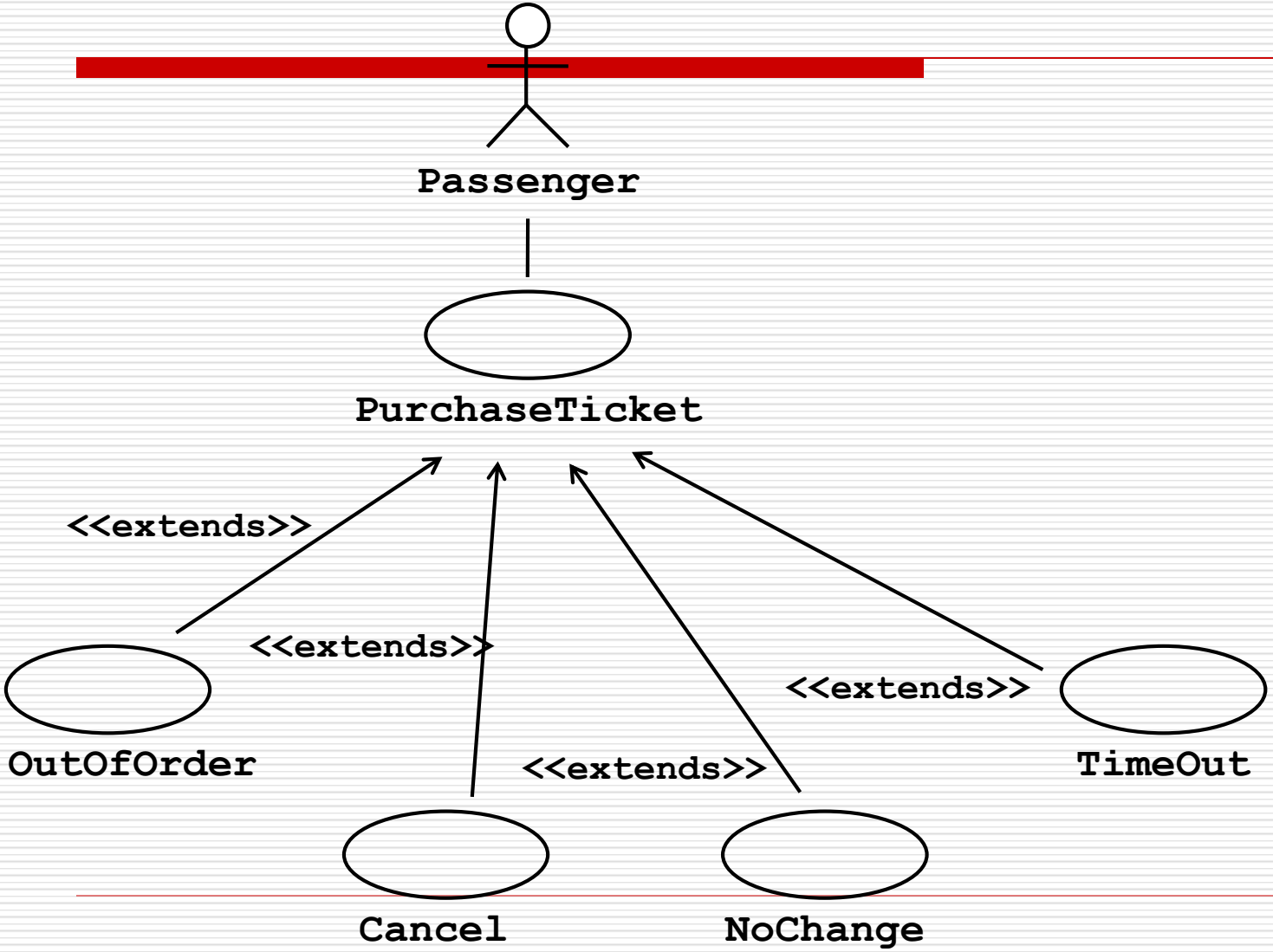



Extend relationship between use cases

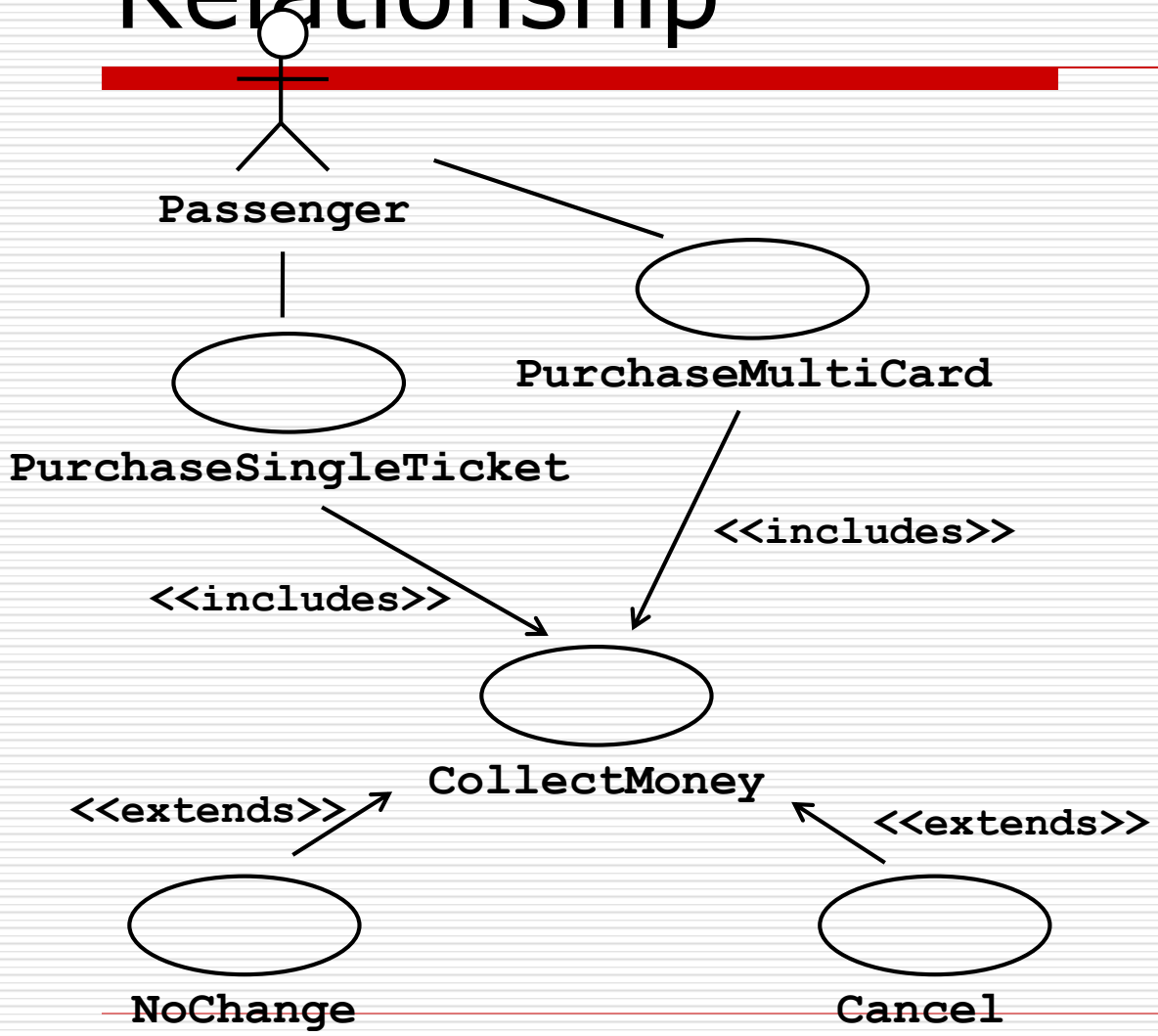
**Symbols in a use case diagram**


Online Shopping System
View Products
Purchase Products
Manager Customer Information
Create Product Review
Manage Products
Report Product Sales
Fulfill Order
Report Inventory
Manage Inventory
Online Customer
Marketing Administrator
Payment Processing System
Warehouse Clerk
Warehouse Manager

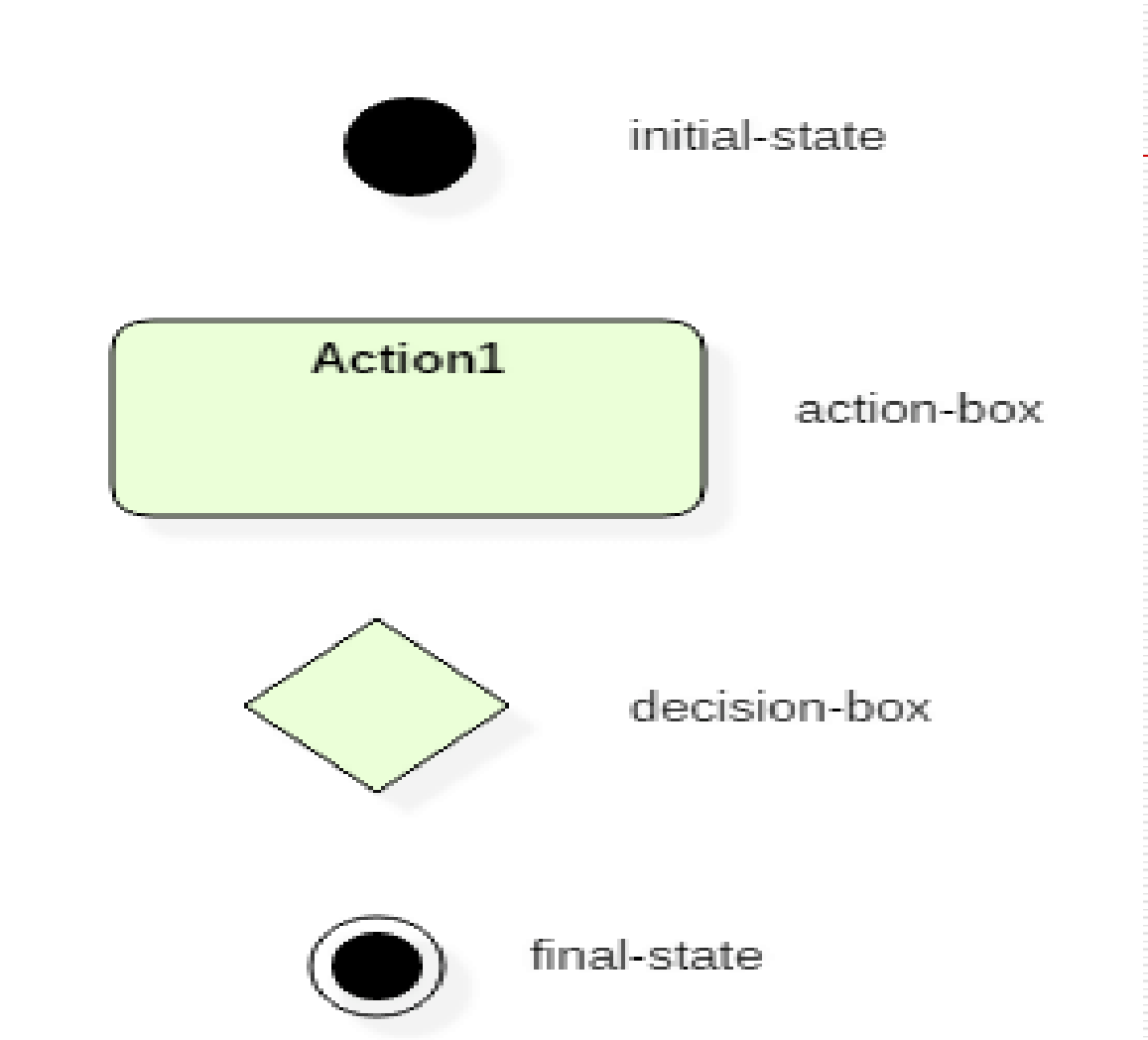
# The *<<extends>>* Relationship



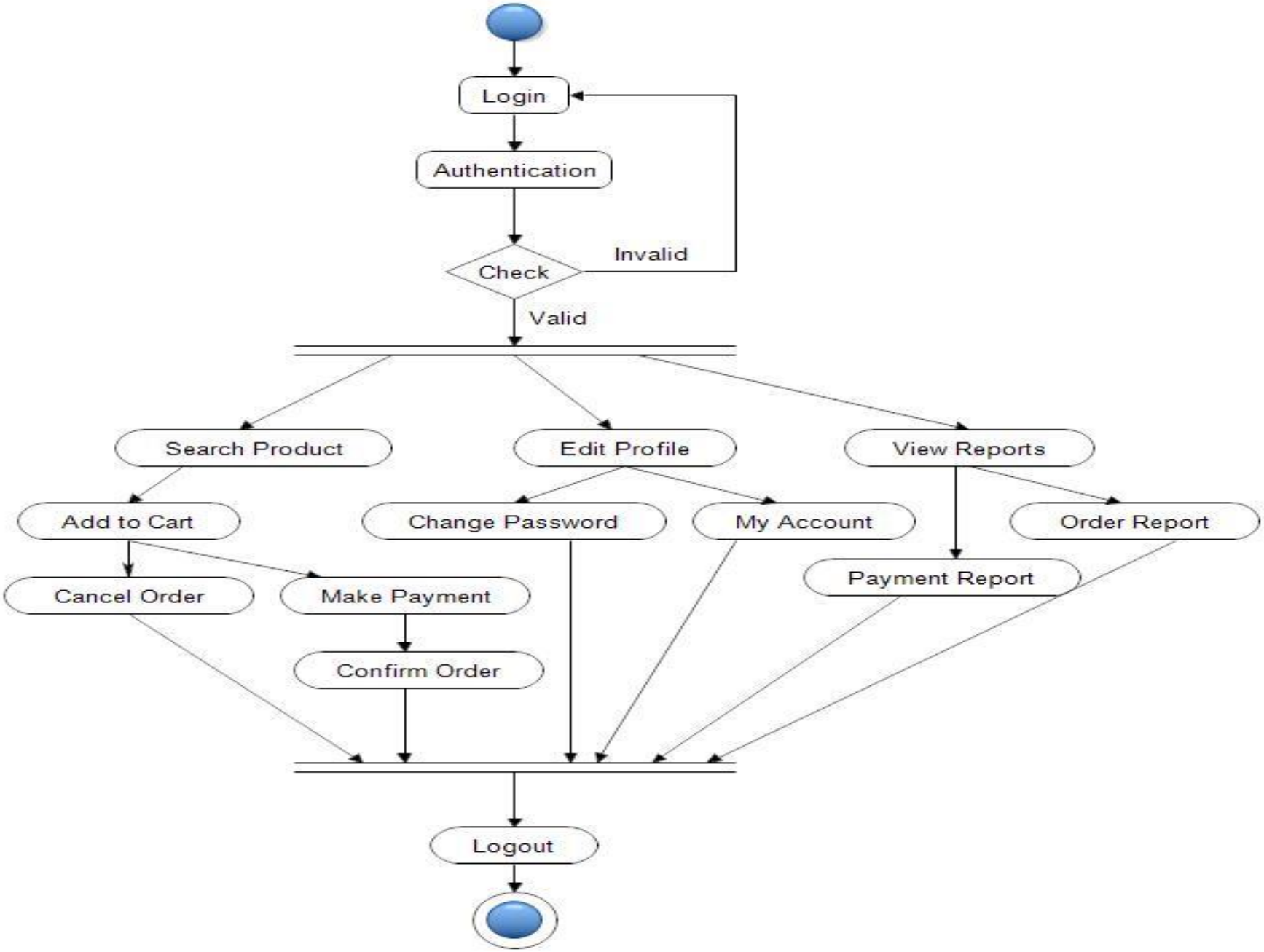
# The *<<includes>>* Relationship



# Activity Diagram



# Activity Diagram for User Side



- Draw Use Case and Activity Diagram for Exam Management System

# CLASS DIAGRAM

- The class diagram depicts a static view of an application. It represents the types of objects residing in the system and the relationships between them. A class consists of its objects, and also it may inherit from other classes.
- A class diagram is used to visualize, describe, document various different aspects of the system, and also construct executable software code.
- It shows the attributes, classes, functions, and relationships to give an overview of the software system.

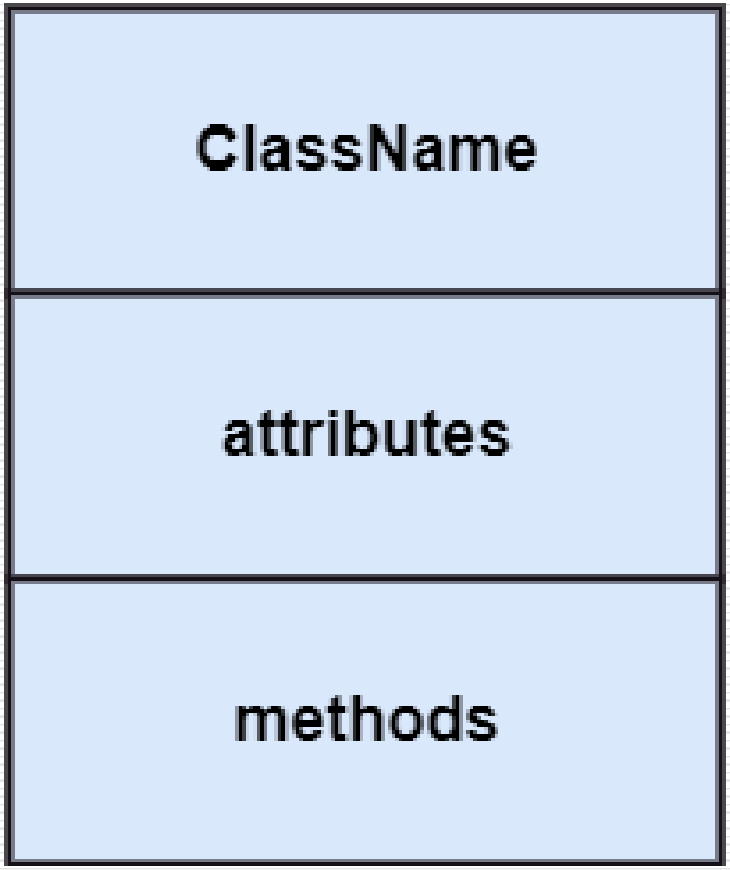
# Purpose of Class Diagram

- The main purpose of class diagrams is to build a static view of an application.
- It describes the major responsibilities of a system.
- It is a base for component and deployment diagrams.
- It incorporates forward and reverse engineering.

# Benefits of Class Diagram

- It can represent the object model for complex systems.
- It reduces the maintenance time by providing an overview of how an application is structured before coding.
- It provides a general schematic of an application for better understanding.
- It represents a detailed chart by highlighting the desired code, which is to be programmed.
- It is helpful for the stakeholders and the developers.

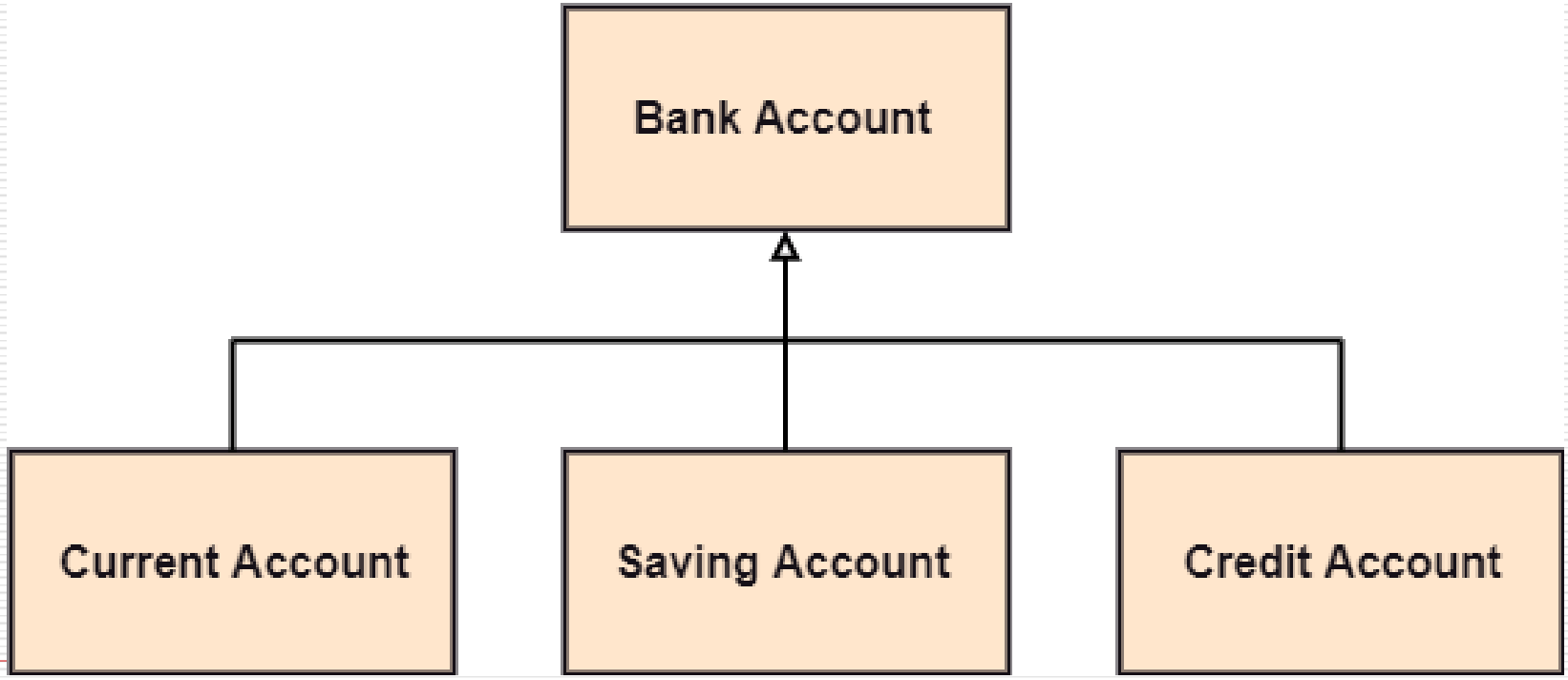
# The class diagram is made up of three sections:



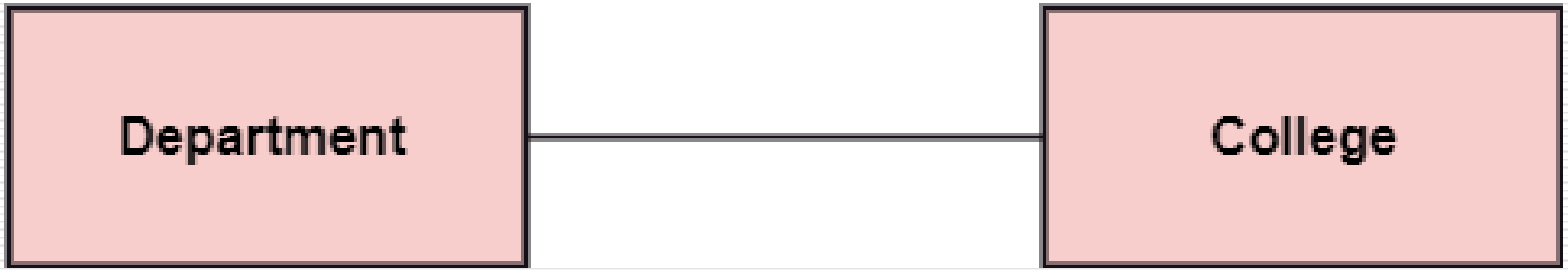
# Relationships

- **Dependency:** A dependency is a semantic relationship between two or more classes where a change in one class cause changes in another class. It forms a weaker relationship. In the following example, Student_Name is dependent on the Student_Id.

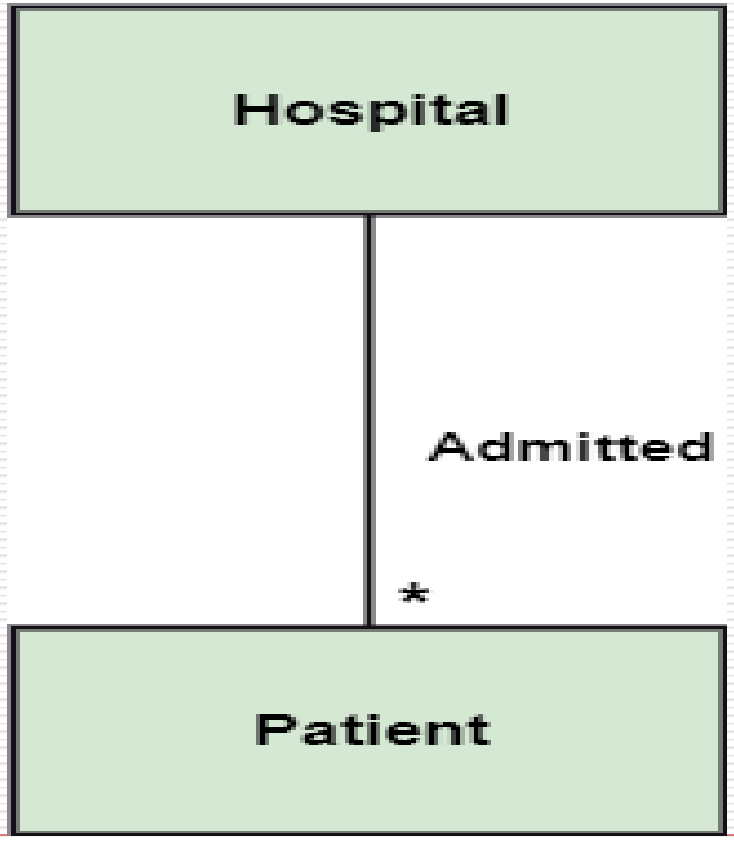
- **Generalization:** A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class. For example, The Current Account, Saving Account, and Credit Account are the generalized form of Bank Account.



- **Association:** It describes a static or physical connection between two or more objects. It depicts how many objects are there in the relationship. For example, a department is associated with the college.

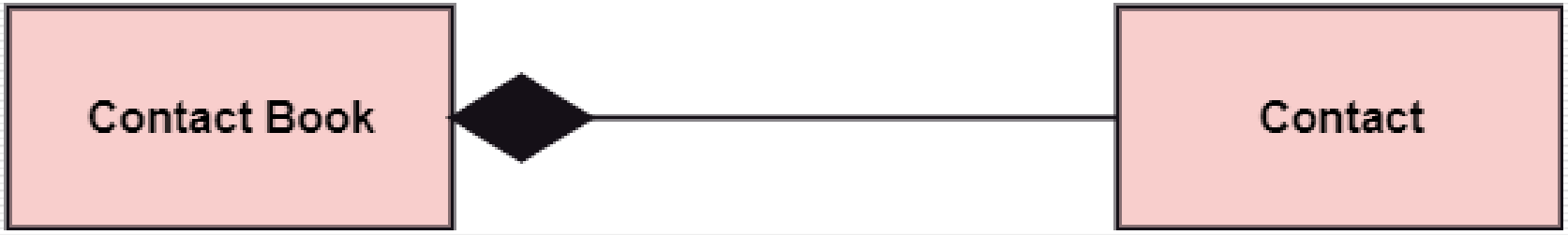


- **Multiplicity:** It defines a specific range of allowable instances of attributes. In case if a range is not specified, one is considered as a default multiplicity. For example, multiple patients are admitted to one hospital.



- **Aggregation:** An aggregation is a subset of association, which represents has a relationship. It is more specific then association. It defines a part-whole or part-of relationship. In this kind of relationship, the child class can exist independently of its parent class.
- The company encompasses a number of employees, and even if one employee resigns, the company still exists.

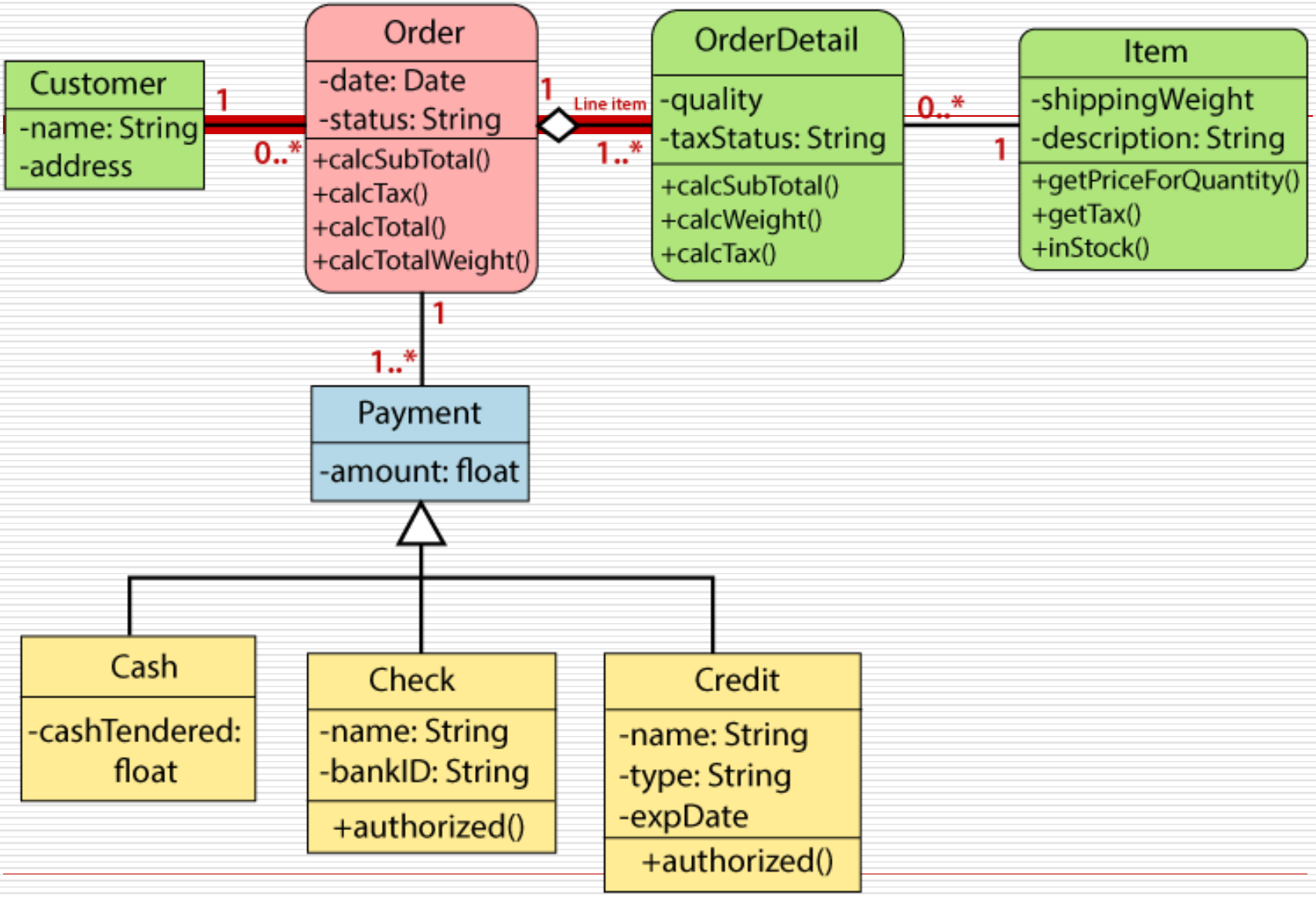
- **Composition:** The composition is a subset of aggregation. It portrays the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded. It represents a whole-part relationship.
- A contact book consists of multiple contacts, and if you delete the contact book, all the contacts will be lost.



# How to draw a Class Diagram?

- To describe a complete aspect of the system, it is suggested to give a meaningful name to the class diagram.
- The objects and their relationships should be acknowledged in advance.
- The attributes and methods (responsibilities) of each class must be known.
- A minimum number of desired properties should be specified as more number of the unwanted property will lead to a complex diagram.
- Notes can be used as and when required by the developer to describe the aspects of a diagram.
- The diagrams should be redrawn and reworked as many times to make it correct before producing its final version.

# Example


Customer
-name: String
-address
1
0..*
Order
-date: Date
-status: String
+calcSubTotal()
+calcTax()
+calcTotal()
+calcTotalWeight()
1
Line item
1..*
OrderDetail
-quality
-taxStatus: String
+calcSubTotal()
+calcWeight()
+calcTax()
0..*
1
Item
-shippingWeight
-description: String
+getPriceForQuantity()
+getTax()
+inStock()
1
1..*
Payment
-amount: float
Cash
-cashTendered: float
Check
-name: String
-bankID: String
+authorized()
Credit
-name: String
-type: String
-expDate
+authorized()

# 4.2 Data Flow Diagrams

- Through a structured analysis technique called data flow diagrams (DFD), the systems analyst can put together a graphical representation of data processes throughout the system.
- The data flow approach emphasizes the logic underlying the system.

- DFD is a graphical aid(help) for defining systems inputs, processes and outputs.
- It represents flow of data through the system.

# Purposes / Utility

- DFD is a graphical tool which can be used by the analyst to clarify his understanding of the system to the user.
- DFD can be easily converted into a structure chart which can be used in design phase.

# Two Types:

- Physical Data Flow Diagrams
- Logical Data Flow Diagrams

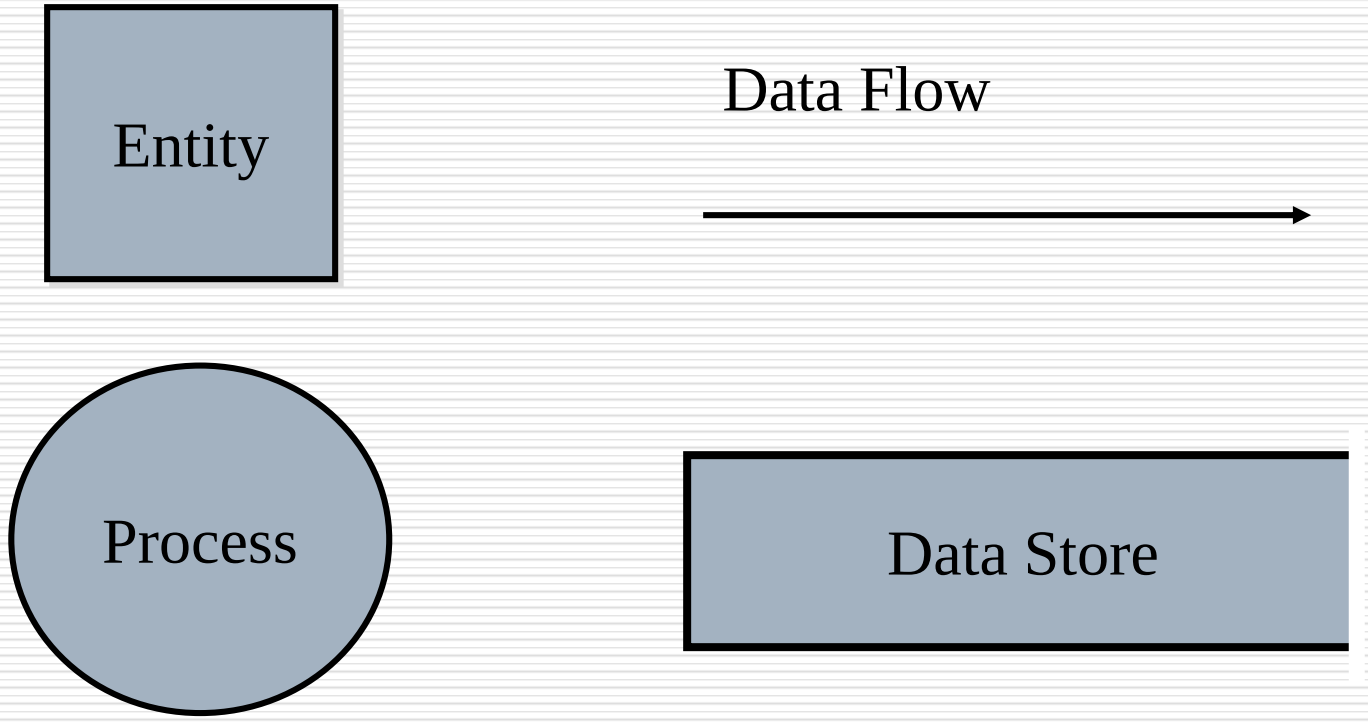
# Physical Data Flow Diagrams (**PDFD**)

- An implementation-dependent view of the current system, showing **what tasks** are carried out and how they are performed. Physical characteristics can include:
  - Names of people
  - Form and document names or numbers
  - Names of departments
  - Master and transaction files
  - Equipment and devices used
  - Locations
  - Names of procedures

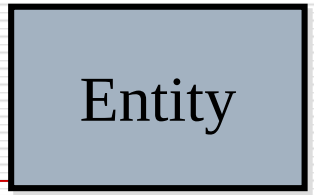
# Logical Data Flow Diagrams (**LDFD**)

- An implementation-independent view of the a system, focusing on the **flow of data** between **processes** without regard for the specific devices, storage locations or people in the system.
  - Order Processing
  - Account Verification
  - Login Data Verification

# Symbols used in DFDs



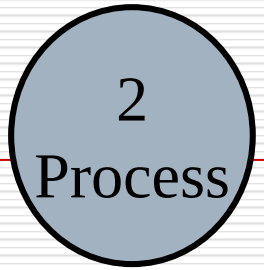
# Conventions used in DFDs




- Entity:
  - **An external entity (sink)** (e.g. a customer, supplier, department, employee) is a source or destination of a data flow which is outside the area of study. Only those entities which originate or receive data are represented on a data flow diagram. The symbol used is an square containing a meaningful and unique identifier.
- Data Flow:

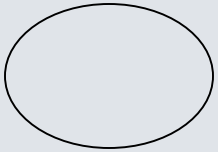
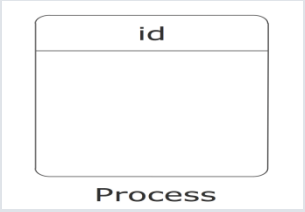
  - **A data flow** shows the flow of information from its source to its destination. A data flow is represented by a line, with arrowheads showing the direction of flow. Information always flows to or from a process and may be written, verbal or electronic.

# Conventions used in DFDs




- Process:
  - **A process** shows a transformation or manipulation of data flows within the system; hence the data flow leaving a process is always labeled differently from the one entering it. Processes represent work being performed within the system. The symbol used is a circle which contains 2 descriptive elements: a unique identifier and a descriptive name


- Data Store:
  - A data store can be thought of as data at rest. It is typically a database but it could be a filing cabinet, a notebook or a datafile. It could also be a permanent or a temporary store. It is represented by an open ended narrow rectangle.

| Symbol | Yourdan | Gane & Sarson |
| --- | --- | --- |
| Process |  |  |
| External Entity |  |  |
| Data Flow |  |  |
| Data Store |  |  |

**Difference Between Flowchart and Data Flow Diagram (DFD)**

| S.No | Flowchart | Data Flow Diagram (DFD) |
|---|---|---|
| 1 | Flow chart presents steps to complete a process | Data flow diagram presents the flow of data |
| 2 | Flow chart does not have any input from or output to external source | Data flow diagram describes the path of data from external source to internal store or vice versa. |
| 3 | The timing and sequence of the process is aptly shown by a flow chart | The processing of data is taking place in a particular order or several processes are taking simultaneously is not described by a data flow diagram. |
| 4 | Flow chart shows how to make a system function. | Data flow diagrams define the functionality of a system |
| 5 | Flow charts are used in designing a process | Data flow diagram are used to describe the path of data that will complete that process. |
| 6 | We have following types of flowcharts,<br>• System flow chart<br>• Data flow chart<br>• Document flow chart<br>• Program flow chart | We have following types of data flow diagrams,<br>• Physical data flow diagrams<br>• Logical data flow diagrams |

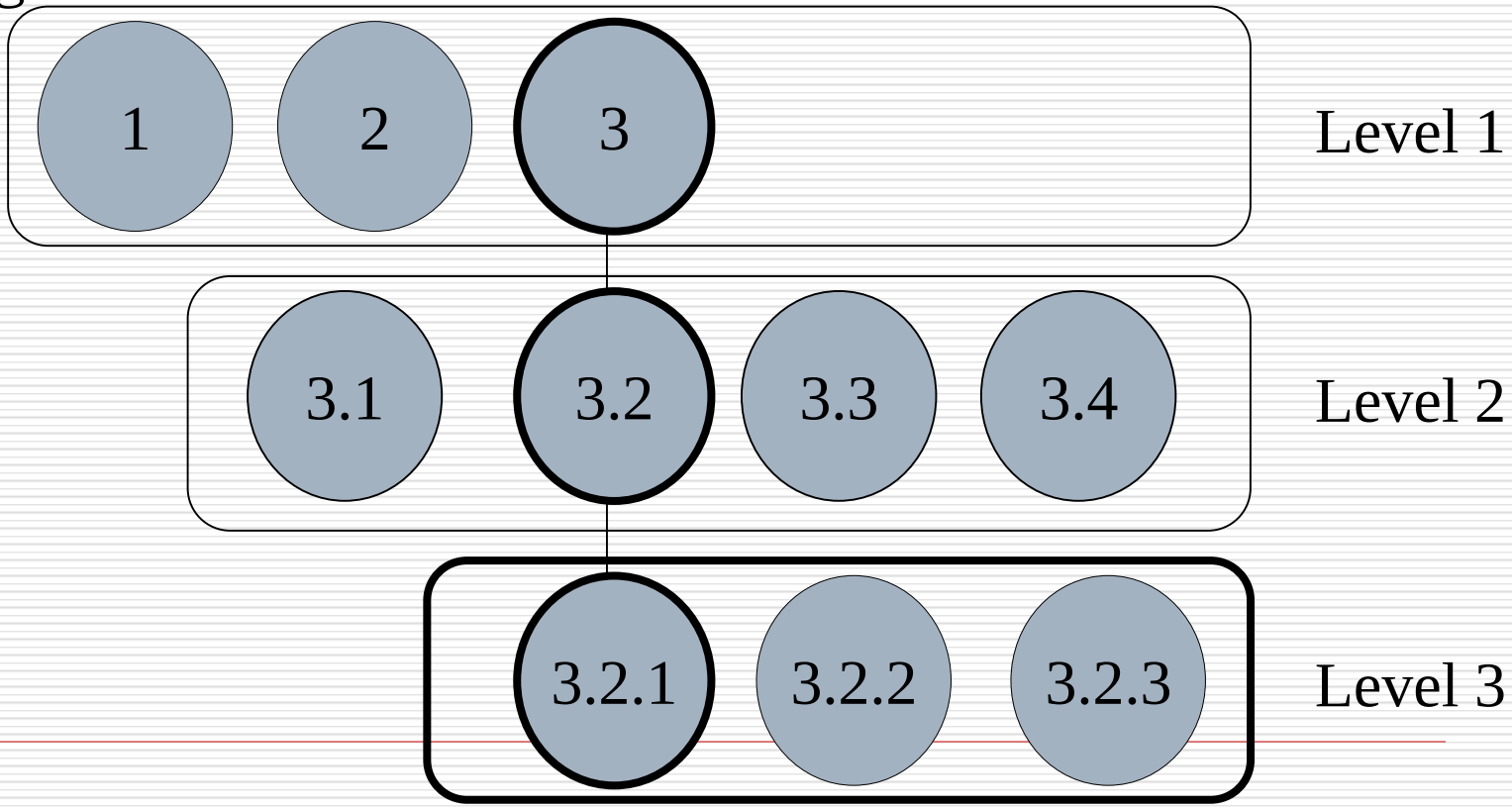
# Rules for Using DFD Symbols

- Data Flow That Connects

| | YES | NO |
|---|---|---|
| **A process to another process** | ✓ | |
| **A process to an external entity** | ✓ | |
| **A process to a data store** | ✓ | |
| **An external entity to another external entity** | | ✓ |
| **An external entity to a data store** | | ✓ |
| **A data store to another data store** | | ✓ |

# DFD Process Numbering Rules

- The process boxes on the level 1 diagram should be numbered arbitrarily, so that no priority is implied. Even where data from one process flows directly into another process, this does not necessarily mean that the first one has to finish before the second one can begin.



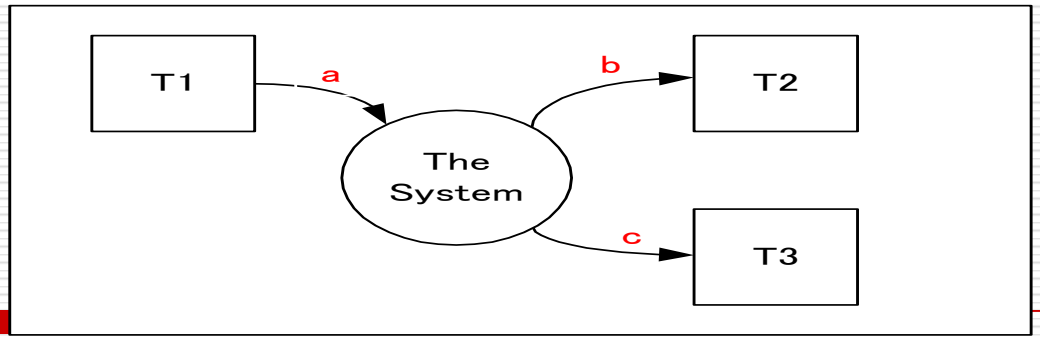



Figure 0 the context diagram

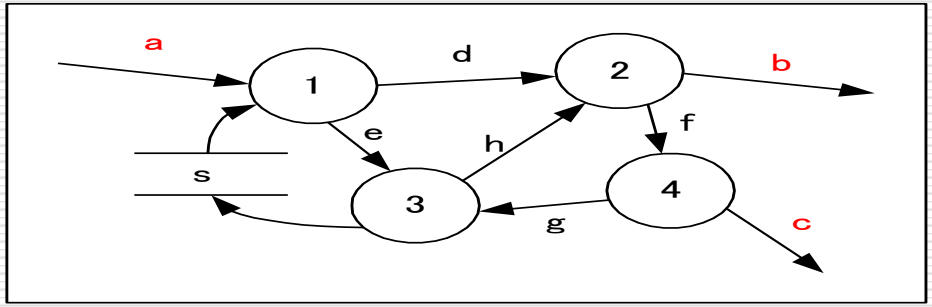



Figure 1 the decomposition of process The System

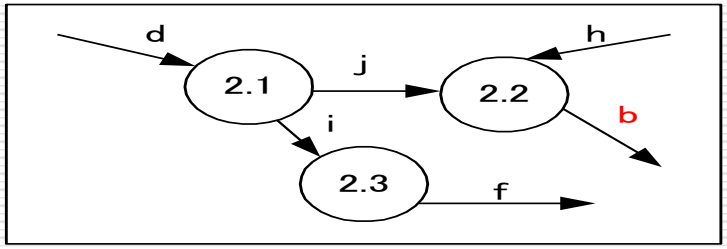



Figure 2

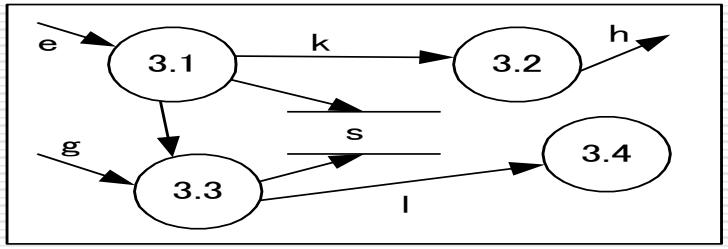



Figure 3

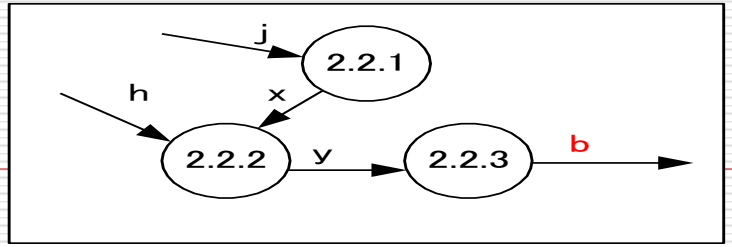
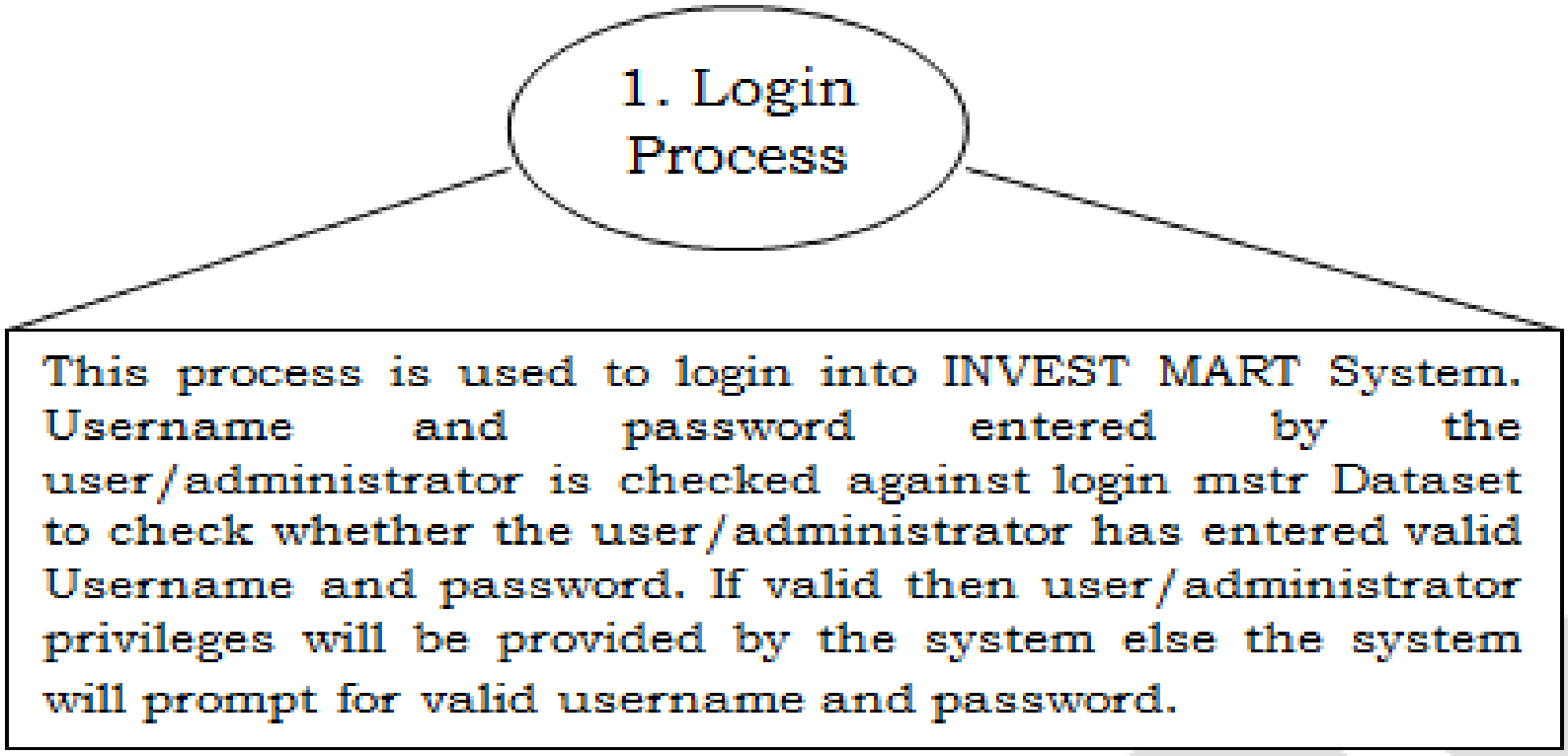



Figure 4

# Process Specifications

- A process specification describes the purpose of processes depicted in the DFD, the inputs to the process, and the output. It describes what the process should do and not how it should do it.

# Example:


1. Login Process
This process is used to login into INVEST MART System. Username and password entered by the user/administrator is checked against login mstr Dataset to check whether the user/administrator has entered valid Username and password. If valid then user/administrator privileges will be provided by the system else the system will prompt for valid username and password.

# Data Dictionary

- A data dictionary is a collection of data about data. It maintains information about the definition, structure, and use of each data element that an organization uses.

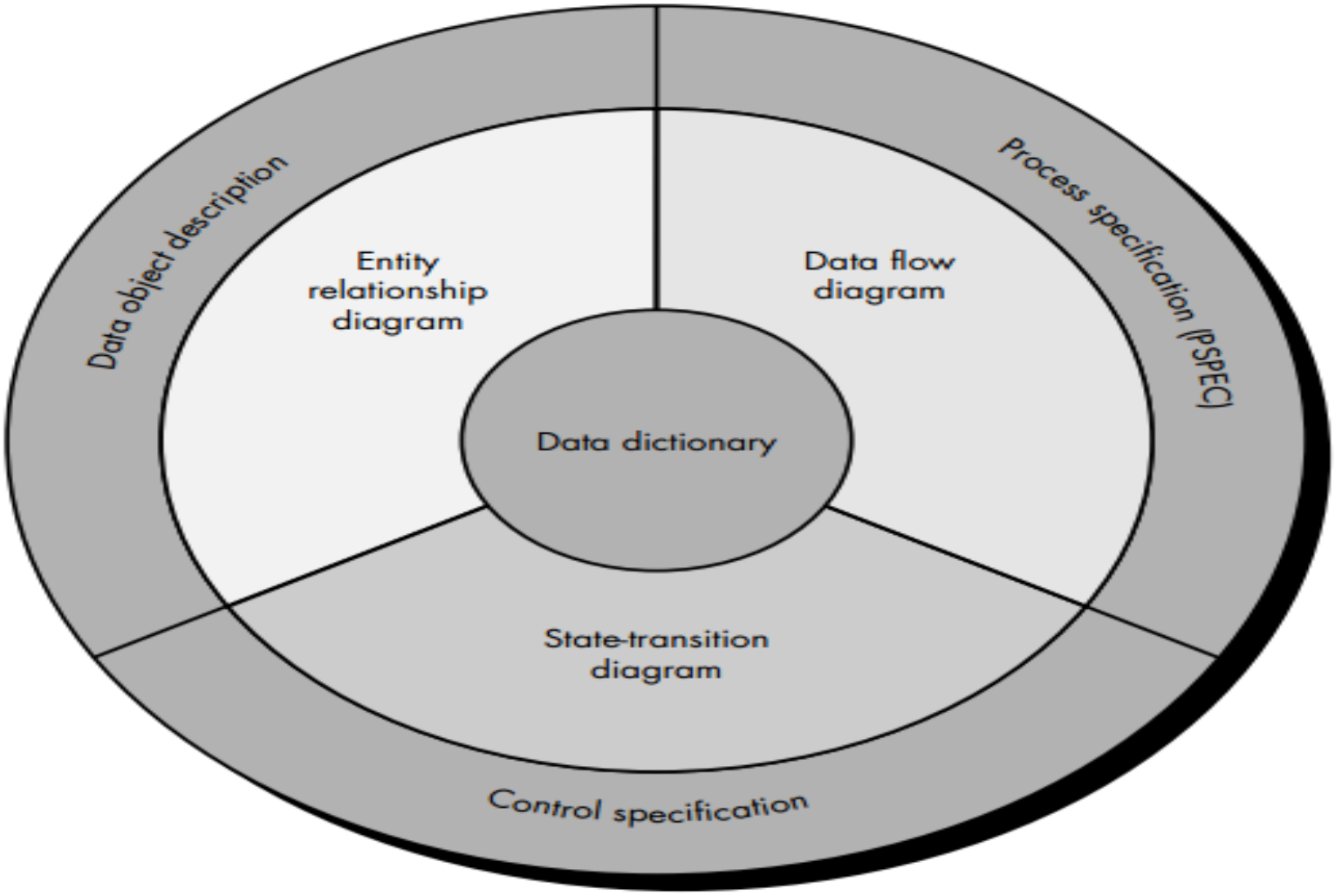
# Basic elements

- Name—the primary name of the data or control item, the data store or an external entity.
- Alias—other names used for the first entry.
- Where-used/how-used—a listing of the processes that use the data or control item and how it is used (e.g., input to the process, output from the process, as a store, as an external entity.
- Description—other information about data types, preset values (if known), restrictions or limitations, and so on.

# Example:

| | |
|---|---|
| Name | Login |
| Alias | Login mstr |
| Where/How to use | Data store, used to store the login details of the administrator of the our application |
| Description | Username + Password |

# Elements of Analysis model



# Data Dictionary

- A data dictionary is a **collection of data about data**. It maintains information about the definition, structure, and use of each data element that an organization uses.

# Data Flow Diagrams

- Through a **structured analysis** technique called data flow diagrams (DFD), the systems analyst can put together a graphical representation of data processes throughout the system.
- The data flow approach emphasizes the logic underlying the system.

# E- R Diagrams

- An **entity relationship** model, also called an **entity-relationship** (**ER**) **diagram**, is a graphical representation of entities and their relationships to each other, typically used in computing in regard to the organization of data within databases or information systems.

# State Transition Diagram

- One way to characterize change in a system is to say that its objects change their **state in** response to events and to time.
- A state diagram shows the various states of a **single object.**
- e.g. When you throw a switch, a light changes its state from **off to on.**

# Process Specifications ( PSPEC )

- A process specification describes the purpose of **processes** depicted in the DFD, the inputs to the process, and the output. It describes what the process should do and not how it should do it.

# Control Specifications ( CSPEC )

- The control specification (CSPEC) represents the **behavior of the system** (at the level from which it has been referenced). The CSPEC contains a state transition diagram that is a sequential specification of behavior.

# Data Object Description

- Data object description is the process of **documenting** a complex software system design as an easily understood diagram, using text and symbols to represent the way data needs to flow.

# Introduction

**System design** is the process of **defining(explain)** the components, modules, interfaces, and data for a **system** to satisfy(convenient) specified requirements.

**System** development is the process of creating or altering **systems**, along with the processes, practices, models, and methodologies(methods) used to develop them.

# Translating of Analysis Model to a Software Design

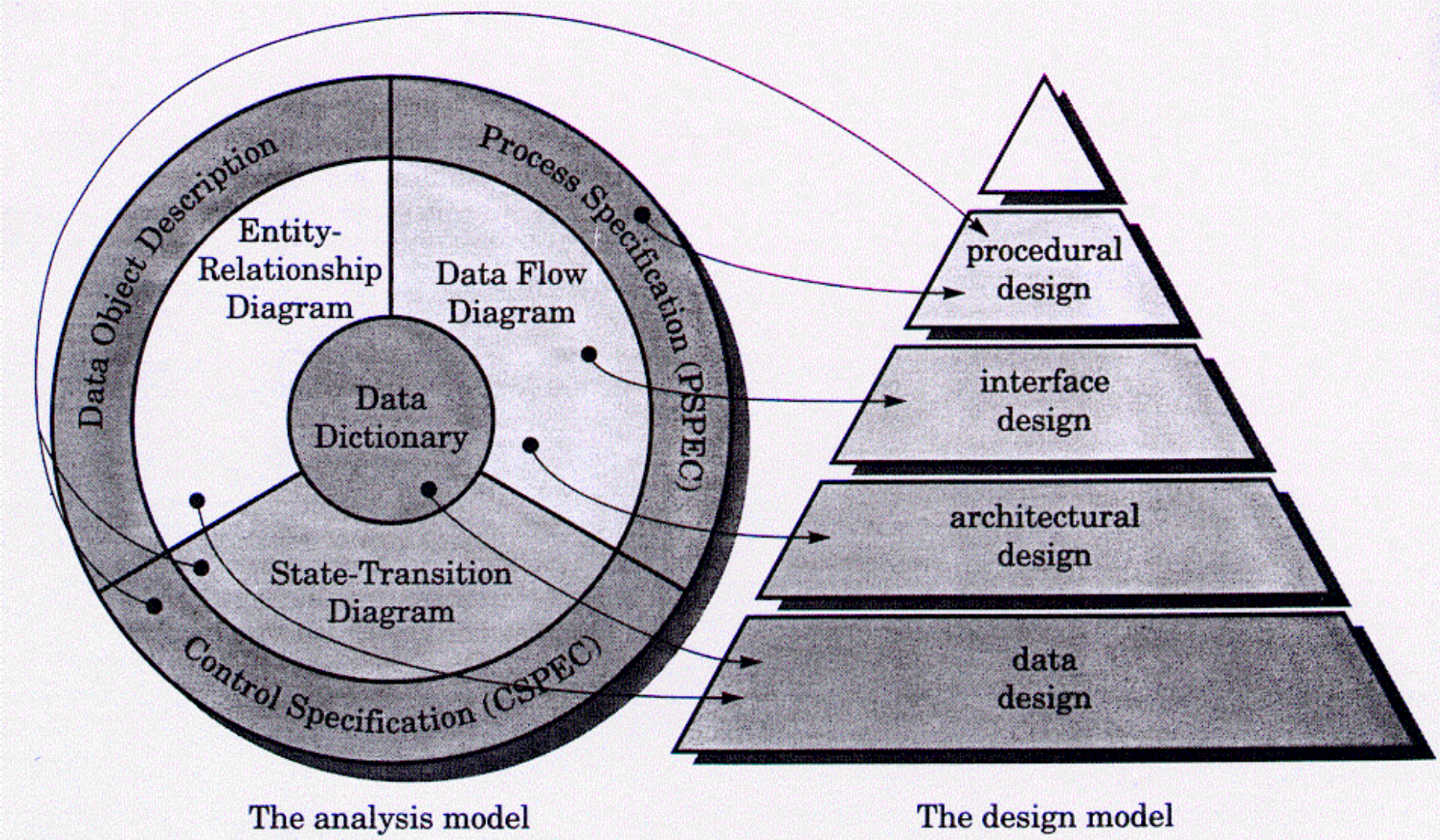



**FIGURE 13.1.** Translating the analysis model into a software design

# 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 program

## Interface Design

- Describes how the software **communicates** with itself, to systems that interact with it and with humans.

## Procedural Design

- Transforms **structural elements** of the architecture into a **procedural description** of software construction

# The Design Process

- Design must **enable all requirements** of the analysis model and implicit needs of the customer to be met
- Design must be **readable** and an **understandable** guide for **coders**, **testers** and **maintainers**
- The design should address the **data**, **functional** and **behavioral domains** of implementation

# Design Guidelines

- A design should exhibit in a **hierarchical organization**
- A design should be **modular**
- A design should contain both **data** and **procedural** abstractions
- Modules should exhibit **independent** functional characteristics
- Interfaces should **reduce complexity**
- A design should be obtained from a **repeatable** method, driven by analysis

# Design Principles

- The Design process should not suffer from "**tunnel Vision**"
- The design should be **traceable** to analysis model
- The design should not **reinvent the wheel** - REUSE
- The design should "**minimize the intellectual distance**" between the software and the problem as it exist in the real world

- The design should exhibit **uniformly** and **integration**.
- The design should be **structured** to accommodate change.
- **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 errors.

# Characteristics of Design

For good quality software to be produced, the software design must also be of good quality. Now, the matter of concern is how the quality of good software design is measured? This is done by observing certain factors in software design. These factors are:

1. Correctness
2. Understandability
3. Efficiency
4. Maintainability

**Correctness :-** A good design should correctly implement all the functionalities identified in the SRS document.

- **Understandability :-** A good design is easily understandable.

- **Efficiency :-** It should be efficient.

- **Maintainability:-** It should be easily amenable to change.

# Design Concepts/Types/Heuristics

- Abstraction
- Refinement
- Modularity
- SW Architecture
- Functional Independence

# Abstraction

- Wasserman: “Abstraction permits one to concentrate on a problem at **some level** of abstraction without regard to **low level details**”
- Essence of abstraction is to deal with the problem at a **higher level ignoring low level** and possibly not so important details concerning the problem at hand.

- Data Abstraction
  - This is a named collection of data that describes a data object
  - e.g. sqrt() function
- Procedural Abstraction
  - Instructions are given in a named sequence
  - Each instruction has a limited function
  - e.g. Send Mail
- Control Abstraction
  - A program control mechanism without specifying internal details, e.g., semaphore.

# Refinement

- Refinement is a process where one or several instructions of the program are **decomposed** into more **detailed** instructions.
- Stepwise refinement is a top down strategy
  - Basic architecture is developed iteratively
  - Step wise hierarchy is developed
- Forces a designer to develop low level details as the design progresses
  - Design decisions at each stage

# Modularity

- In this concept, software is **divided** into separately **named** and addressable **components** called **modules**
- Follows "**divide** and **conquer**(overcome)" concept, a complex **problem** is broken down into several **manageable pieces**
- Let $p_1$ and $p_2$ be two program parts, and E the effort to solve the problem. Then,

  $$\mathbf{E(p_1+p_2) > E(p_1)+E(p_2)}$$

- A need to divide software into optimal sized modules

# Modularity & Software Cost

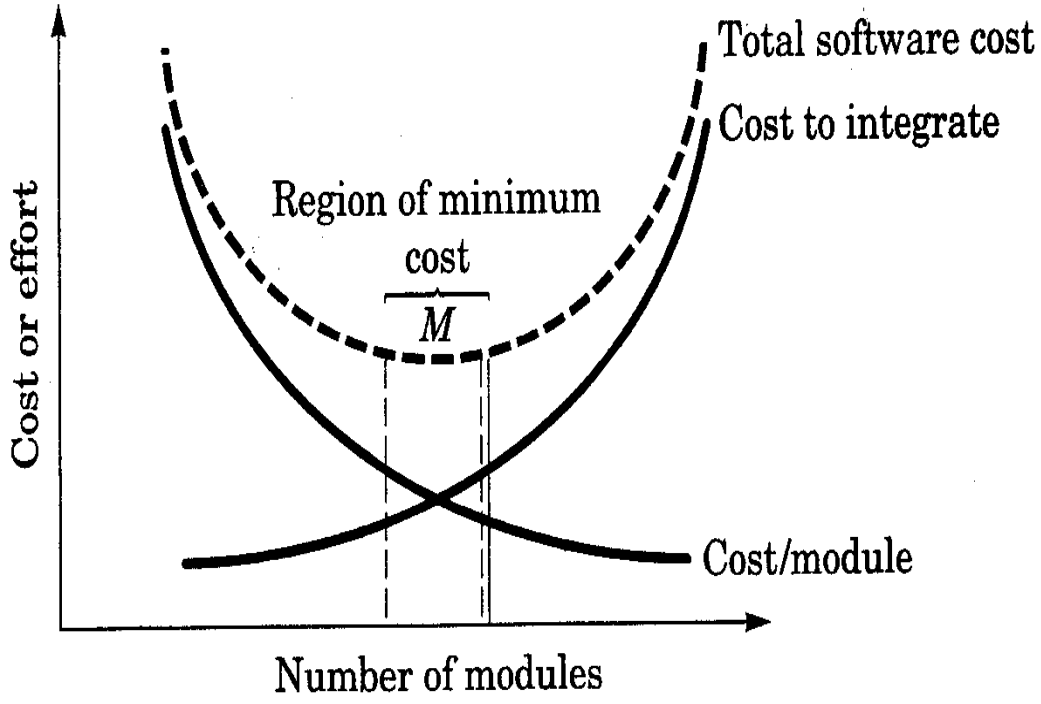
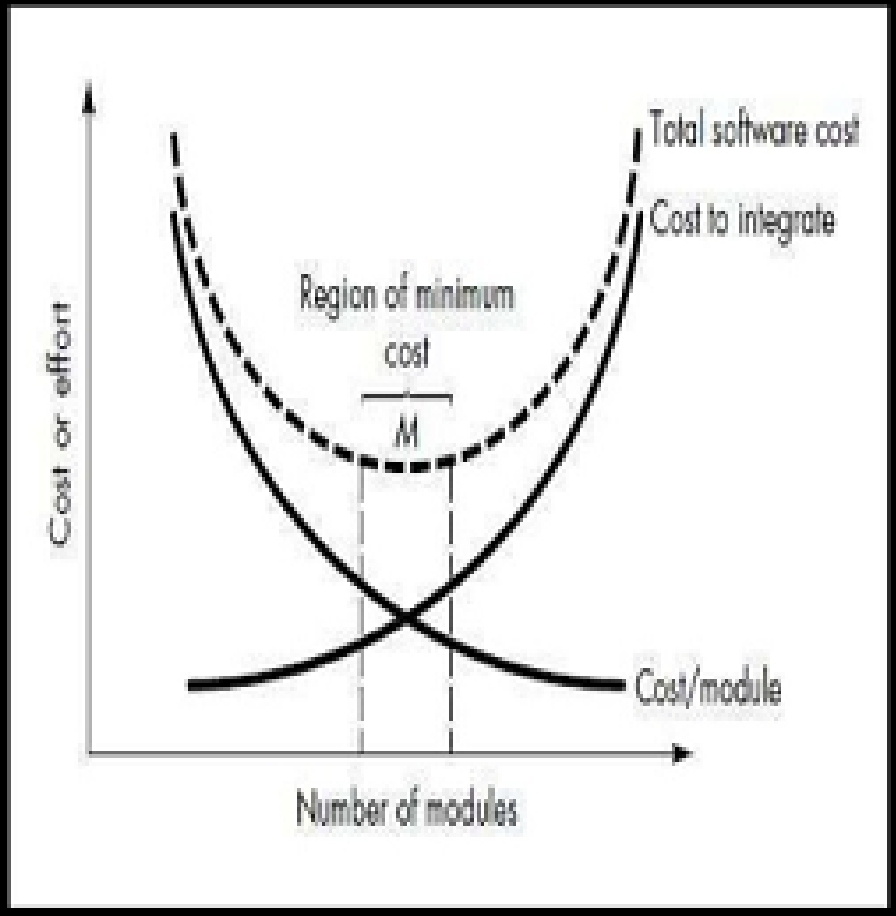



**FIGURE 13.2.**
Modularity and software cost

# Modularity Modules' Cost Verses Size

The effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows.



# Modularity

Objectives of modularity in a design method

- ***Modular Decomposability***
  - Provide a systematic mechanism to **decompose** a problem into sub problems
- ***Modular Composability***
  - Enable **reuse** of existing components
- ***Modular Understandability***
  - Can the module be understood as a **stand alone** unit? Then it is easier to understand and change.

# Modularity

- ***Modular Continuity***
  - If small changes to the system requirements result in changes to **individual** modules, rather than **system-wide** changes, the impact of the **side effects** is reduced

***Modular Protection***

- If there is an **error** in the module, then those **errors** are **localized** and not spread to other modules

# Software Architecture

Desired properties of an architectural design

- Structural Properties
  - This defines the components of a system and the manner in **which** these **interact** with one **another**.
- Extra Functional Properties
  - This addresses **how** the design architecture achieves requirements for **performance**, reliability and security
- Families of Related Systems
  - The ability to **reuse** architectural building blocks

# Structural Diagrams

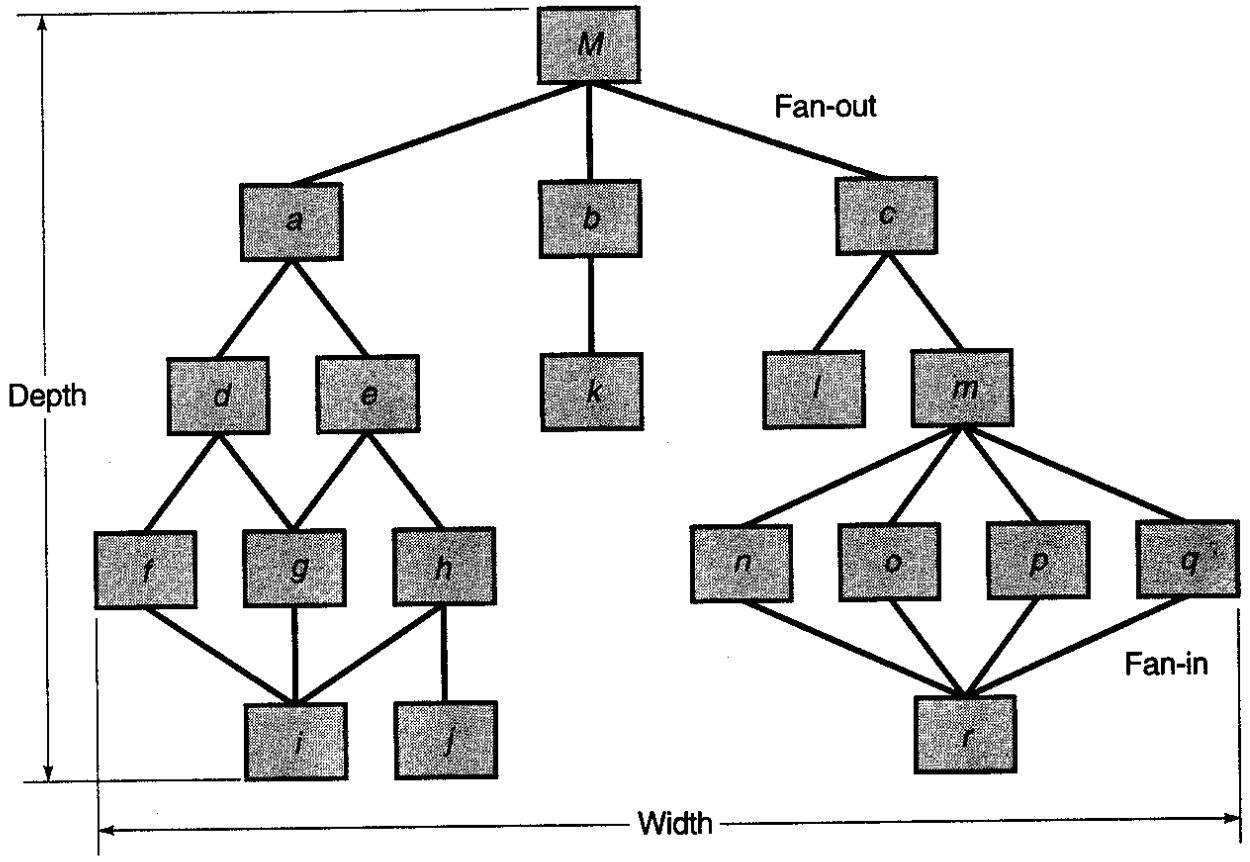



**FIGURE 13.3.**
Structure terminology

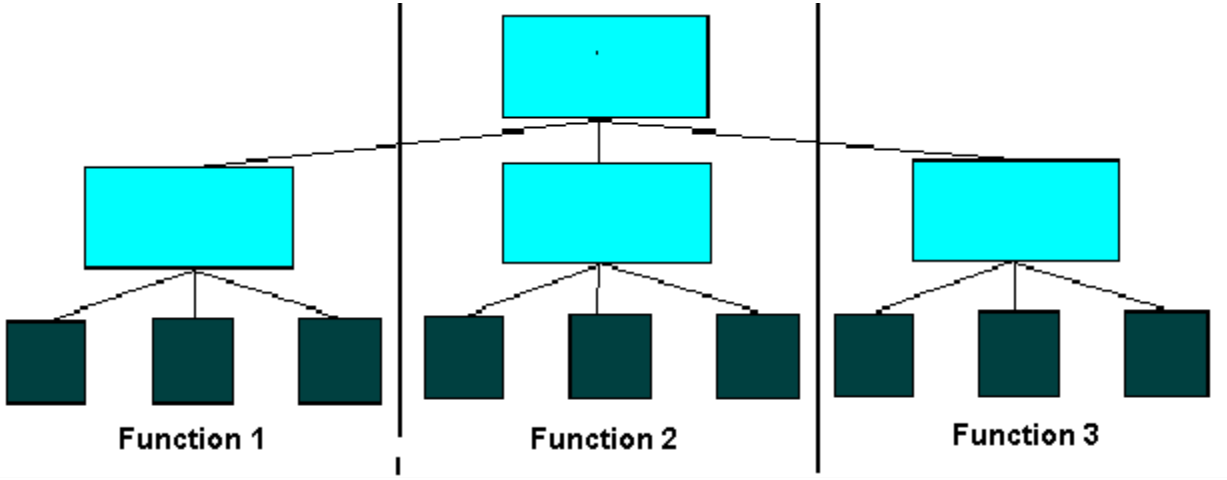
# FAN-IN & FAN-OUT

- **Fan-in** is a measure of the number of functions or methods that call some other function or method (say X). **Fan-out** is the number of functions that are called by function X.

# Program Structure Partitioning

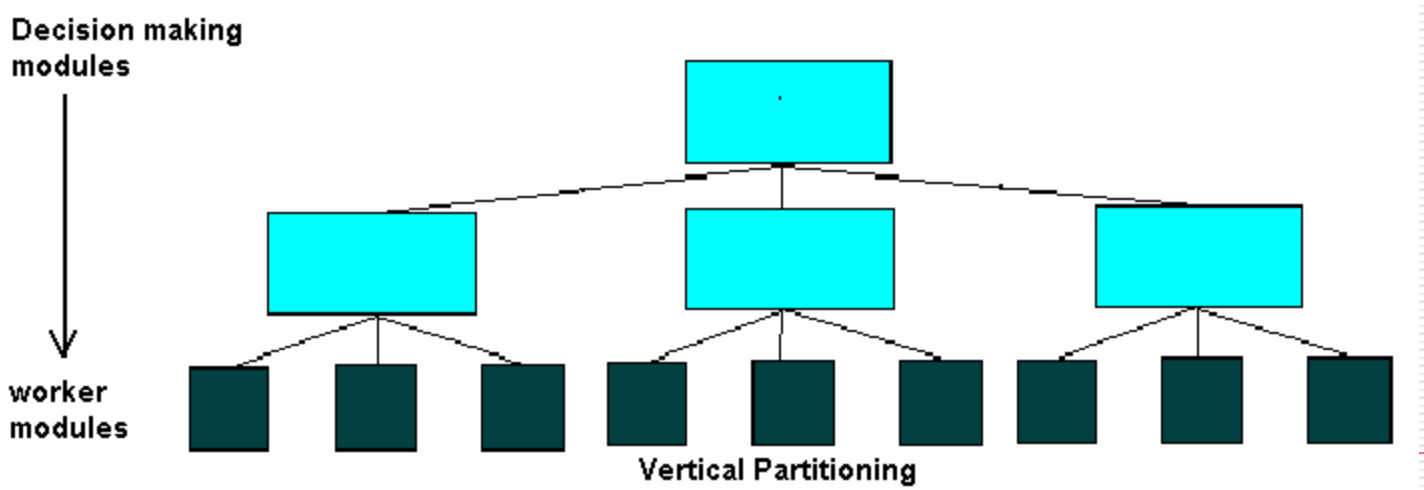
- Horizontal Partitioning
  - Easier to test
  - Easier to maintain
  - Propagation of fewer side effects
  - Easier to add new features

F1 (Ex: Input) F2 (Process) F3(Output)



# Program Structure Partitioning

- Vertical Partitioning
  - Control and work modules are distributed top down
  - Top level modules perform control functions
  - Lower modules perform computations
    - Less susceptible(capable) to side effects
    - Also very maintainable



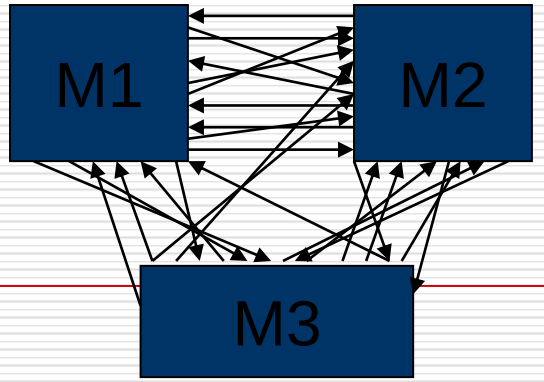
# Functional Independence OR Effectiveness of Modular Design

- Designing modules in such a way that **each module** has a **specific functional requirements**.
- Critical in dividing system into **independently** implementable parts
- Measured by two qualitative criteria
  - **Cohesion**
    - Relative functional strength of a module
  - **Coupling**
    - Relative interdependence among modules

- High cohesion
- Low coupling

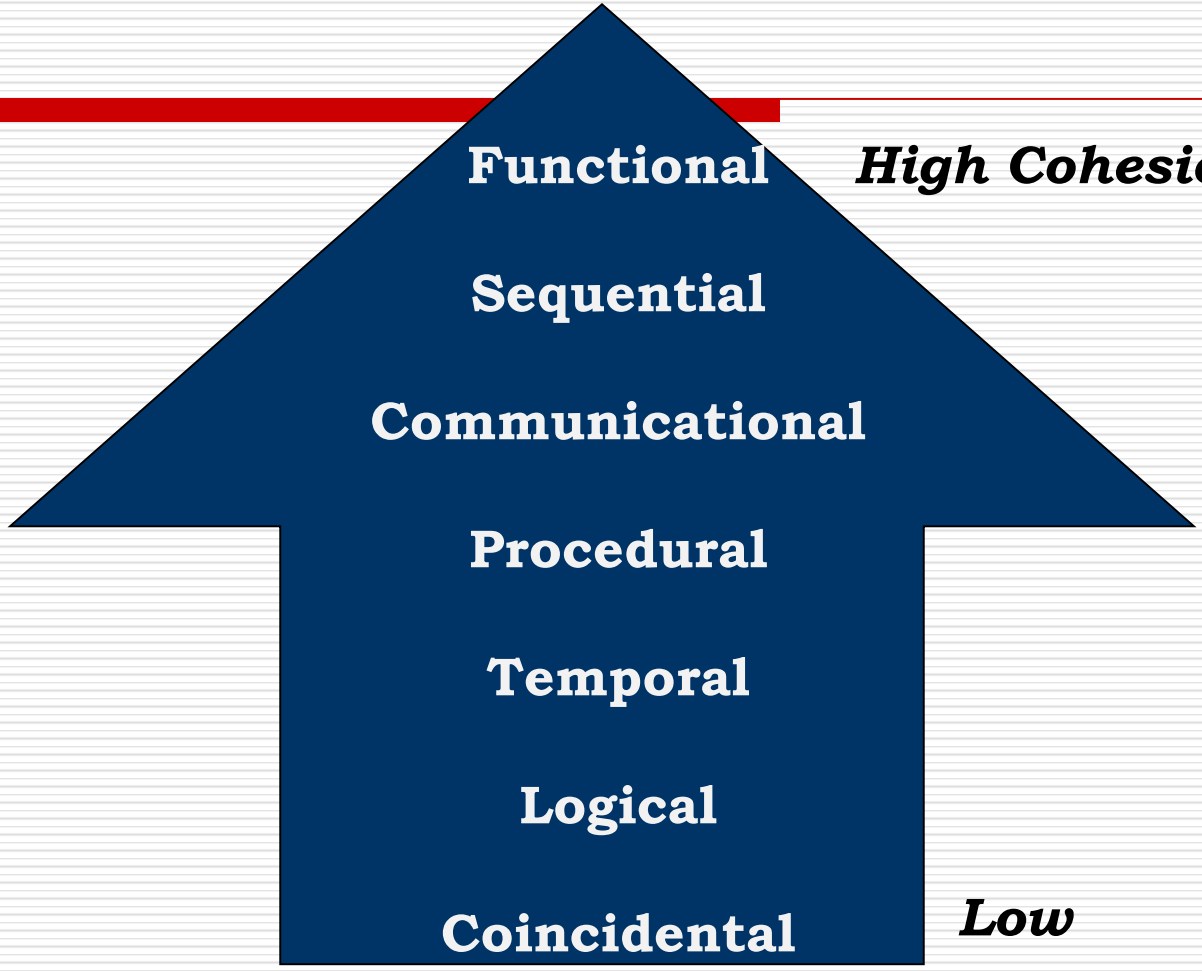
| Cohesion | Coupling |
| --- | --- |
| **Cohesion** is the indication of the relationship within **module**. | **Coupling** is the indication of the relationships between modules. |
| Cohesion shows the module's relative **functional** strength. | Coupling shows the relative **independence** among the modules. |
| Cohesion is a degree (quality) to which a component / module focuses on the **single** thing. | Coupling is a degree to which a component / module is connected to the **other** modules. |
| While designing you should strive(effort) for **high cohesion** i.e. a cohesive component/ module focus on a single task (i.e., **single-mindedness**) with little interaction with other modules of the system. | While designing you should strive for **low coupling** i.e. **dependency** between modules should be less. |
| Cohesion is **Intra – Module** Concept (Between units within a **module**) | Coupling is **Inter -Module** Concept. (between or among groups of **modules** ) |

# Cohesion

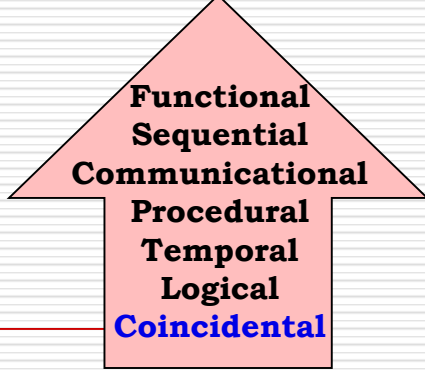



- Definition
  - The degree(quality) to which **all elements** of a **component(objects)** are directed towards a **single task**.
  - The degree to which all **elements(**what gets returned from **components)** directed towards a task are contained in a **single component**.
  - The degree to which all responsibilities of a **single class are related**.
- Internal glue with which component is constructed
- A cohesive module performs a **single task**

# Types of Cohesion



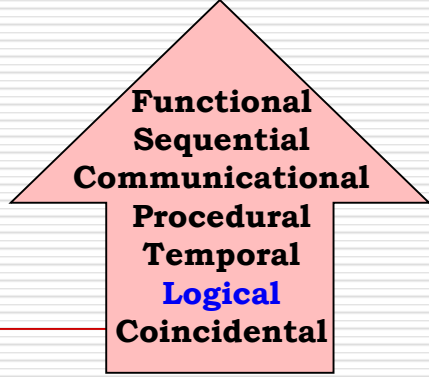
# Coincidental Cohesion




- Def: Parts of the component are unrelated (unrelated functions, processes, or data)
- Parts of the component are only related by their location in source code.
- Elements needed to achieve some functionality are scattered throughout the system.
- Accidental
- Worst form

# Example

1. Print next line
2. Reverse string of characters in second argument
3. Add 7 to 5$^{th}$ argument
4. Convert 4$^{th}$ argument to float

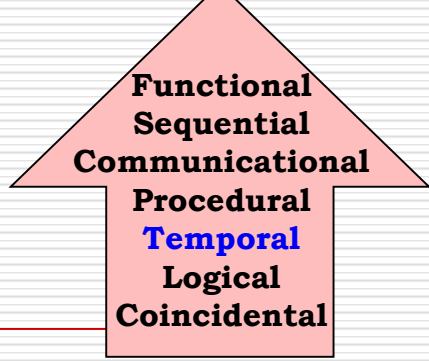



# Logical Cohesion

- Def: Elements of component are related logically and not functionally.
- Several logically related elements are in the same component and one of the elements is selected by the client component.

# Example

- A component reads inputs from tape, disk, and network.
- All the code for these functions are in the same component.
- Operations are related, but the functions are significantly different.

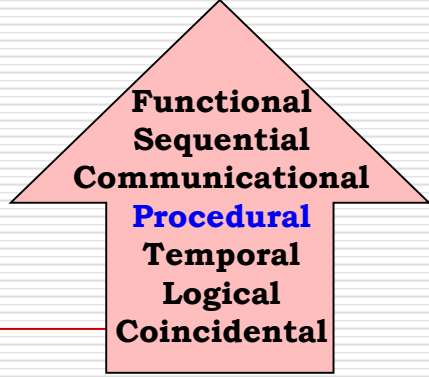



# Temporal Cohesion

- Def: Elements are related by timing involved
- Elements are grouped by when they are processed.
- Example: An exception handler that
  - Closes all open files
  - Creates an error log
  - Notifies user
  - Lots of different activities occur, all at same time

# Example

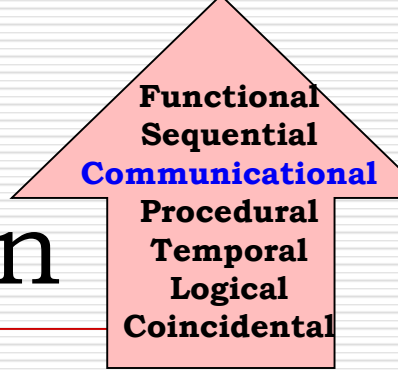
- A system initialization routine: this routine contains all of the code for initializing all of the parts of the system. Lots of different activities occur, all at init time.




# Procedural Cohesion

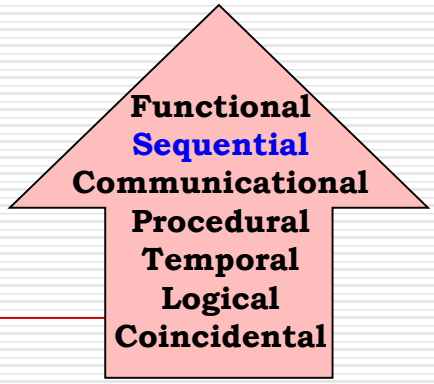
- Def: Elements of a component are related only to ensure a particular order of execution.
- Actions are still weakly connected and unlikely to be reusable.
- Example:
  - …
  - Write output record
  - Read new input record
  - Pad input with spaces
  - Return new record
  - …

# Communicational Cohesion




- Def: Functions performed on the same data or to produce the same data.
- Examples:
  - Update record in data base and send it to the printer
    - Update a record on a database
    - Print the record
  - Fetch unrelated data at the same time.
    - To minimize disk access

# Sequential Cohesion

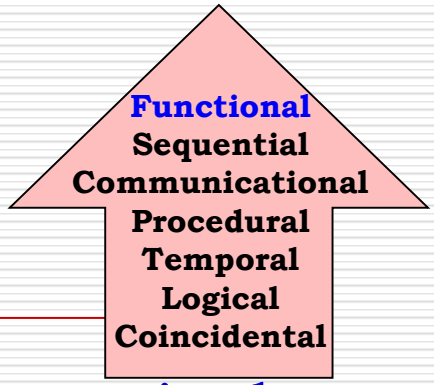



- Def: The output of one part is the input to another.
- *Data flows* between parts (different from procedural cohesion)
- Occurs naturally in functional programming languages
- Good situation

# Example

- Calculate Taxes and Transfer Amount

# Functional Cohesion

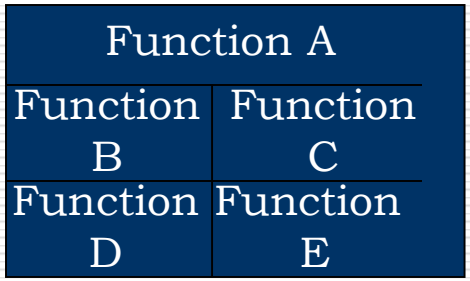



- Def: Every essential element to a single computation is contained in the component.
- Every element in the component is essential to the computation.
- Ideal situation
- What is a functionally cohesive component?
  - One that not only performs the task for which it was designed but
  - it performs only that function and nothing else.

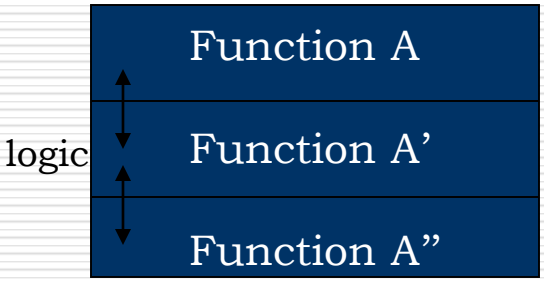
# Example

- Calculate Sales Commission
- Read Data from the file

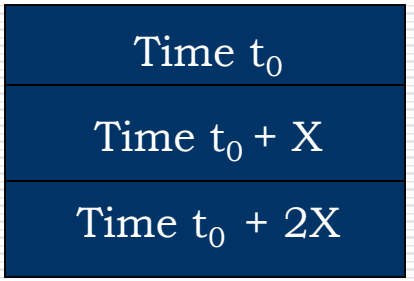
# Examples of Cohesion



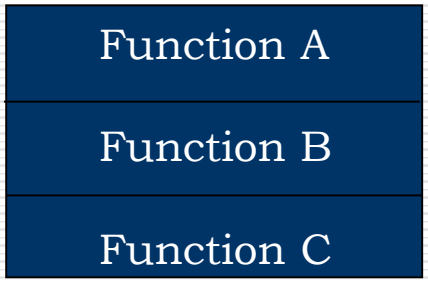

*Coincidental*
Parts unrelated




*Logical*
Similar functions

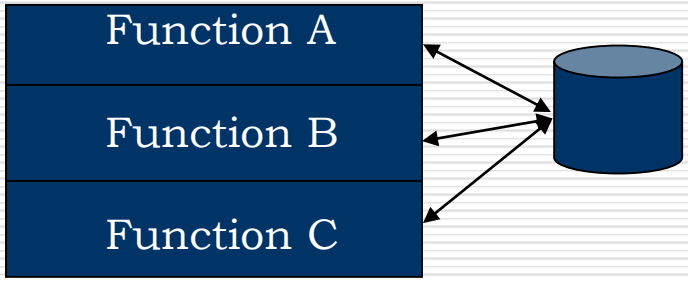



*Temporal*
Related by time

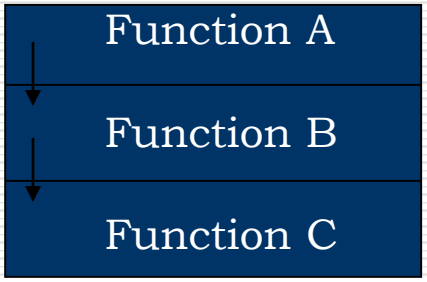



*Procedural*
Related by order of functions

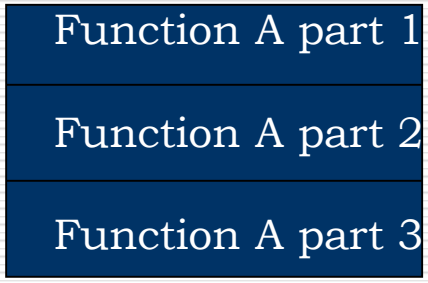
# Examples of Cohesion (Cont.)




*Communicational*
Access same data



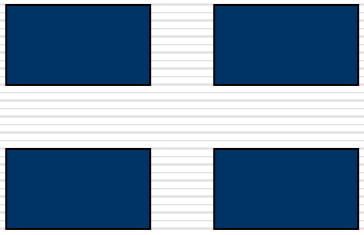

*Sequential*
Output of one is input to another



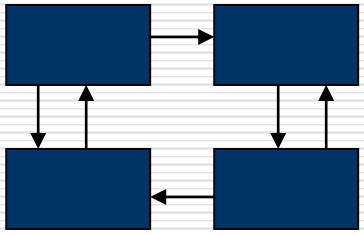

*Functional*
Sequential with complete, related functions

# Coupling (Modules)

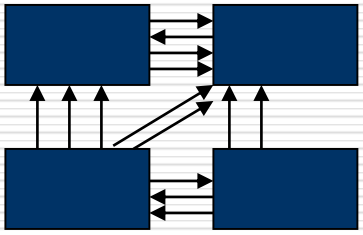
- The degree of dependence such as the amount of interactions among components



No dependencies



Loosely coupled
some dependencies

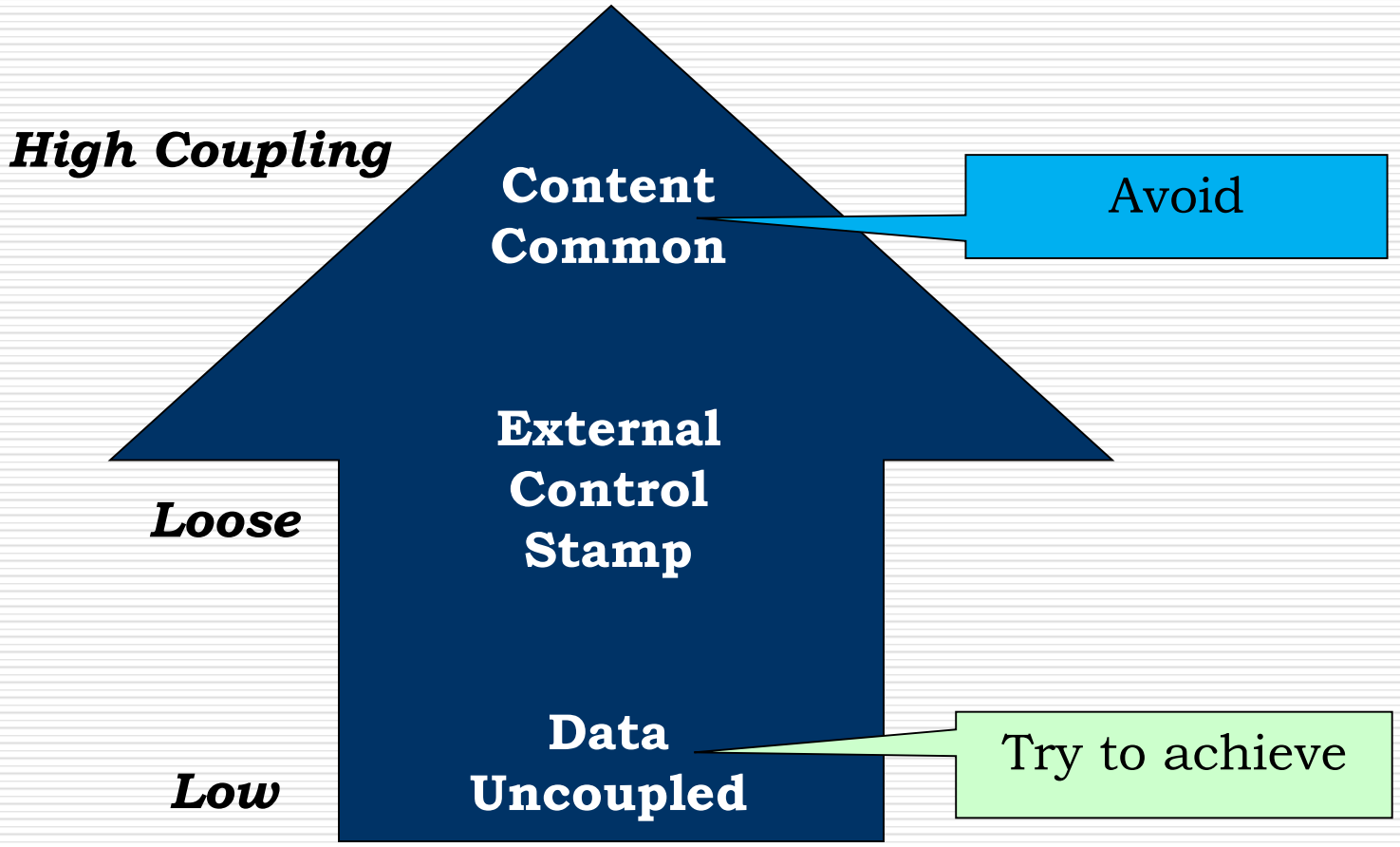


Highly coupled
many dependencies

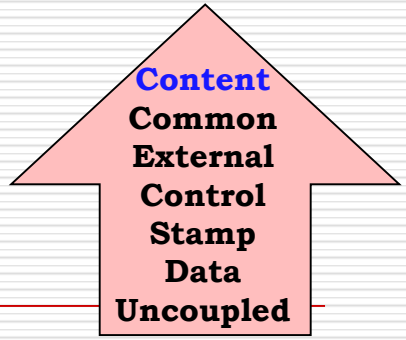
# Coupling

- The degree of dependence such as the amount of interactions among components

# Type of Coupling



# Content Coupling



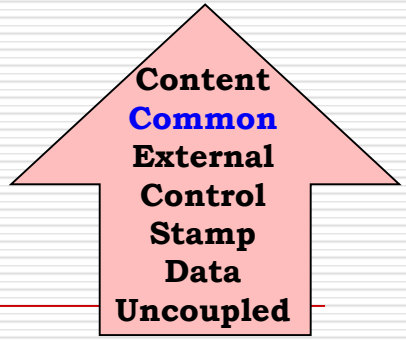

- Def: One component modifies another.
- Example:
  - Component directly modifies another's data
  - Component modifies another's code, e.g., jumps (goto) into the middle of a routine

# Example

Part of a program handles lookup for customer.

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

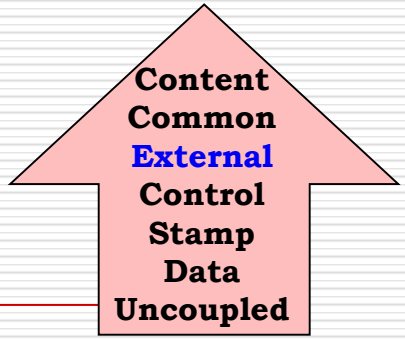
# Common Coupling




- Def: More than one component share data such as global data structures
- Usually a poor design choice because
  - Lack of clear responsibility for the data
  - Reduces readability
  - Difficult to determine all the components that affect a data element (reduces maintainability)
  - Difficult to reuse components
  - Reduces ability to control data accesses

# Example

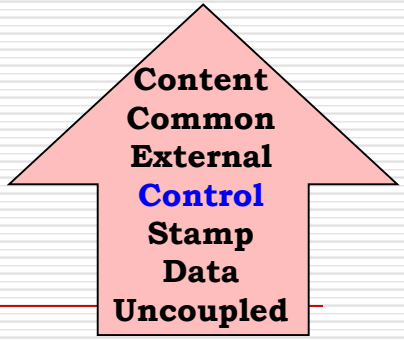
Process control component maintains current data about state of operation. Gets data from multiple sources. Supplies data to multiple sinks. Each source process writes directly to global data store. Each sink process reads directly from global data store.




# External Coupling

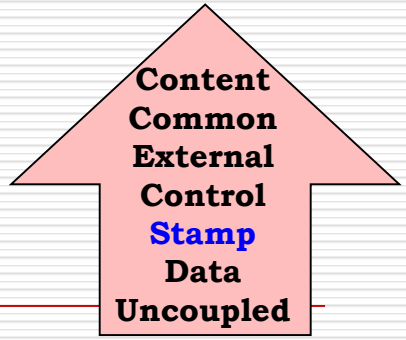
- Def: Two components share something externally imposed, e.g.,
  - External file
  - Device interface
  - Protocol
  - Data format

# Control Coupling




- Def: Component passes control parameters to coupled components.
- May be either good or bad, depending on situation.
  - Bad if parameters indicate completely different behavior
  - Good if parameters allow factoring and reuse of functionality
- Good example: sort that takes a comparison function as an argument.
  - The sort function is clearly defined: return a list in sorted order, where sorted is determined by a parameter.

# Stamp Coupling

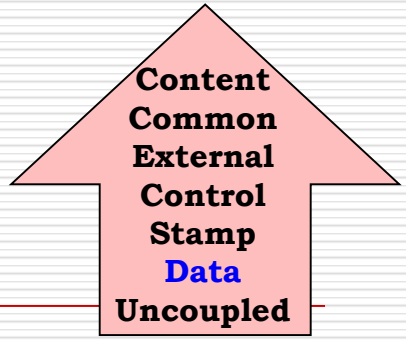



- Def: Component passes a data structure to another component that does not have access to the entire structure.
- Requires second component to know how to manipulate the data structure (e.g., needs to know about implementation).
- The second has access to more information that it needs.
- May be necessary due to efficiency factors: this is a choice made by insightful designer, not lazy programmer.

# Example

## Customer Billing System

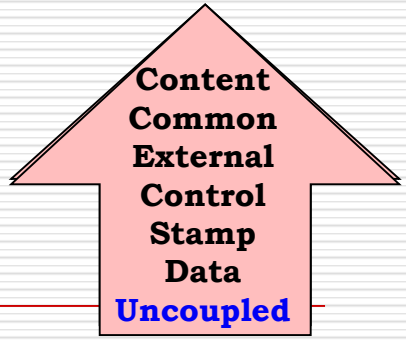
The print routine of the customer billing accepts customer data structure as an argument, parses it, and prints the name, address, and billing information.




# Data Coupling

- Def: Component passes data (not data structures) to another component.
- Every argument is simple argument or data structure in which all elements are used
- Good, if it can be achieved.
- Example: Customer billing system
  - The print routine takes the customer name, address, and billing information as arguments.

# Uncoupled




- Completely uncoupled components are not systems.
- Systems are made of interacting components.

# Modular Design -- Coupling

- Coupling describes the interconnection among modules
- Data coupling
  - Occurs when one module passes local data values to another as parameters
- Stamp coupling
  - Occurs when part of a data structure is passed to another module as a parameter

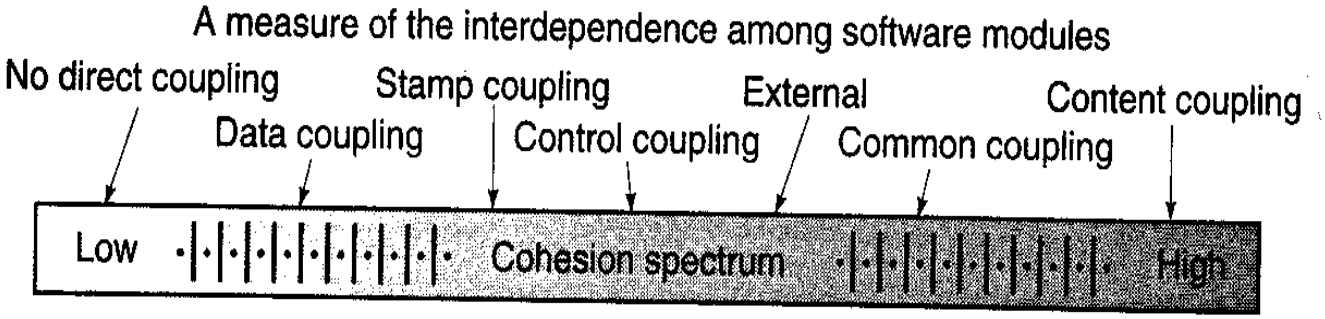



**FIGURE 13.7.**
Coupling

# Modular Design -- Coupling

- Control Coupling
  - Occurs when control parameters are passed between modules
- Common Coupling
  - Occurs when multiple modules access common data areas such as Fortran Common or C extern
- Content Coupling
  - Occurs when a module data in another module
- Subclass Coupling
  - The coupling that a class has with its parent class

# Examples of Coupling

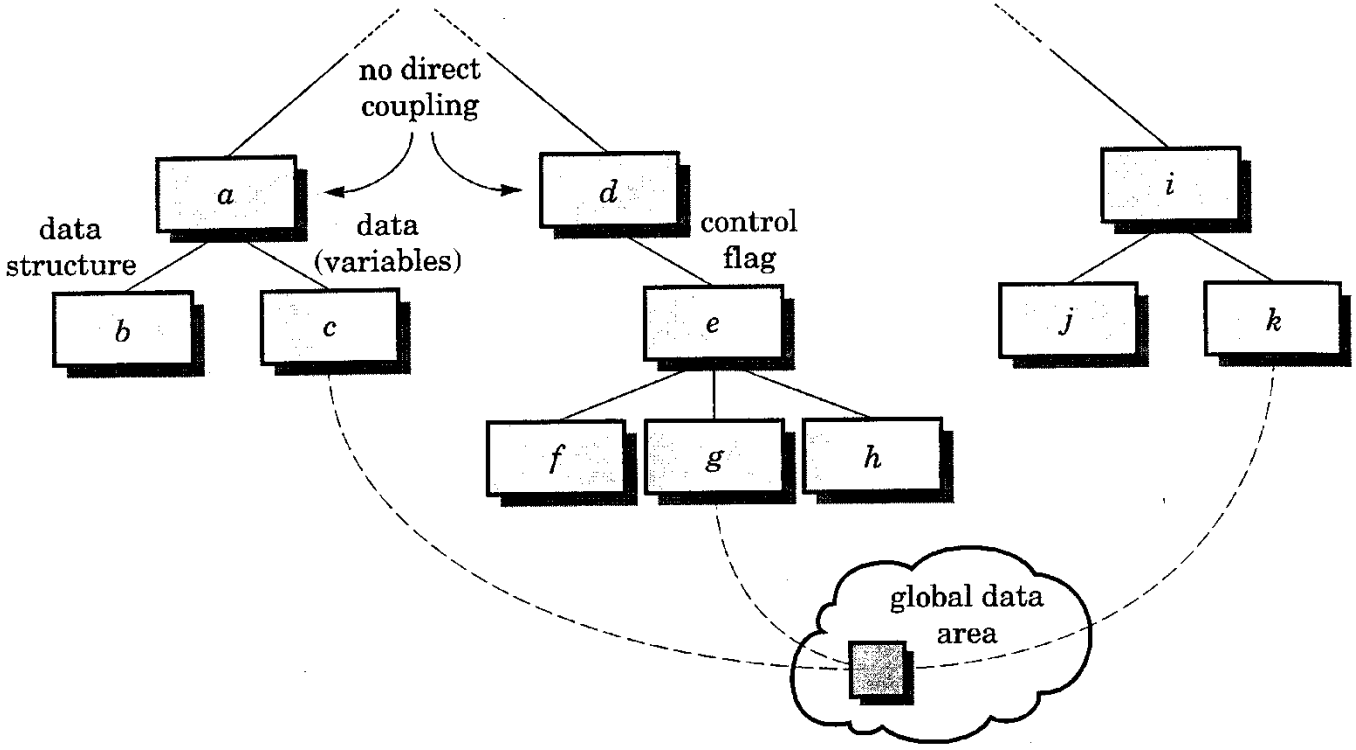



**FIGURE 13.8.** Types of coupling

# Translating of Analysis Model to a Software Design

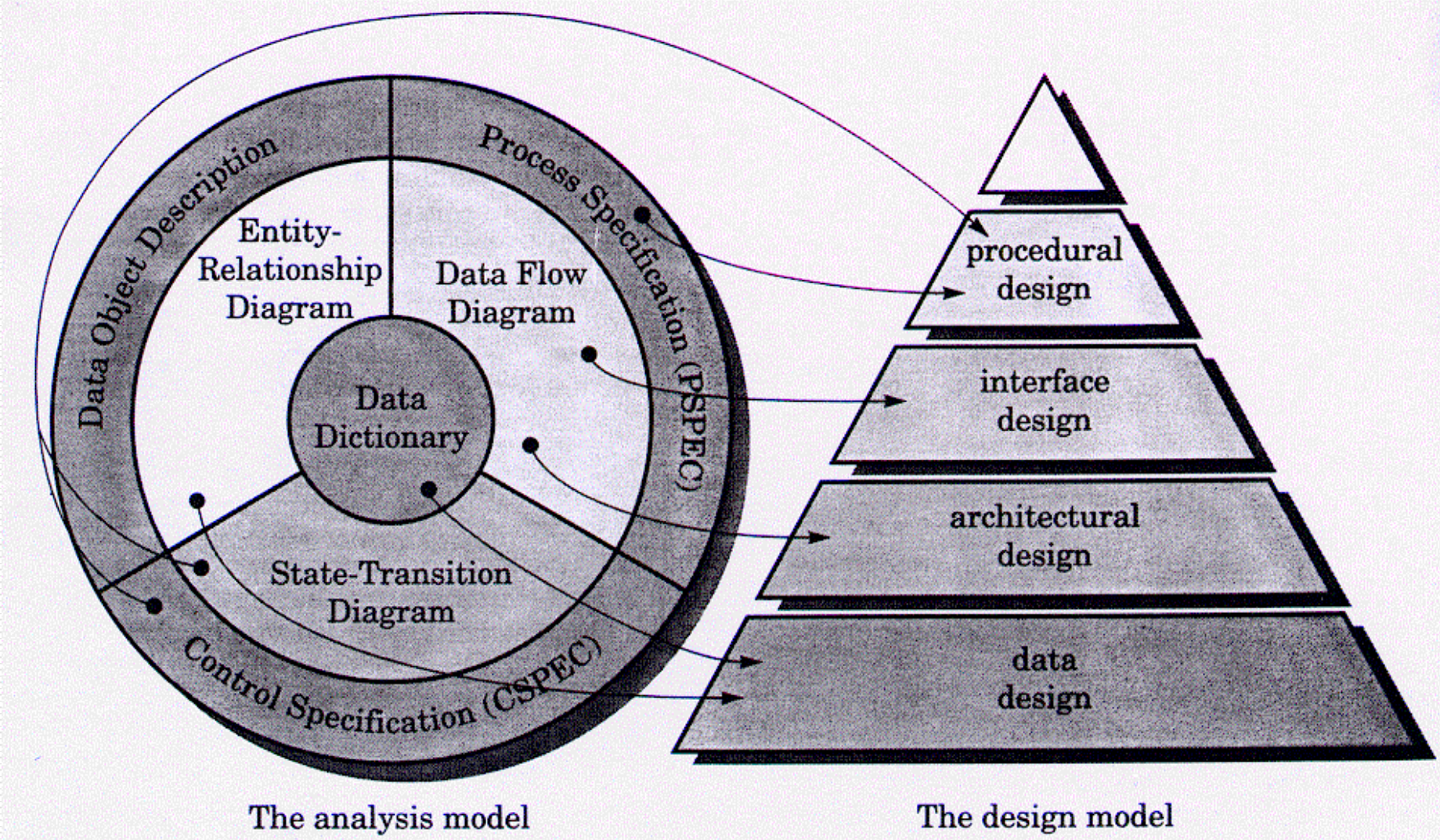



**FIGURE 13.1.** Translating the analysis model into a software design

# Documentation

I. Scope
  A. System objectives
  B. Major software requirements
  C. Design constraints, limitations
II. Data Design
  A. Data objects and resultant data structures
  B. File and database structures
    1. external file structure
      a. logical structure
      b. logical record description
      c. access method
    2. global data
    3. file and data cross reference
III. Architectural Design
  A. Review of data and control flow
  B. Derived program structure
IV. Interface Design
  A. Human–machine interface specification
  B. Human–machine interface design rules
  C. External interface design
    1. Interfaces to external data
    2. Interfaces to external systems or devices
  D. Internal interface design rules
V. Procedural Design
  *For each module:*
  A. Processing narrative
  B. Interface description
  C. Design language (or other) description
  D. Modules used
  E. Internal data structures
  F. Comments/restrictions/limitations
VI. Requirements Cross-Reference
VII. Test Provisions
    1. Test guidelines
    2. Integration strategy
    3. Special considerations
VIII. Special Notes
IX. Appendices

**FIGURE 13.10.**
Design specification outline