

# Unit 3

System Models, Design Engineering, Component level design

*Software Engineering: A Practitioner's Approach, 7/e*

by Roger S. Pressman

**By**  
**G.Kalpana**  
**Asst.professor**

# Syllabus

- **System models:**

Context Models, Behavioral models, Data models, Object models, structured methods.

- **Design Engineering:**

Design process and Design quality, Design concepts, the design model, Modeling

- **component level design:**

design class based components, conducting component level design.

- **User interface design:** Golden rules.

# System Models

## Syllabus:

- Context models
- Behavioural models
- Data models
- Object models
- CASE workbenches

# System models

- System models may be developed during the **Requirements Engineering** process(RE)
- **Abstract descriptions** of systems whose requirements are being analysed
- System models are **graphical representations** that describe business process.

# Objectives

- To Understand the **boundaries of the system** and model its context
- To **describe** behavioural modelling, data modelling and object modelling
- To introduce some of the **notations used** in the Unified Modeling Language (UML)
- To show how CASE workbenches support system modelling
- To document **system specification** as a set of system models

# System modelling

- System modelling helps the analyst to understand the **functionality of the system and models** are used to communicate with customers
- System models are an important bridge between the **analysis and design processes**
- Different models present the system from **different perspectives**
  - **External perspective** showing the system's context or environment
  - **Behavioural perspective** showing the behaviour of the system
  - **Structural perspective** showing the architecture of the system

# Types of System Models

- Different types of system models are based on different types of **abstraction**
- 1)Data flow model
  - how the data is processed at different stages in the system
- 2)a composition model
  - how entities in the system are composed of their entities
- 3)Architectural Model
  - principal subsystem that makeup a system
- 4)Classification model
  - How entities have common characteristics
- 5)Stimulus response model / state transition diagrams
  - how the system reacts to internal and external events

# Structured methods

- Structured methods incorporate **system modelling** as an inherent part of the method
- Methods **define a set of models**, a process for deriving these models and rules and guidelines that should apply to the models
- **CASE tools** support system modelling as part of a structured method



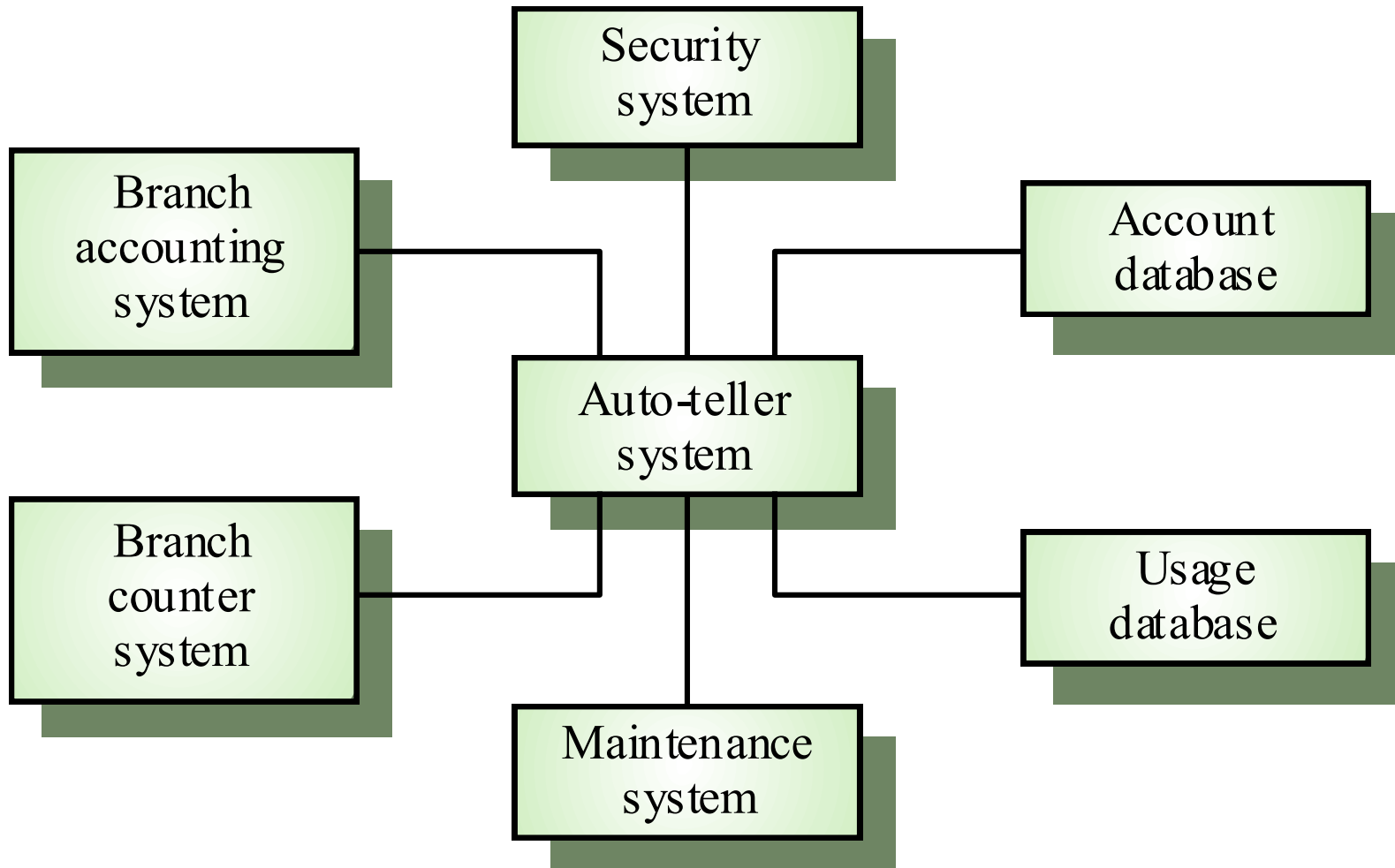
# Method weaknesses

- They do not model **non-functional** system requirements
- They do not usually include information about whether a method is appropriate for a given problem
- They may **produce too much** documentation
- The system models are sometimes **too detailed and difficult** for users to understand

# Context models

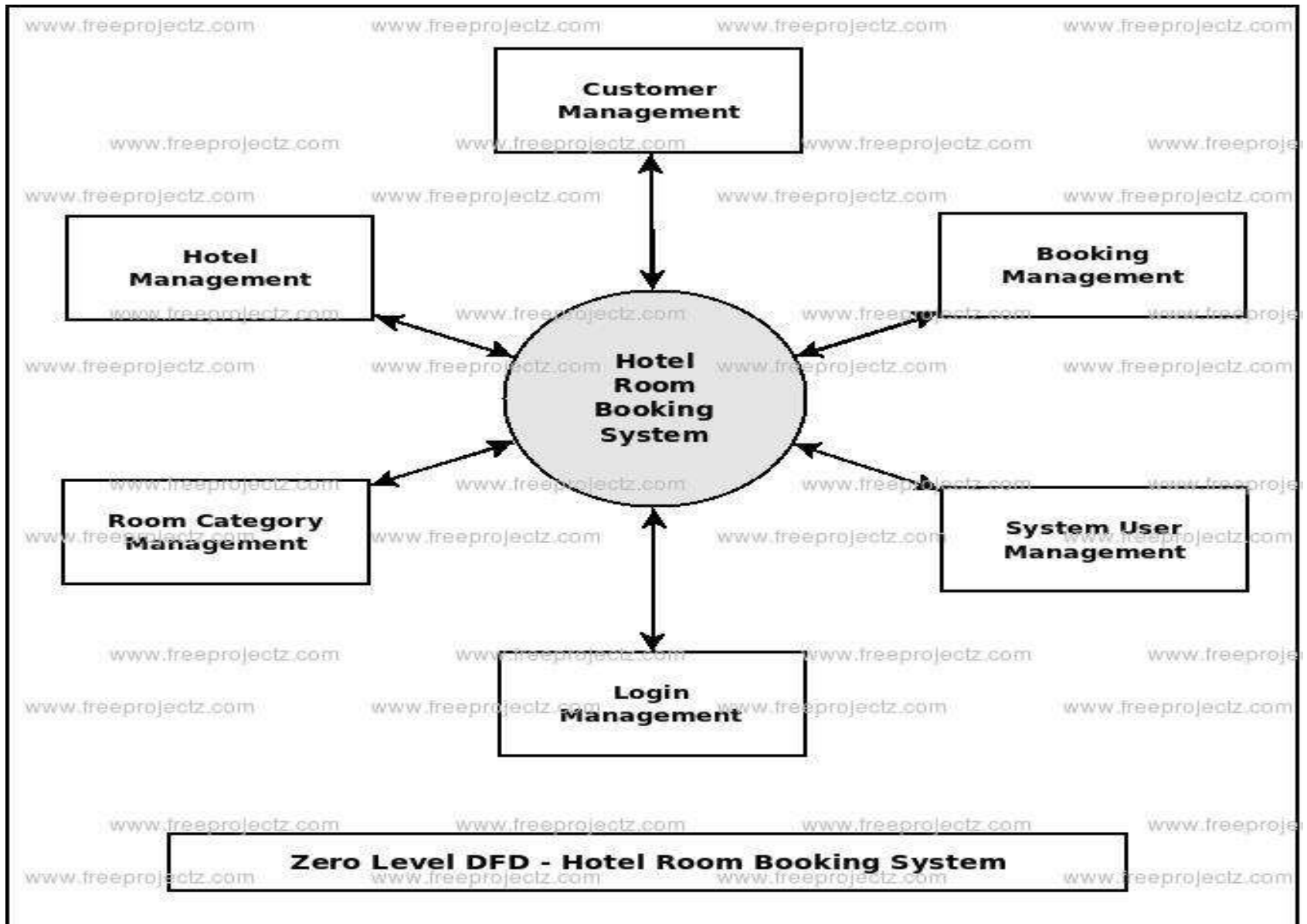
- Context models are used to illustrate the **boundaries** of a system
- Social and organisational concerns may affect the **decision on where to position** system boundaries
- We should make these decisions to Limit the system cost and time needed for analysis
- Architectural models show the a system and its **relationship with other systems**

# The context of an ATM system

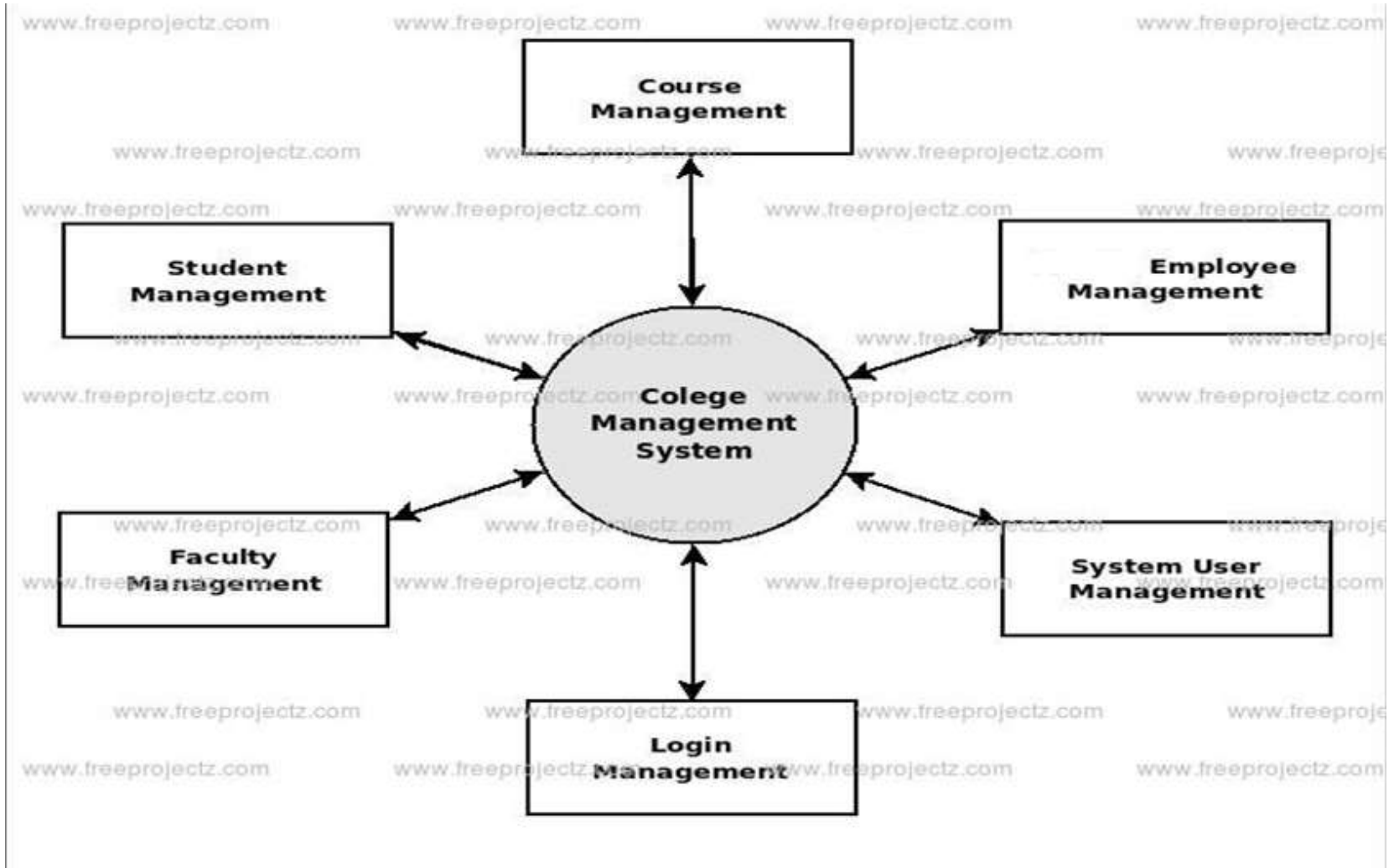


# Cont..

- the fig. shows an architectural model that illustrates the **structure of the information** system that includes a bank Auto Teller Network.
- High-level architectural models are expressed as simple block diagrams where each **sub systems** represented by a **named rectangle**.
- The ATM system is connected to **Account database** and **branch accounting system** and the **security** as well as **maintenance system**.
- The ATM also connected to **usage database** that monitors how the n/w of ATM is used and
- **Counter system** provides the services like backup & printing etc



# Ex:



# Behavioural models

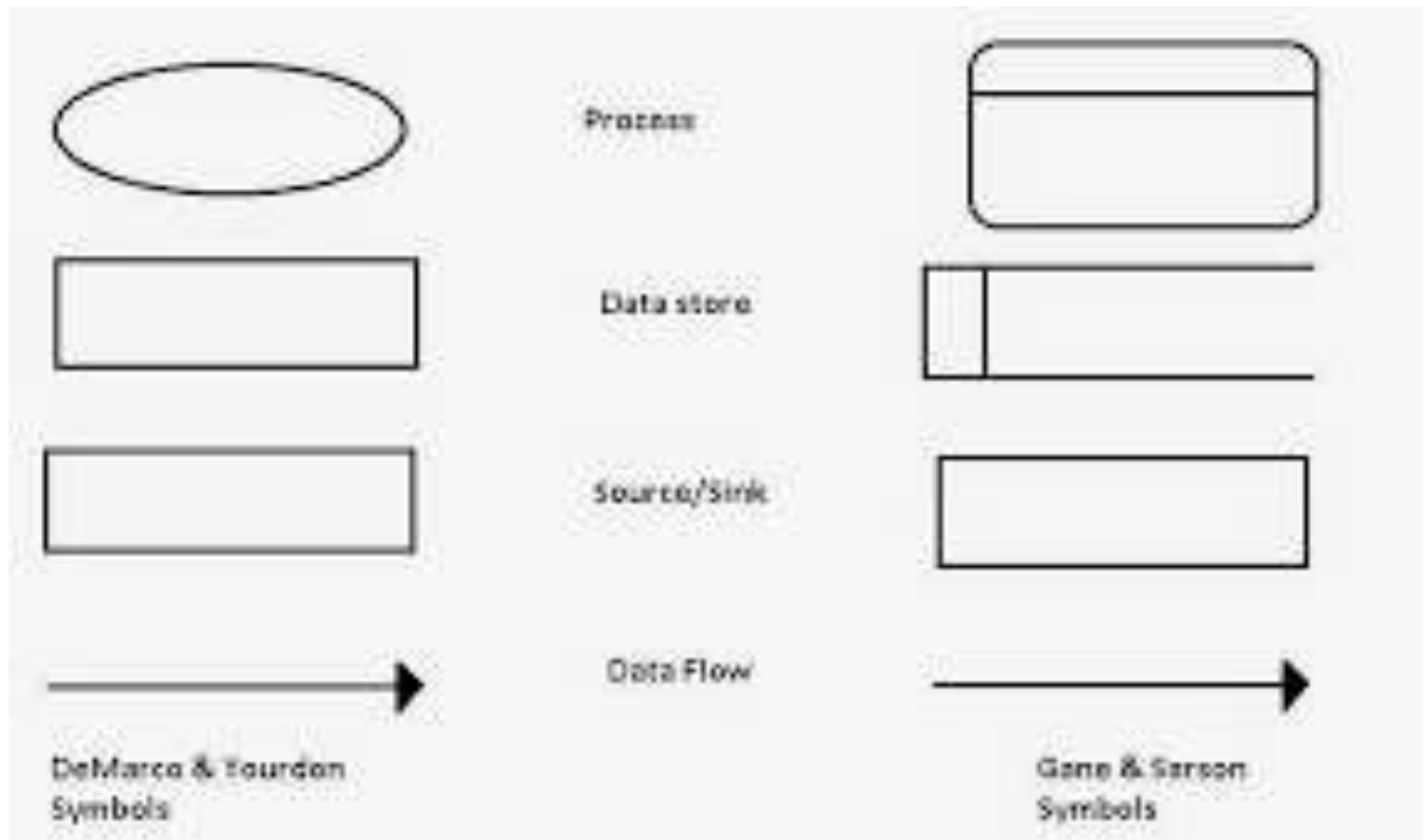
- Behavioural models are used to describe the **overall behaviour of a system**
- Two types of behavioural model are shown here
  - **Data flow models** describes how data is processing in the system
  - **State machine models** describes how the systems reacts to the events or actions
- Both of these models are required for a description of the system's behaviour

## i)Data-flow models

- Most business systems are primarily **driven by data**.
- They are controlled by the **data inputs** to the system
- These models are used at the **analysis level** to model the way in which data is processed
- Data flow diagrams are used to model the system's **data processing**
- Simple and intuitive notation that customers can **understand**
- Show **end-to-end** processing of data



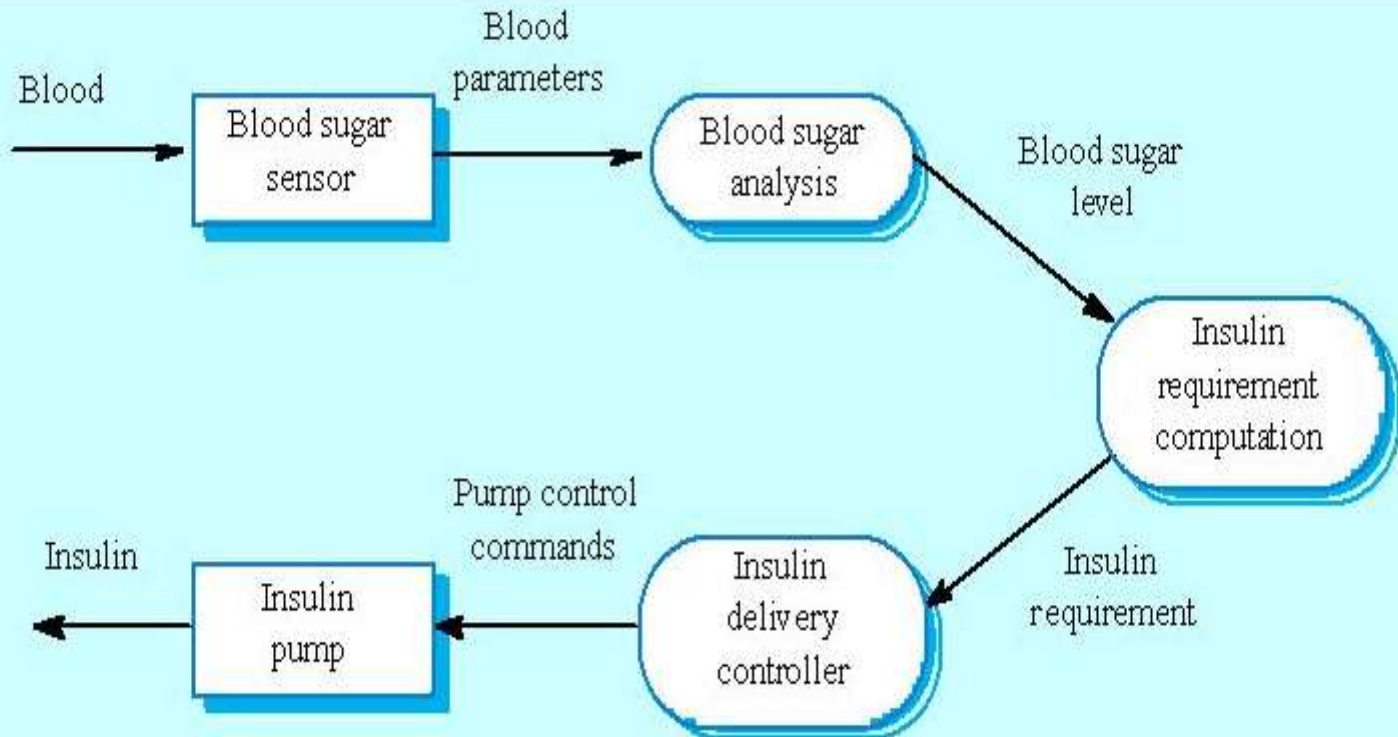
# Cont..



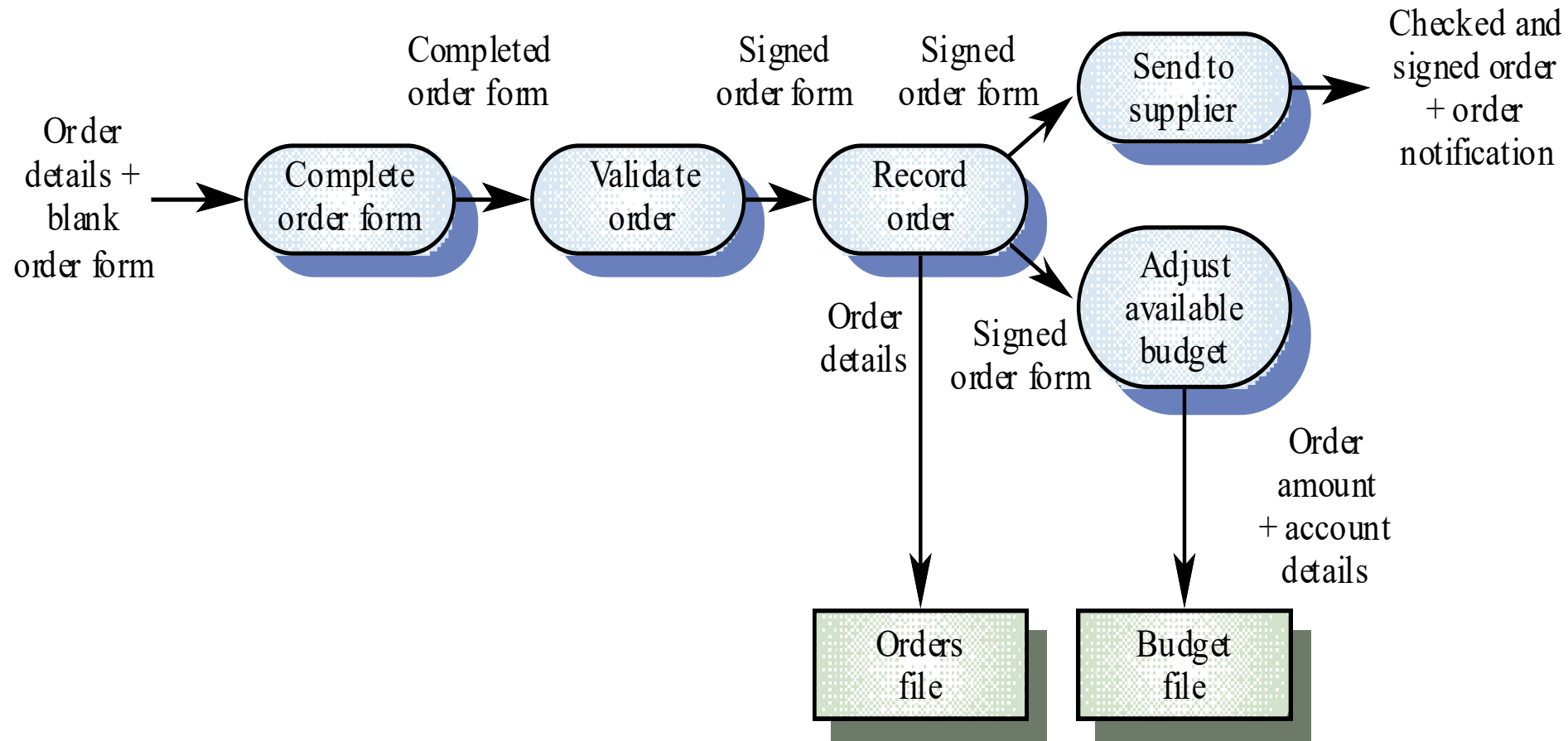
# Cont..

- The **notations used** in this model are **functional processing**(rounded rectangles),**data stores**(rectangles)and **data movements** between functions(labelled arrows)
- DFD are used to show how data flows through a **sequence of processing**
- DFDs model the system from a **functional perspective** where each transformation represents a single function or process
- DFD 's are used in showing the **data exchange between a system and other systems** in its environment

# DFD for insulin pump



# Order processing DFD



# Cont..

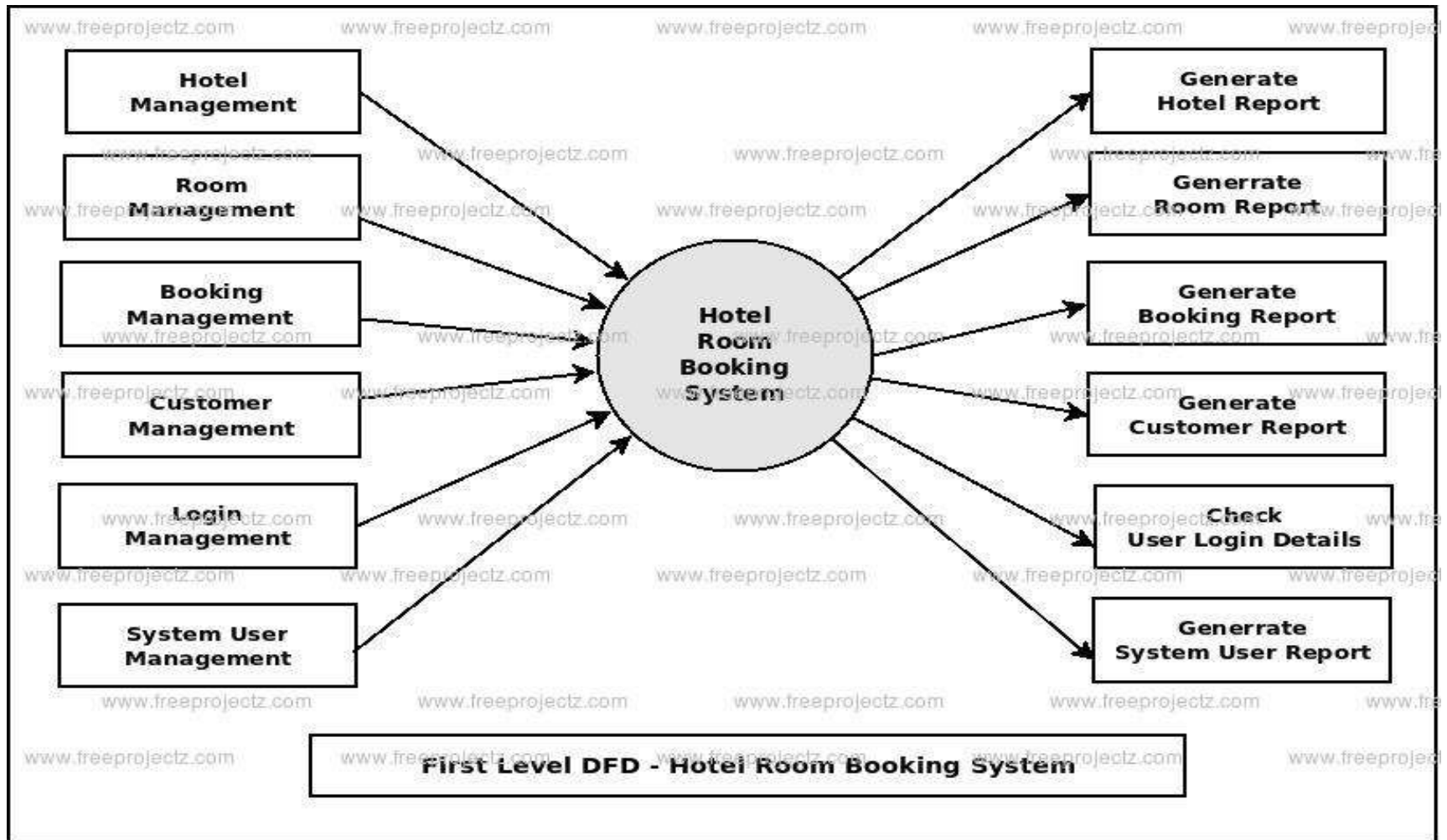
## Advantages Of DFD's:

- It helps us to understand the **functioning and the limits** of a system.
- It is a graphical representation which is very **easy to understand** as it helps visualize contents.
- Data Flow Diagram represent **detailed and well explained diagram** of system components.
- It is used as the part of **system documentation** file.
- Data Flow Diagrams can be understood by both **technical or nontechnical** person because they are very easy to understand.

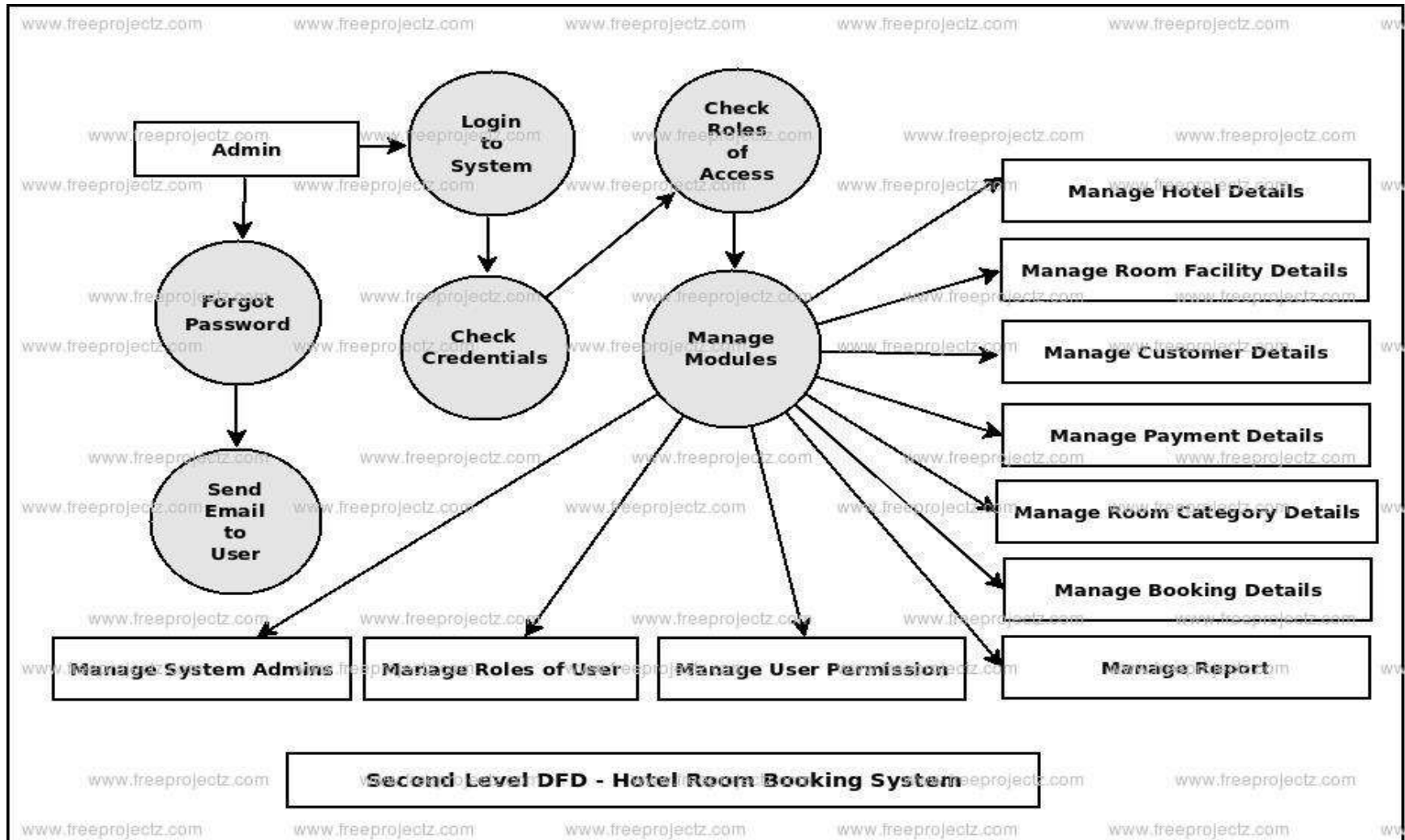
## Disadvantages of DFD

- At times DFD can confuse the programmers regarding the system.
- Data Flow Diagram takes long time to be generated, and many times due to this reasons analysts are denied permission to work on it.

# Ex: First level DFD Hotel Room Booking system



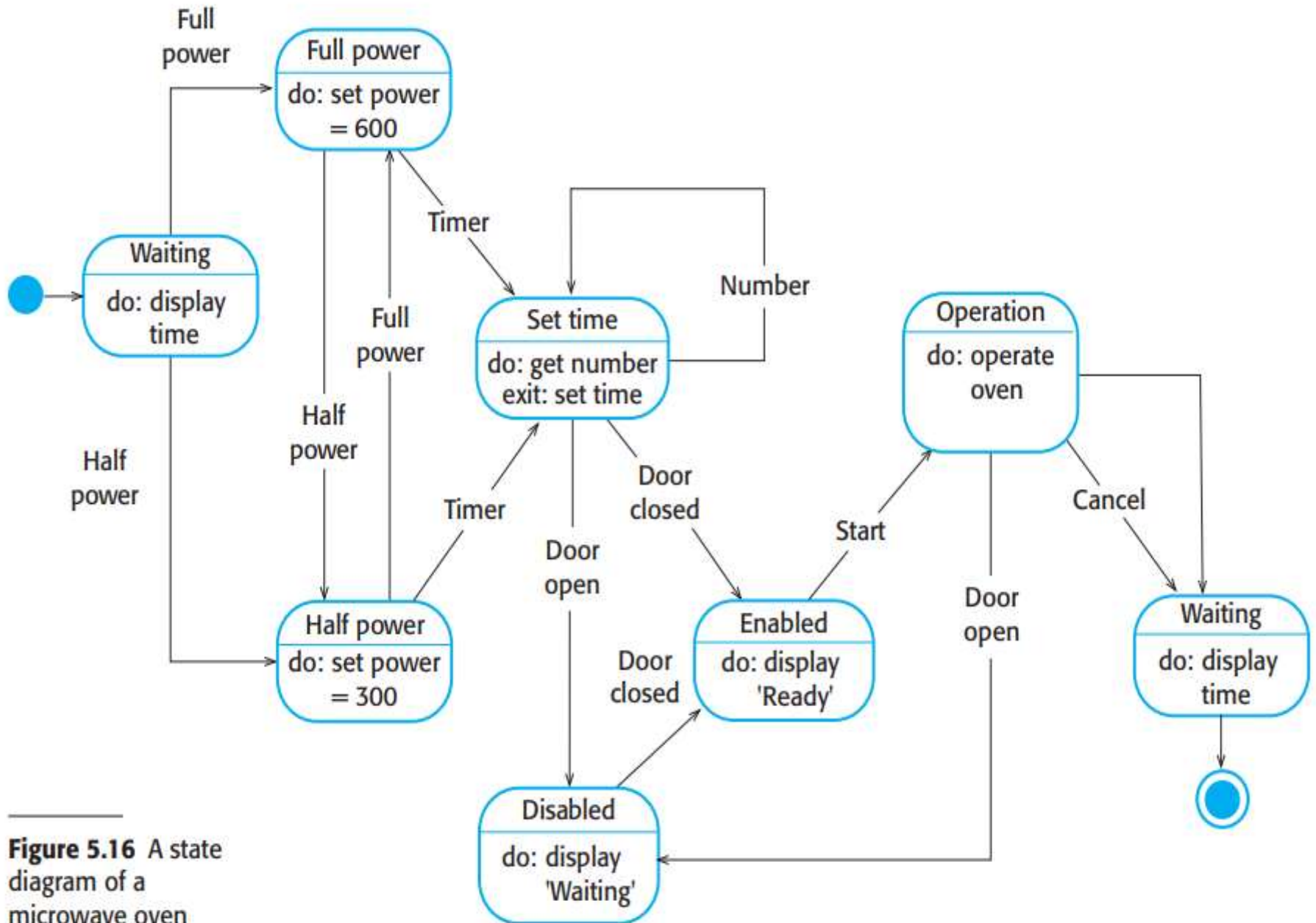
# Level 2 DFD



# State machine models

- These model the **behaviour** of the system in response to external and internal events
- They show the system's **responses to stimuli** so are often used for modelling **real-time systems**
- This model shows **system states and events** that cause **transitions** from one state to other.
- we represent **system states as nodes and events as arcs** between these nodes.
- When an event occurs, the system moves from **one state to another**
- State machine models are an integral part of the **UML**





**Figure 5.16** A state diagram of a microwave oven

# Microwave oven state description

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

# Microwave oven stimuli

<b>Stimulus</b>	<b>Description</b>
Half power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer buttons
Number	The user has pressed a numeric key
Door open	The oven door switch is not closed
Door closed	The oven door switch is closed
Start	The user has pressed the start button
Cancel	The user has pressed the cancel button

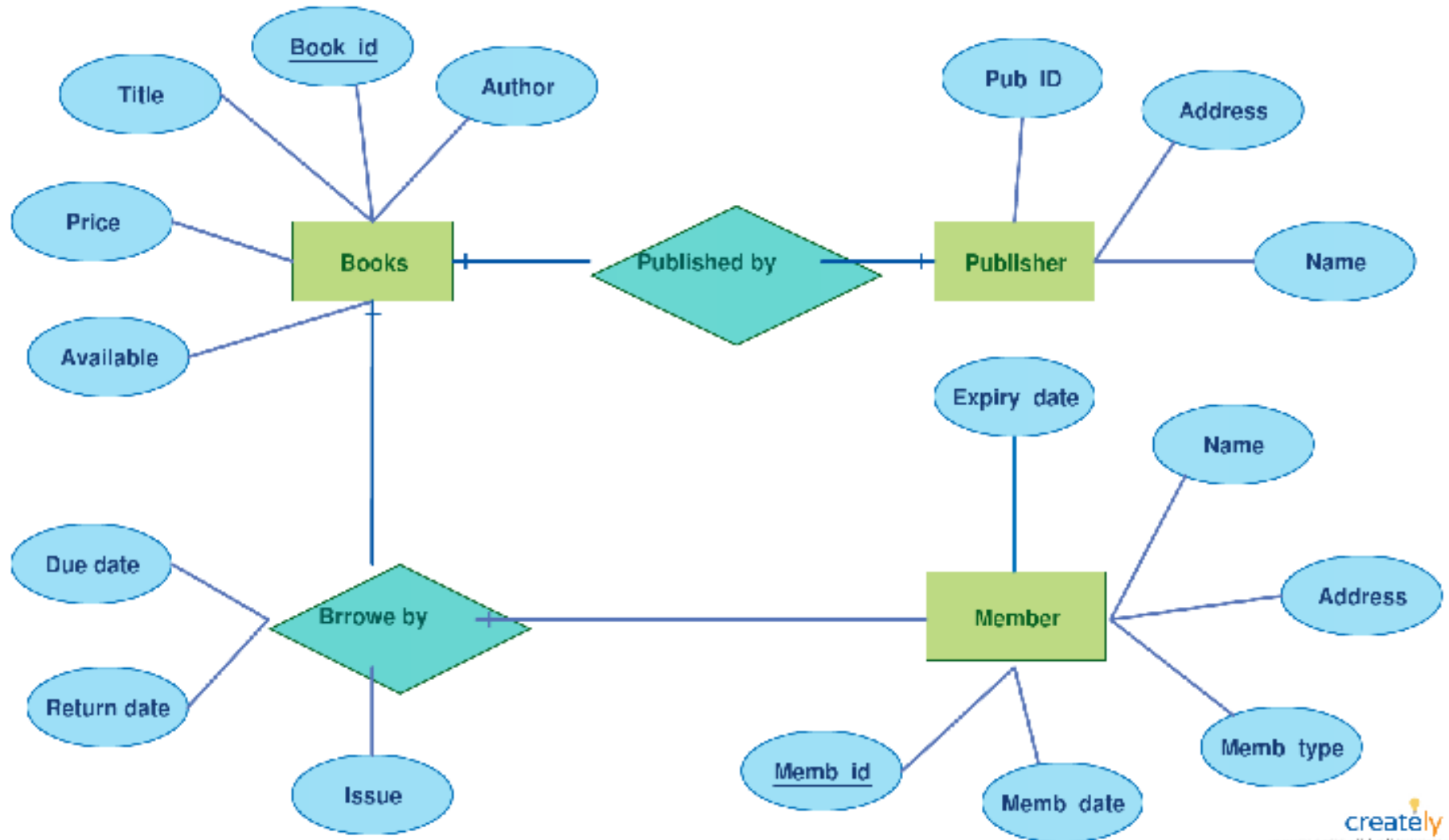
# Cont..

- In the Ex.fig. Microwave oven **equipped with buttons** to set the power and the timer to start the oven
- The sequence of actions used here are
  - select the power level(half or Full)
  - Input the cooking time
  - press start, and food is cooked in given time
- Here rounded rectangles in the diagram represents **system status**.
- These rectangles include a brief **descriptions of the actions** taken in that state
- Advantages:
  - understanding is easy
- Drawback:
  - The no.of possible states increasing rapidly for large systems structuring is **complex**

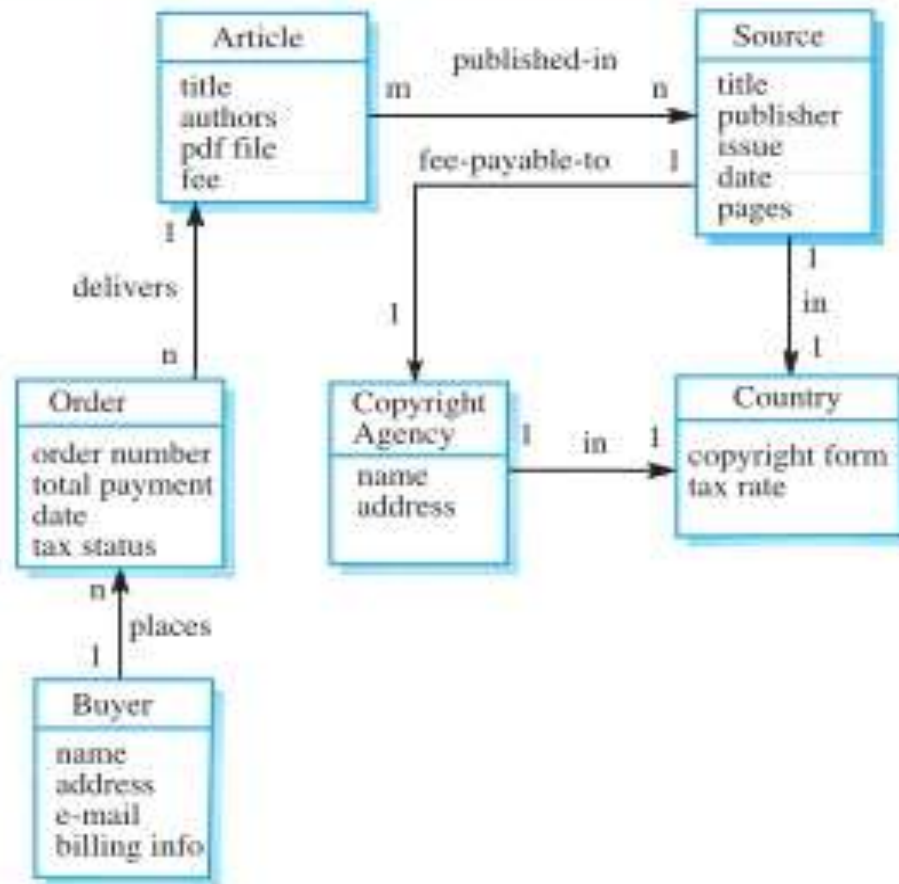
# Semantic Data models

- Most of large s/w systems make use of a **large database of information**
- Used to describe the **logical structure** of data processed by the system
- Most widely used data modelling technique is **entity-relation-attribute** modelling(ERAT)
- **Entity-relationship** models have been widely used database design.
- No specific notation provided in the UML but objects and associations can be used

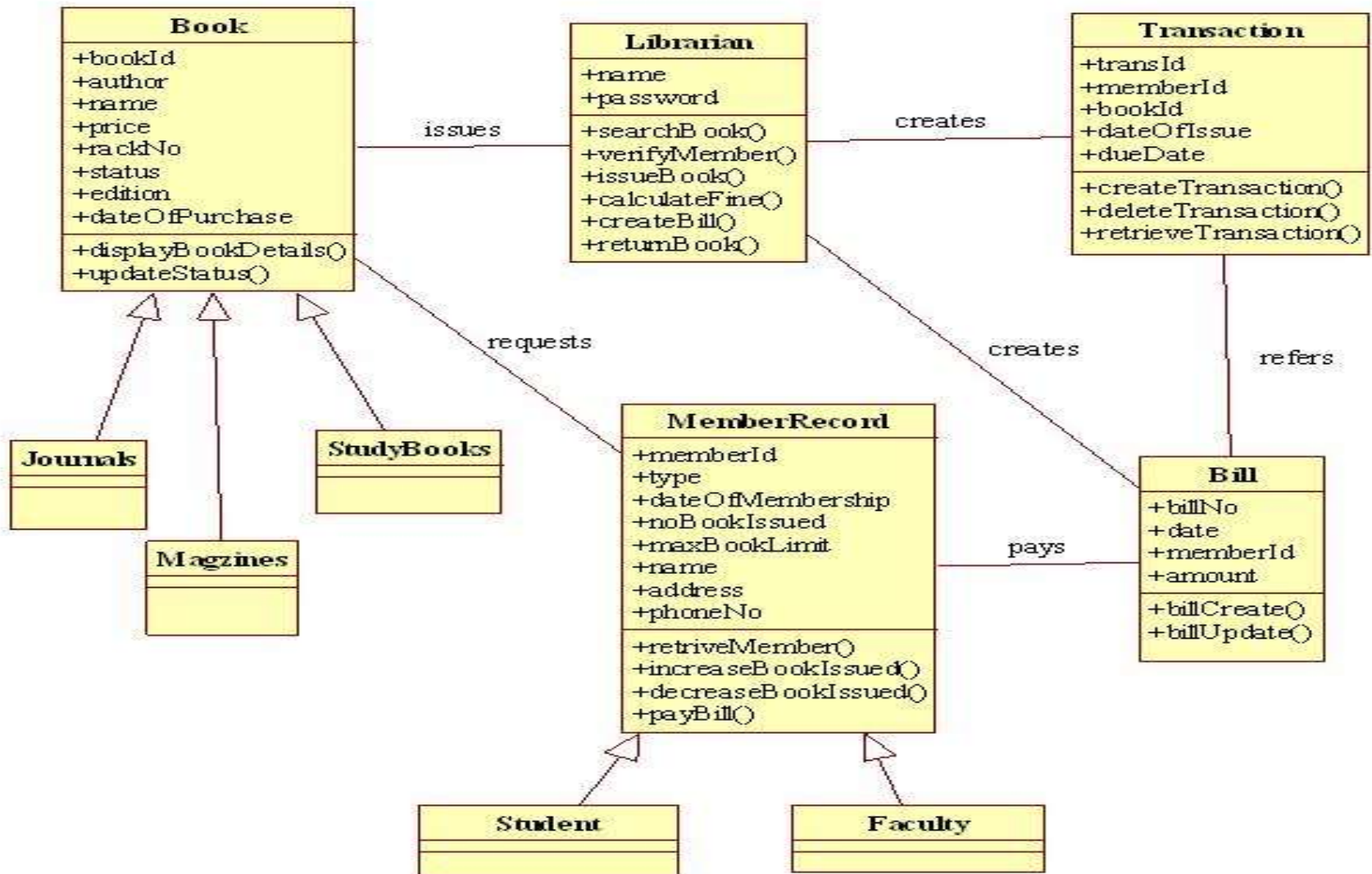
## E-R Diagram for Library Management System



# Library semantic model



# Class diagram for LIBSYS





# Object models

- Object models describe the system in terms of object classes
- **Object** is an instance of a **class**.
- **Object** is a real world entity such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.
- A class is a logical entity
- **Class** is a blueprint or template from which **objects** are created.
- **Class** is a group of similar **objects**.

# Cont..

- An **object class** is an abstraction over a set of objects with common attributes and the services (operations) provided by each object
- Various object models may be produced
  - Inheritance models
  - Aggregation models
  - Interaction models

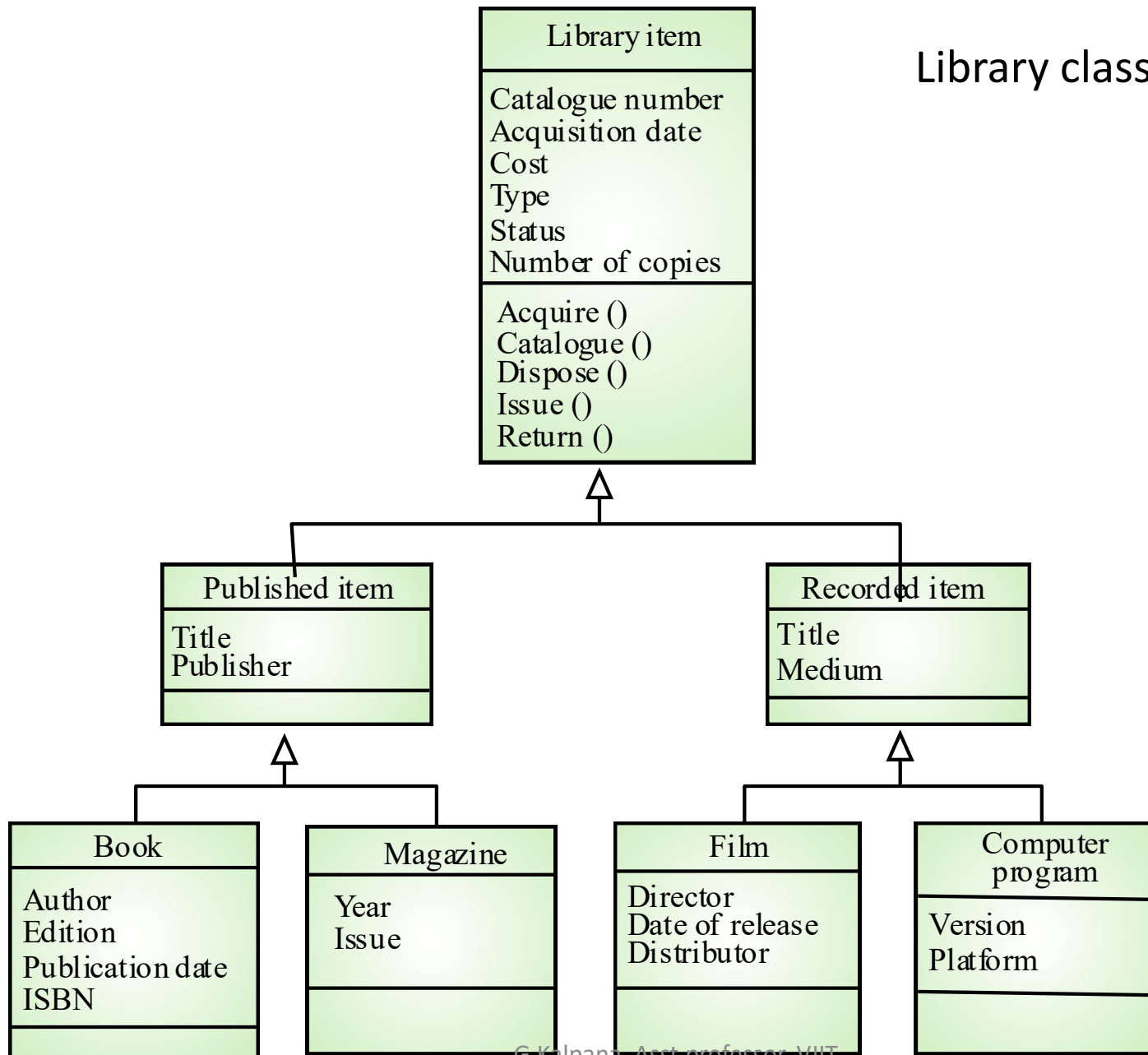
# Unified Modelling Language

- UML is used for object-oriented **analysis and design**
- Has become an **effective standard** for object-oriented modelling
- Notation
  - Object classes are **rectangles** with the name at the top, attributes in the middle section and operations in the bottom section
  - **Relationships** between object classes (known as associations) are shown as lines linking objects
  - **Inheritance** is referred to as generalisation and is shown 'upwards' rather than 'downwards' in a hierarchy

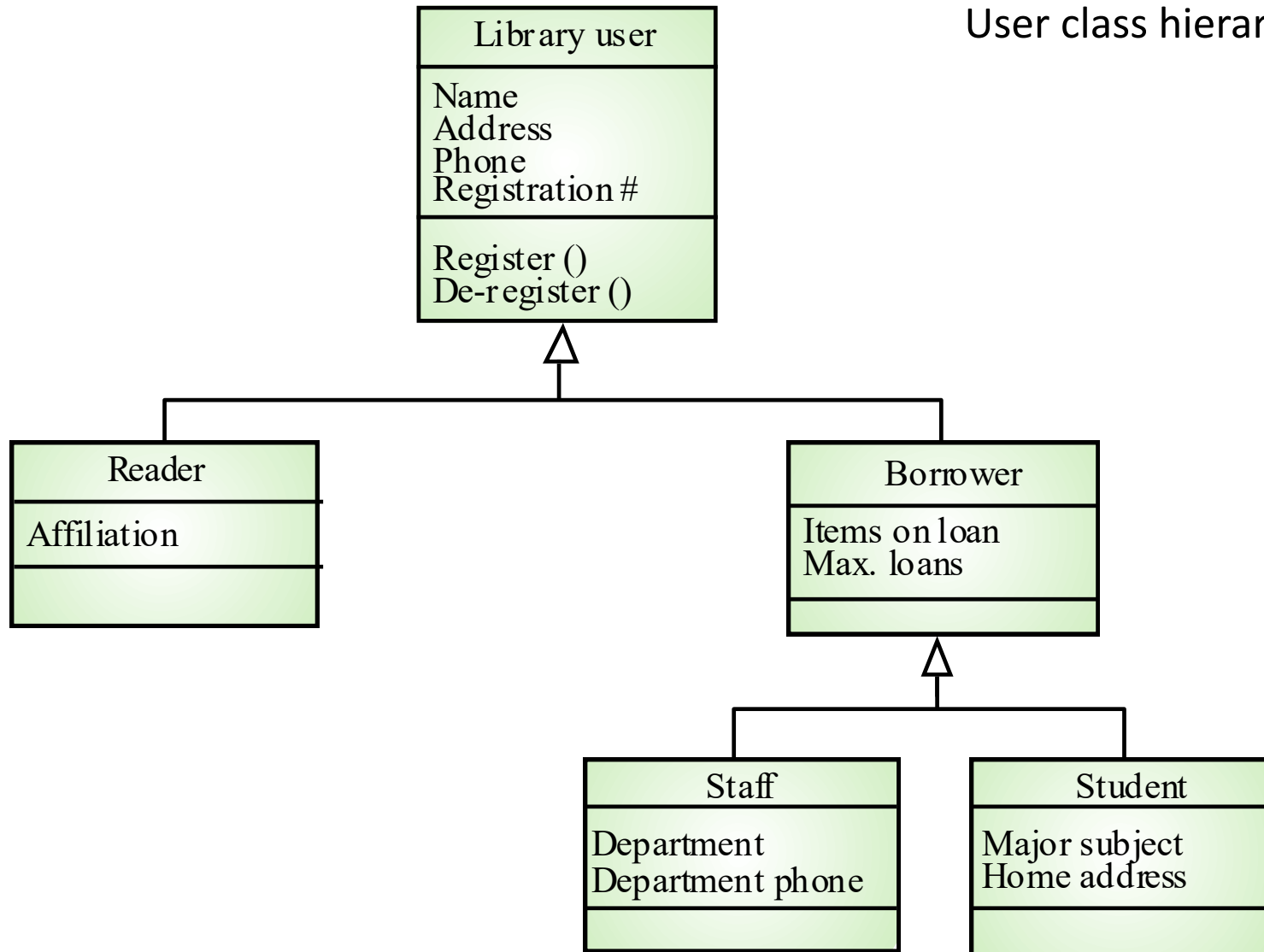
# Inheritance models

- Organise the domain object classes into a **hierarchy**
- Classes at the top of the hierarchy reflect the **common features** of all classes
- Object classes inherit their **attributes and services** from one or more super-classes. these may then be specialised as necessary

## Library class hierarchy



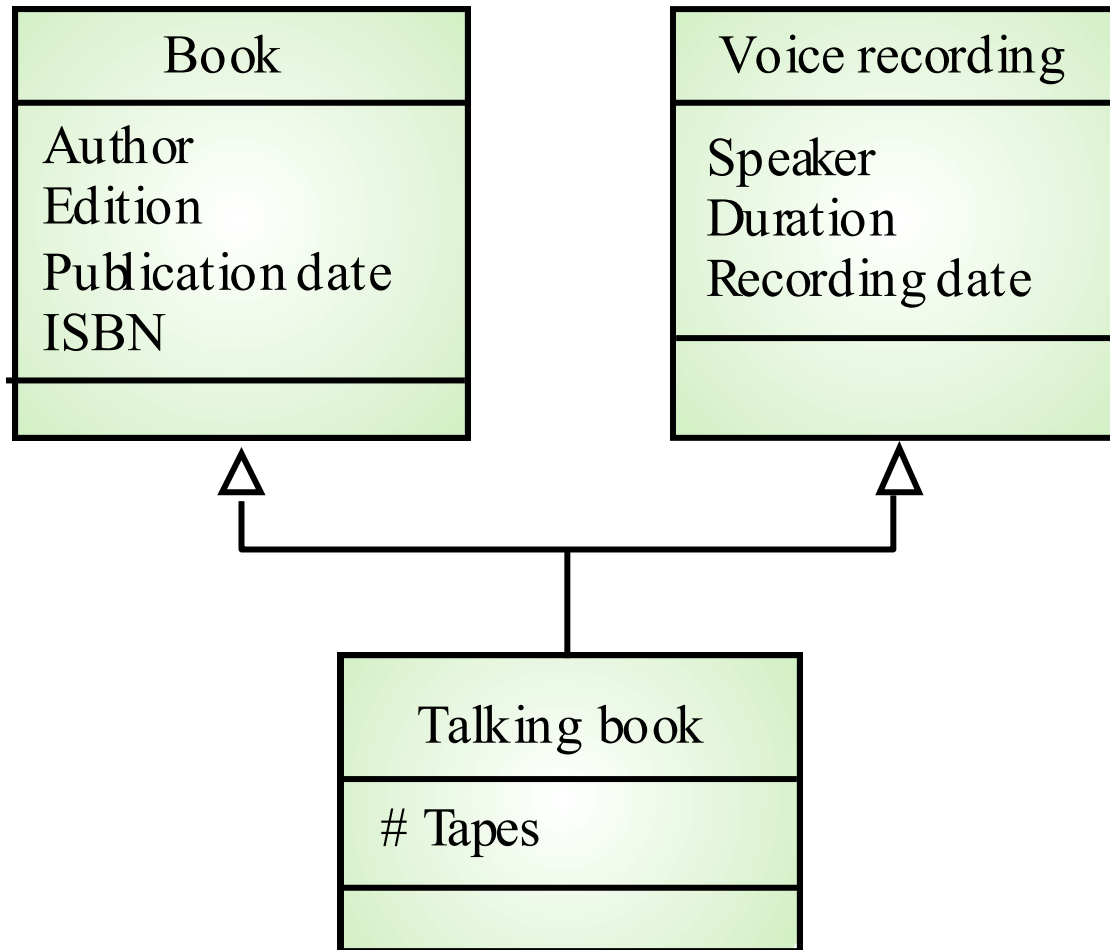
## User class hierarchy



# Multiple inheritance

- Inheriting the attributes and services from **multiple** parent classes
- Can lead to semantic conflicts where attributes/services with the same name in different super-classes have **different semantics**
- Makes class hierarchy **reorganisation** more complex

# Multiple inheritance






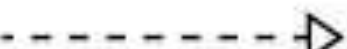


## Aggregation model (composition model)

- Aggregation model shows how classes which are collections are **composed of other classes**
- It is whole-part relationship in semantic data models

# Cont..

UML symbols

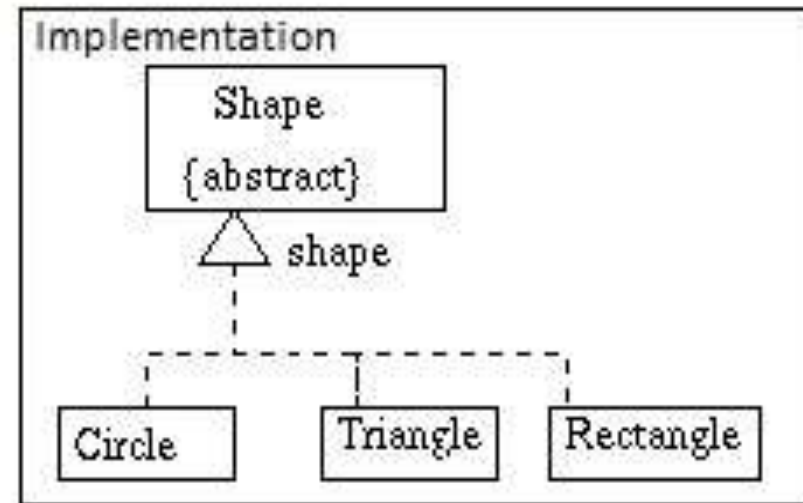
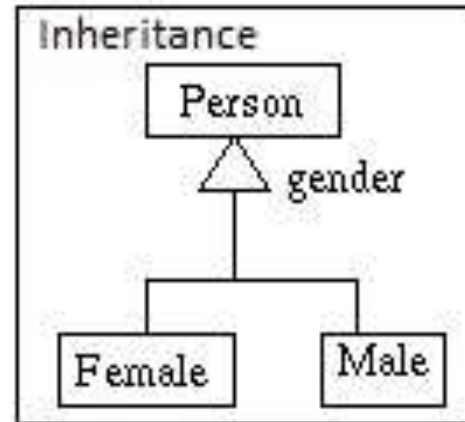
Association	Symbol
Composition	
Aggregation	
Inheritance	
Implementation	



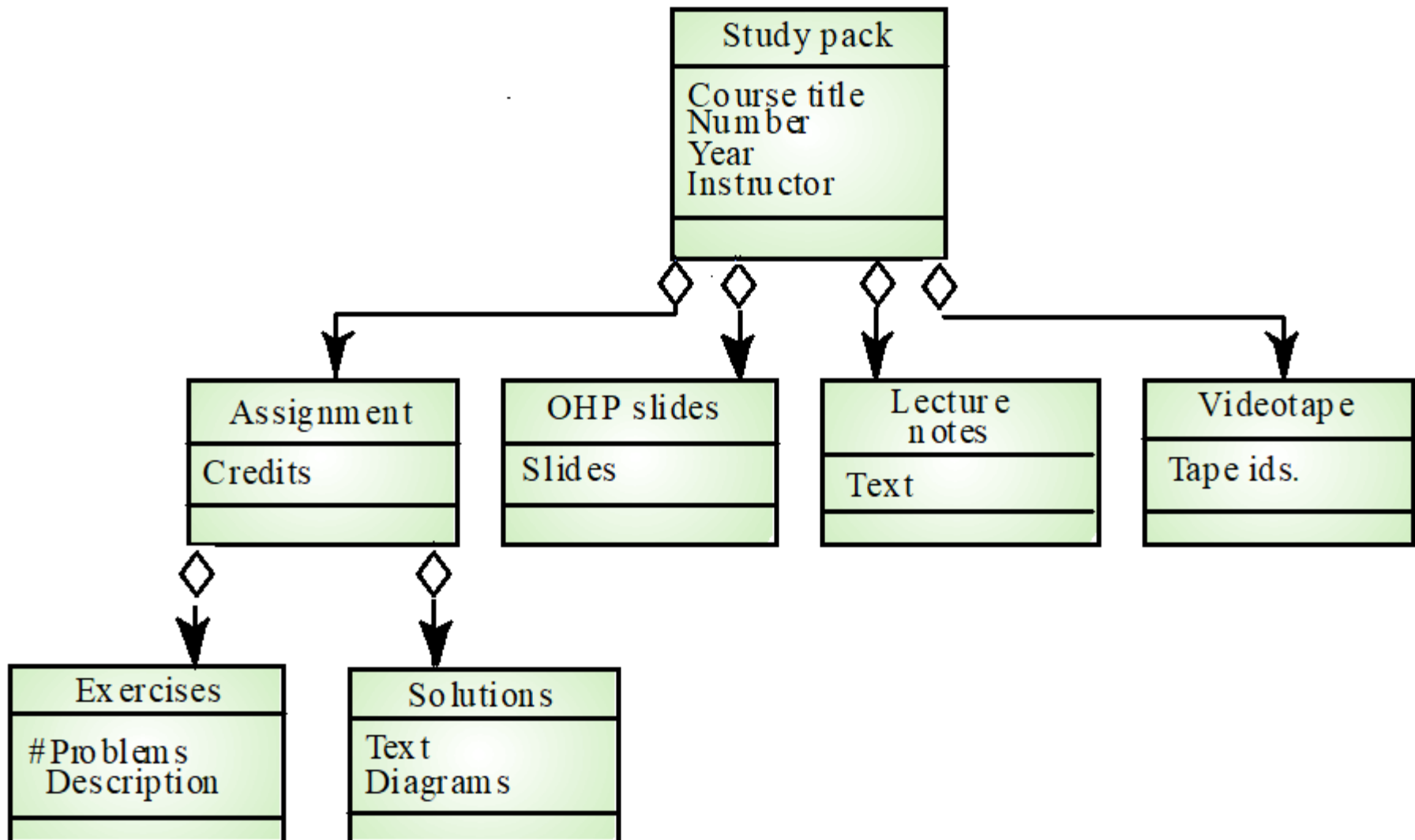
Composition: every car has an engine.



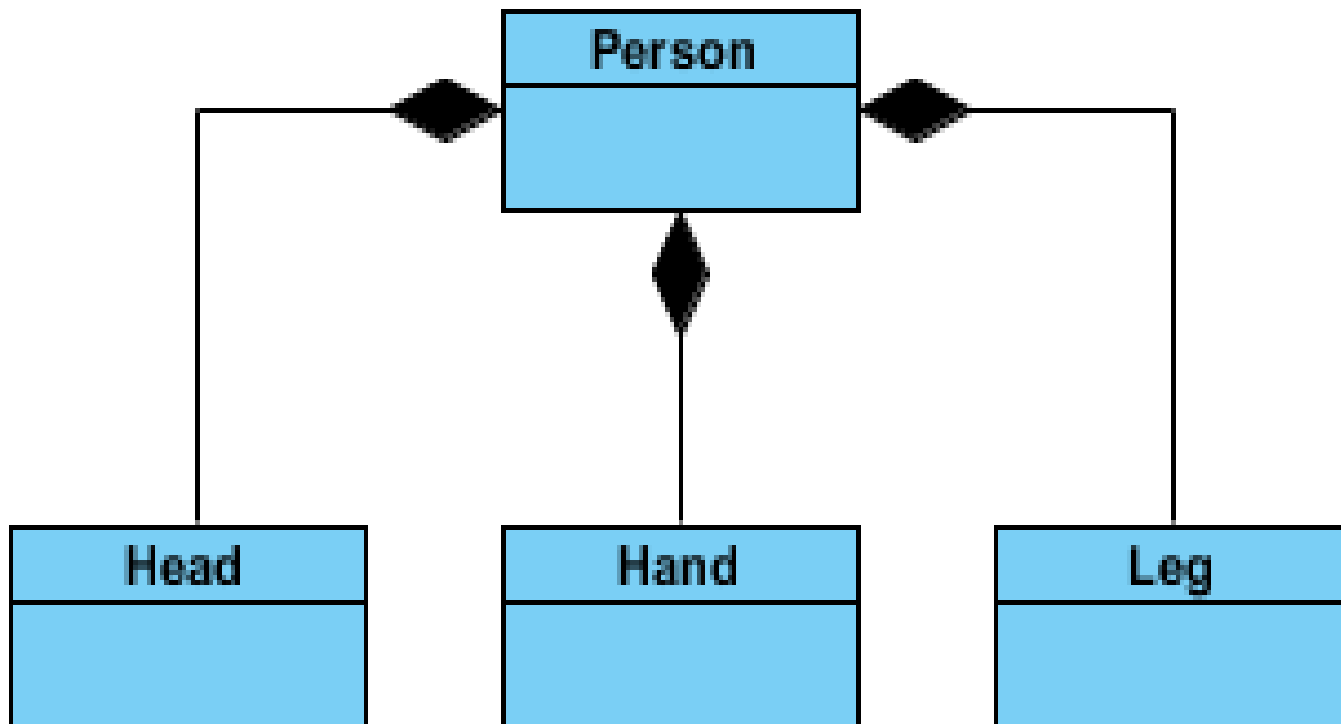
Aggregation: cars may have passengers, they come a



# Ex: Aggregation model



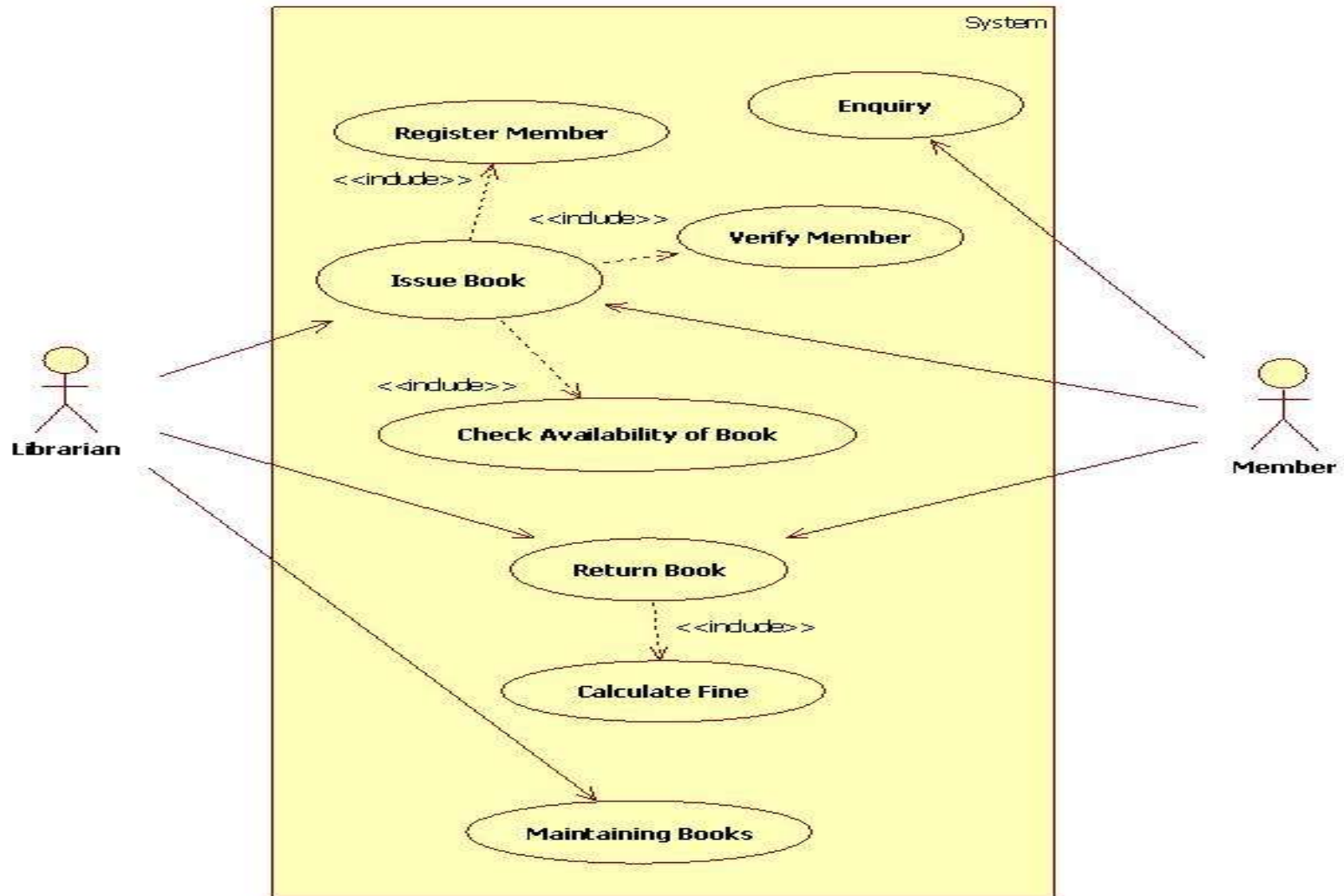
# Ex:composition model



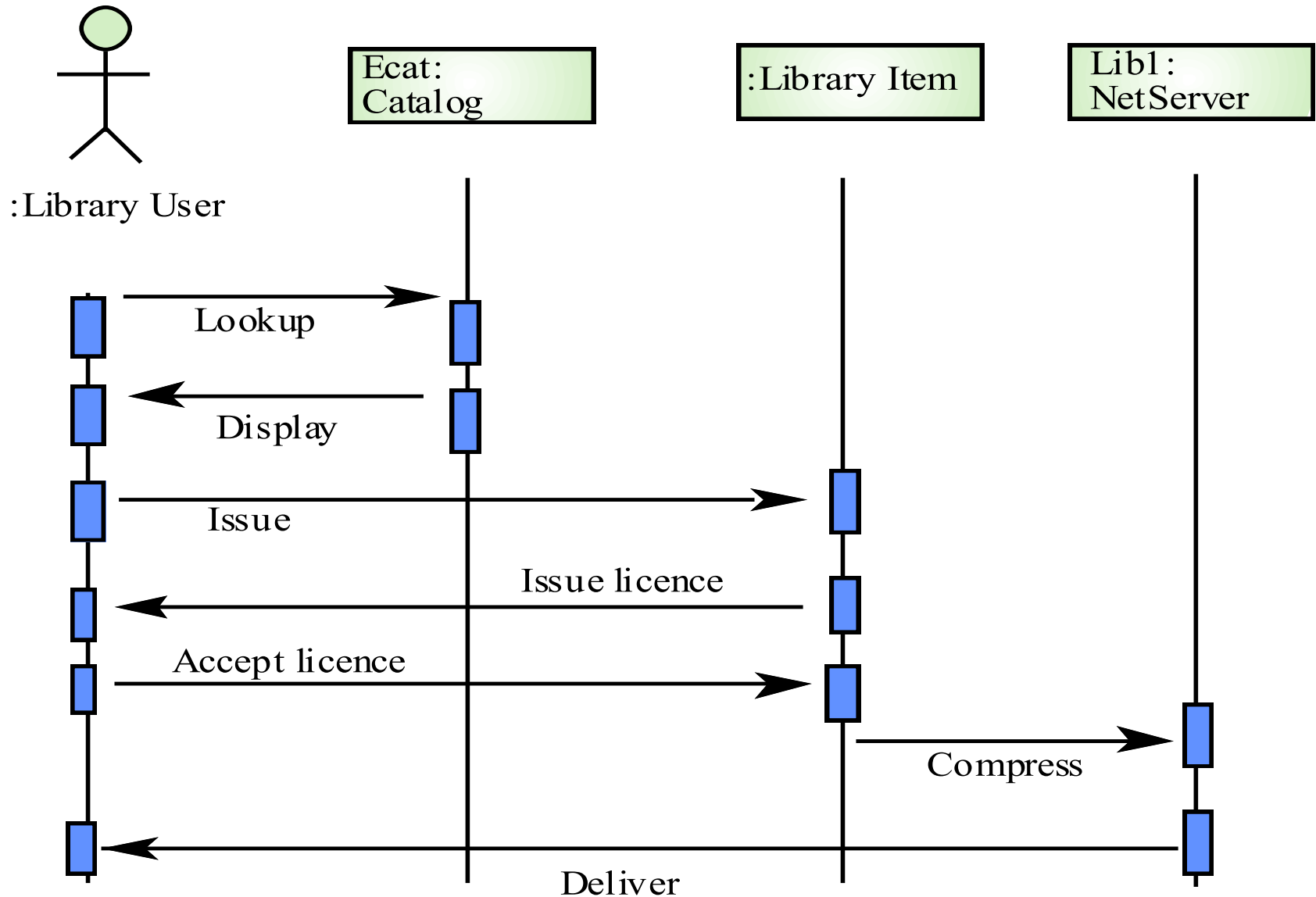
# Object behaviour modelling

- To model the behaviour of objects we need to show the **operations provided by the object** are used
- One way to model the behaviour is to use **UML use-cases**, that show the **sequence of actions** involved
- A behavioural model shows the **interactions between objects** to produce some particular system behaviour that is specified as a **use-case**
- **Sequence diagrams** (or collaboration diagrams) in the UML are used to model **interaction between objects**

# Use-case diagram for LIBSYS



# Object behavioural modelling Ex: Behaviour modelling of LIBSYS (Issue of electronic items)

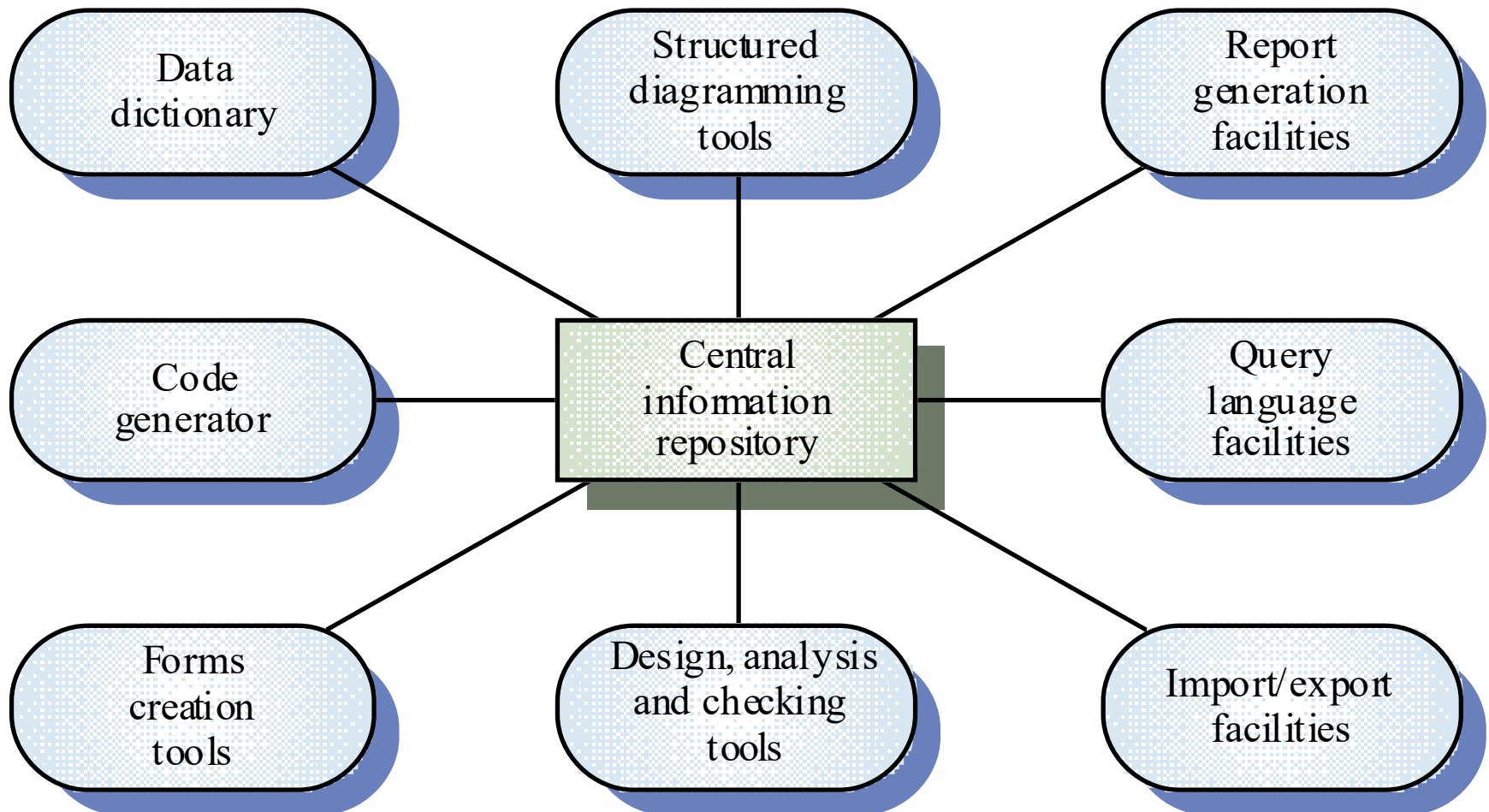


# Structured methods

- Structured method is a **Systematic** way of producing models of a system.
- Developed in 1970 to support s/w analysis and design
- These methods Provides a **frame work** for detailed system modeling as a part of req.elicitation and analysis
- These methods define a **process** that may be used to derive the system models and a **set of rules and guidelines** to the models.
- **CASE tools** are available for method support like editing , code generation , model checking capabilities
- Advantages:
  - applied for large projects to reduce work and fast delivery
  - reduced cost



# An CASE Tool workbench



# Components of CASE workbenches

- **Diagram editors-** used to create object models, data models, behavioural models etc.
- **Design analysis and checking tools-** used to process the design and report on errors and anomalies
- **Repository query languages-** to find designs and related information
- **A data dictionary-** maintains information about attributes used in Database
- **Report definition and generation tools-** take information from central store and automatically generate system documentation
- **Forms definition tools-** allow screen and document formats to be specified
- **Import/export facilities** — Allow interchanging of information from the central store repository with other development tools
- **Code generation-** that generate code or code skeleton automatically from the design captured in the central store

# Design Engineering(part II)

## **Syllabus:**

Design process and Design quality,  
Design concepts,  
the design model,  
Modeling

# introduction

- Design engineering encompass the set of **principles ,concepts and practices** that lead to the development of a **high quality** system or product.
- Design is a place where creativity rules- where **customer requirements, business needs, and technical considerations** all come together in the formulation of a product or system
- Design creates a **representation or model** of the s/w
- but unlike the **analysis model** (that focus on describing required **data, function , and behavior**),

# Cont..

- The **design model** provides detail about the software **data structures, architecture, interfaces, and components**
- **Design engineers** conduct each of the design task.
- The design model can be assessed for **quality** and be improved before code is generated and tests are conducted
  - Does the design contain errors, inconsistencies, or omissions?
  - Are there better design alternatives?
  - Can the design be implemented within the constraints, schedule, and cost that have been established?

# Cont..

- A designer must practice diversification and convergence
  - The designer **selects** from design components, component solutions, and knowledge available through **catalogs, textbooks, and experience**
  - The designer then **chooses** the elements from this collection that meet the requirements
  - Convergence occurs as **alternatives** are considered and rejected until one particular configuration of components is chosen

# Why is it important

- Design allows a **software engineer** to model the system or product that is to be built
- This model can be **assessed for quality** and improved before code is generated, tests are conducted
- Design is the place where **software quality** is established.

# Design process and Quality

- Software design is an **iterative process** through which requirements are translated into a **blueprint** for constructing the software
  - Design begins at a **high level** of abstraction that can be directly traced to the specific **system objective**
  - \_and more detailed **data, functional, and behavioral** requirements
  - As design iteration occurs, **subsequent refinement** leads to design representations at much **lower levels** of abstraction
  - The quality of the evolving design is assessed with a series of **formal technical reviews or design walkthroughs**

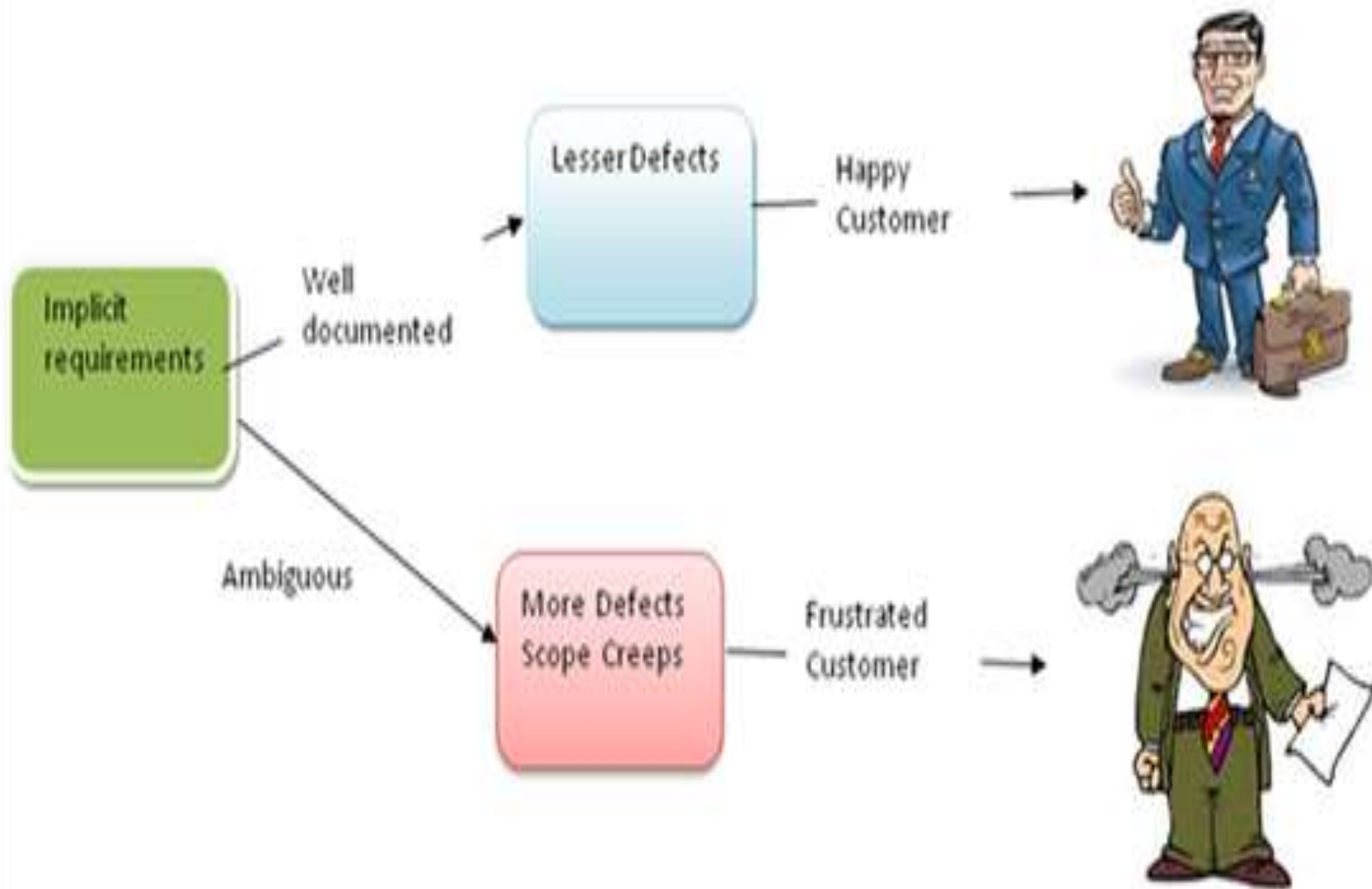


# Goals of Good Design

- The design must **implement** all of the **explicit requirements** contained in the analysis model
- It must also accommodate all of the **implicit** requirements desired by **the customer**
- The design must be a **readable and understandable guide** for those who generate code, and for those who test and support the software
- The design should provide a **complete picture** of the software, addressing the **data, functional, and behavioral** domains from an implementation perspective

# Design Quality Guidelines

- 1) A design should exhibit **an architecture** that
  - a) Has been created using recognizable **architectural styles or patterns**
  - b) Is composed of **components** that exhibit good design characteristics
  - c) Can be implemented in an **evolutionary** fashion, thereby facilitating implementation and testing
  
- 1) A design should be **modular**; that is, the software should be logically partitioned into **elements or subsystems**
  
- 1) A design should contain **distinct representations** of data, architecture, interfaces, and components
  
- 1) A design should lead to **data structures** that are appropriate for the classes to be implemented and are drawn from recognizable data patterns



# Cont..

- 5) A design should lead to **components** that exhibit independent **functional characteristics**
- 5) A design should lead to interfaces that **reduce the complexity of connections** between components and with the external environment
- 5) A design should be derived using a **repeatable method**.
- 5) A design should be represented using a **notation** that effectively communicates its meaning

# Design Quality Attributes(FURPS)

- **Functionality** – is assessed by evaluating the **feature set** and capabilities of the program, the security of overall system
- **Usability** – is assessed by considering **human factors** overall aesthetics, consistency and documentation
- **Reliability**- is evaluated by measuring the **frequency and severity of failure**, the accuracy of output results, the mean-time to failure(MTTF),the ability to recover from failure, and the predictability of the program.
- **Performance**- is measured by processing speed , response time, resource consumption, throughput, and efficiency.
- **Supportability**- combines the ability to extend the program adaptability, serviceability- these three attributes represent a more common term maintainability.

# Design Concepts

- Fundamental s/w design concepts provide the necessary framework for “getting right.”

1) Abstraction

2) Architecture

3) Patterns

4) Modularity

5) Information Hiding

6) Functional independence

7) Refinement

8) Refactoring

# Cont..

- 1) Abstraction

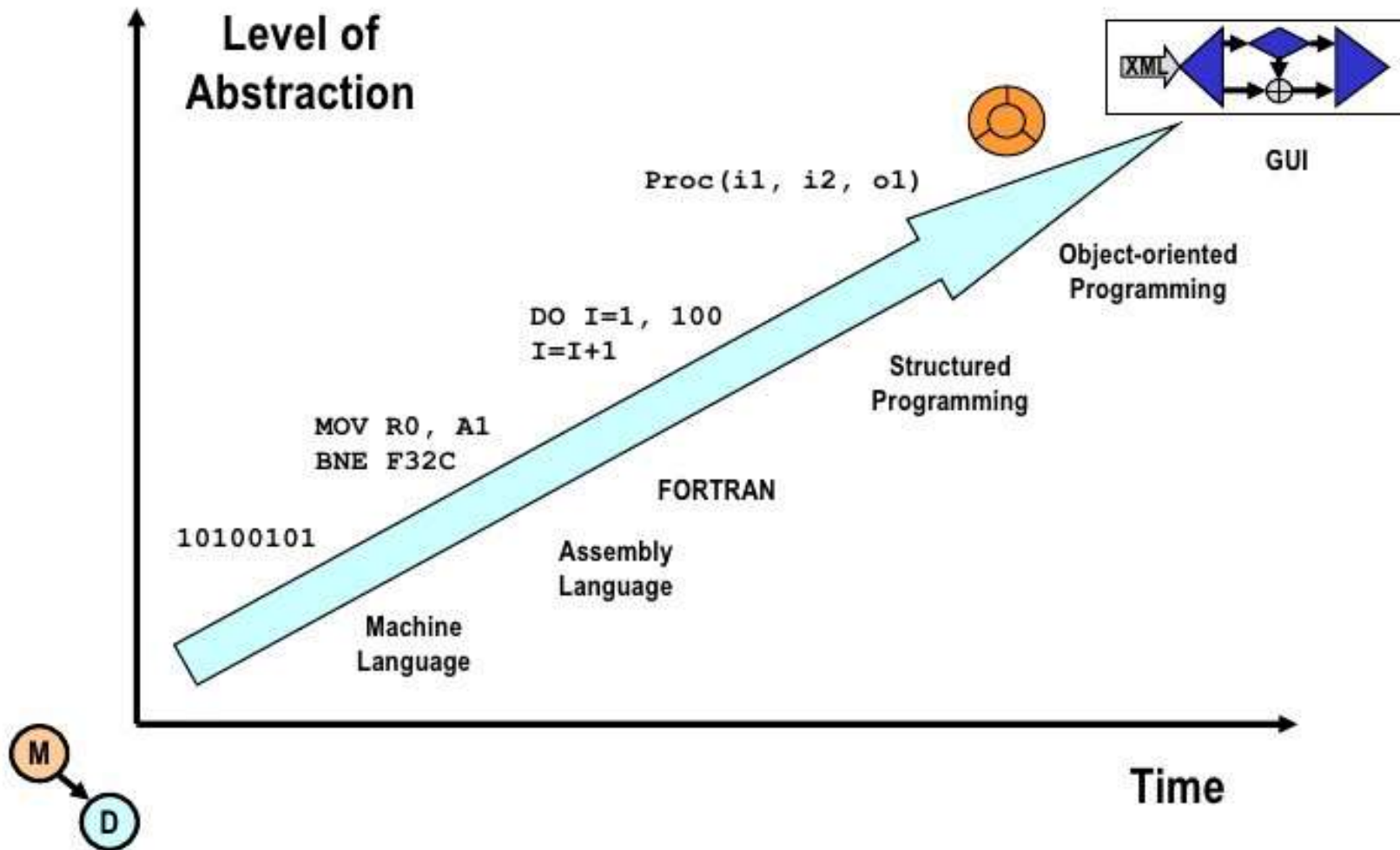
- When we consider a modular solution to any problem, many levels of abstraction can be posed.
- At the **highest level** of abstraction, a solution is stated in **broader terms** using the language of the problem environment.
- At **lower levels** of abstraction, a **more detailed** description of the solution is provided.
- There are two types of abstraction available,

**Data abstraction** – a named collection of data that describes a data object  
Ex: pen – color,ink,size,price,name

**Procedural abstraction** – a sequence of instructions that have a specific and limited function

Ex: display menu, get user options

# Computer Science Is About Abstraction





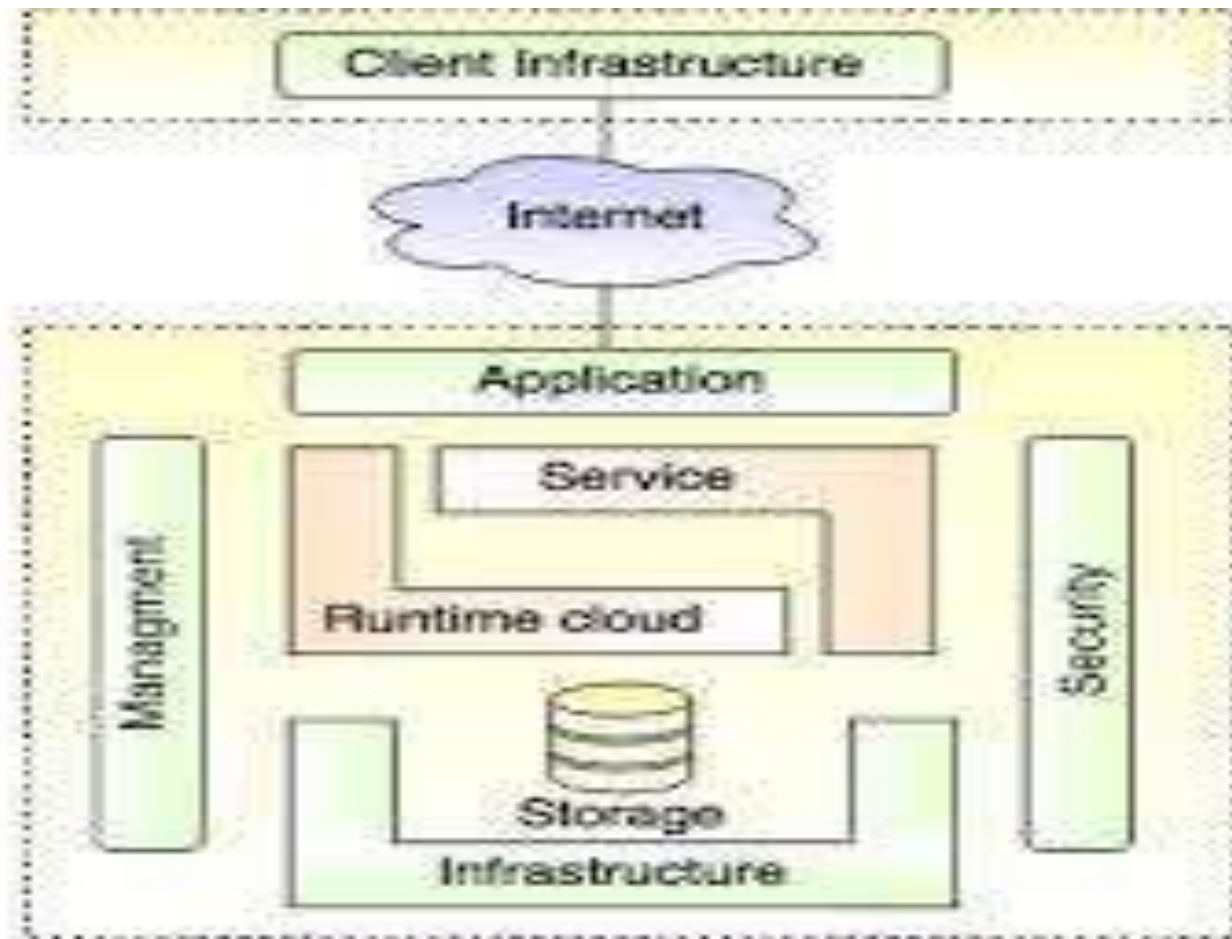
# Cont..

- 2) Architecture
  - The **overall structure** of the software and the ways in which the structure provides conceptual integrity for a system
  - The architectural design can be represented using one or more of a number of **a different models**.
  - **Structural model** represents architecture as an organized **collection of program components**.
  - Consists of components, connectors, and the relationship between them

# Cont..

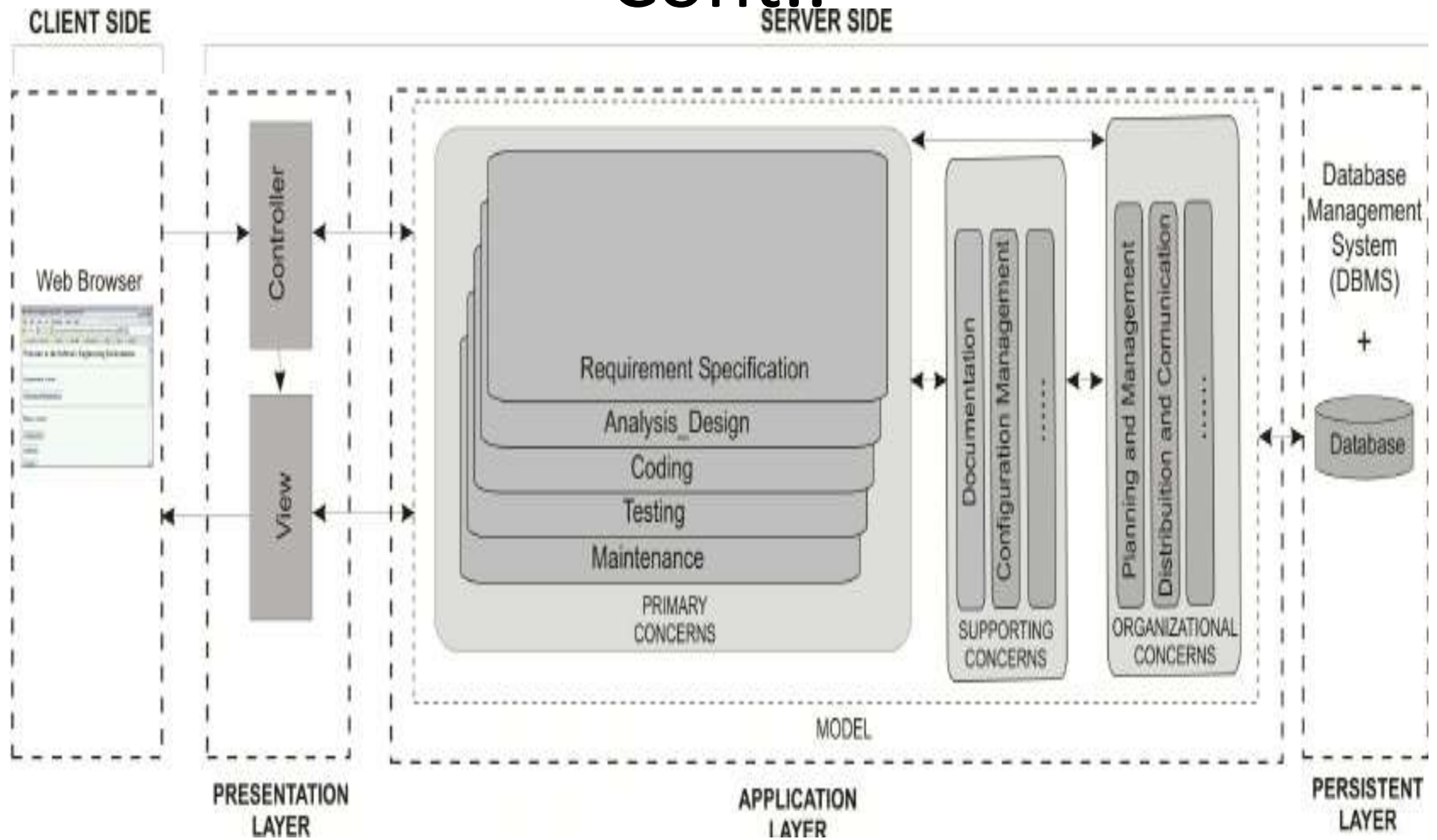
- **Frame work models** increase the level of design abstraction by attempting to identify **repeatable architectural design frameworks** that are encountered in similar type of applications.
- **Dynamic models** address the behavioral aspects of the program architecture , indicating how the **structure or system configuration may change** as a function of external events.
- **Process models** focus on the design of the **business or technical** process that the system must accommodate
- **Functional models** can be used to represents the **functional hierarchy** of a system.

## Ex. for framework Architecture model

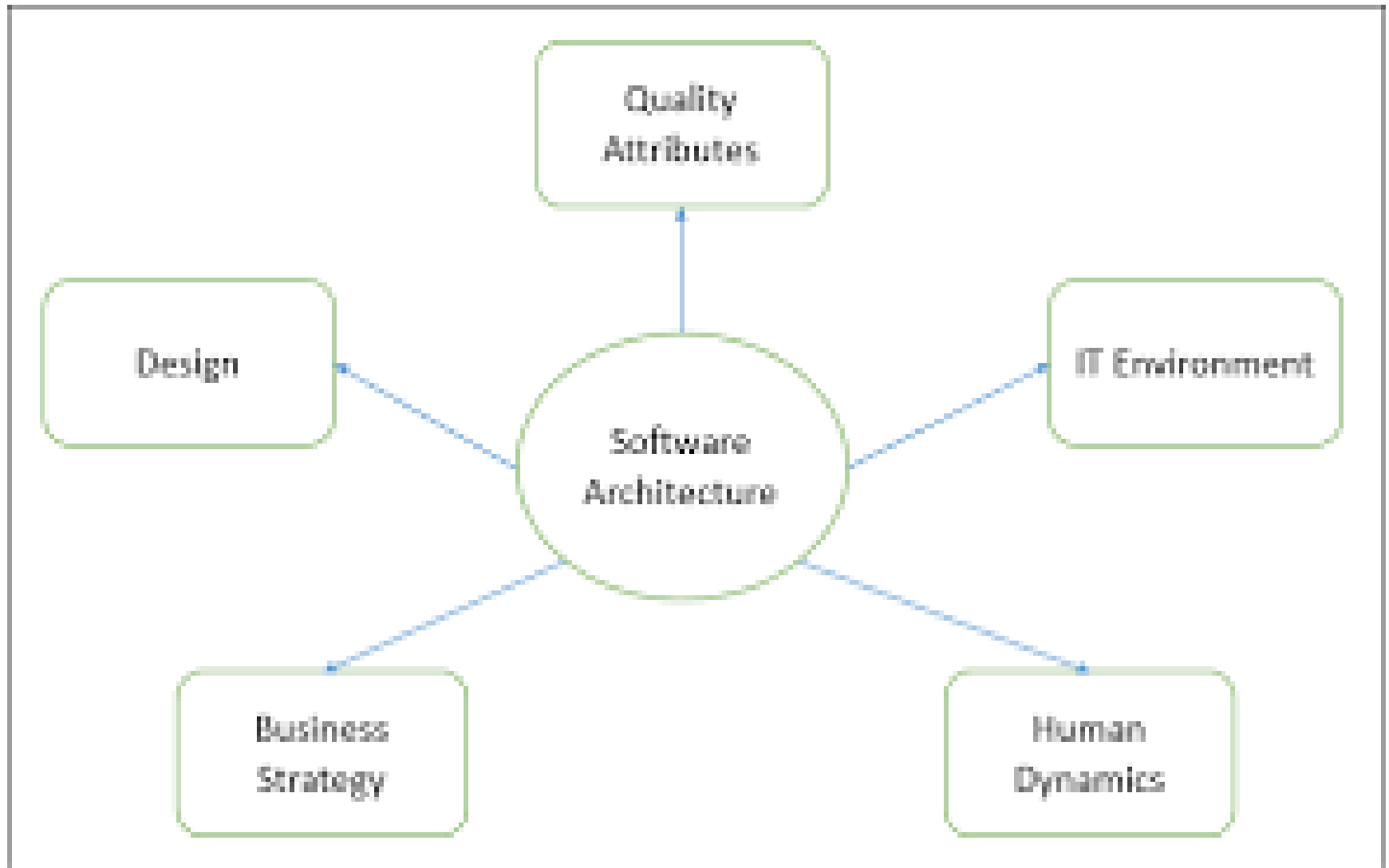


**Fig. - Graphical View of Cloud Computing Architecture**

# Cont..



# Cont..



# Cont..

- 3) Patterns
  - A design structure that solves a particular design problem within a specific context
  - It provides a description that enables a designer to determine
    - i) whether the pattern is applicable to the current work
    - ii) whether the pattern can be reused, and
    - iii) whether the pattern can serve as a guide for developing similar , but functionally or structurally different pattern.

# Cont..

## *Design Pattern Template*

*Pattern name* —describes the essence of the pattern in a short but expressive name

*Intent*—describes the pattern and what it does

*Also-known-as*—lists any synonyms for the pattern

*Motivation*—provides an example of the problem

*Applicability*—notes specific design situations in which the pattern is applicable

- Structure*—describes the classes that are required to implement the pattern
- Participants*—describes the responsibilities of the classes that are required to implement the pattern
- Collaborations*—describes how the participants collaborate to carry out their responsibilities
- Consequences*—describes the “design forces” that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented
- Related patterns* —cross-references related design patterns



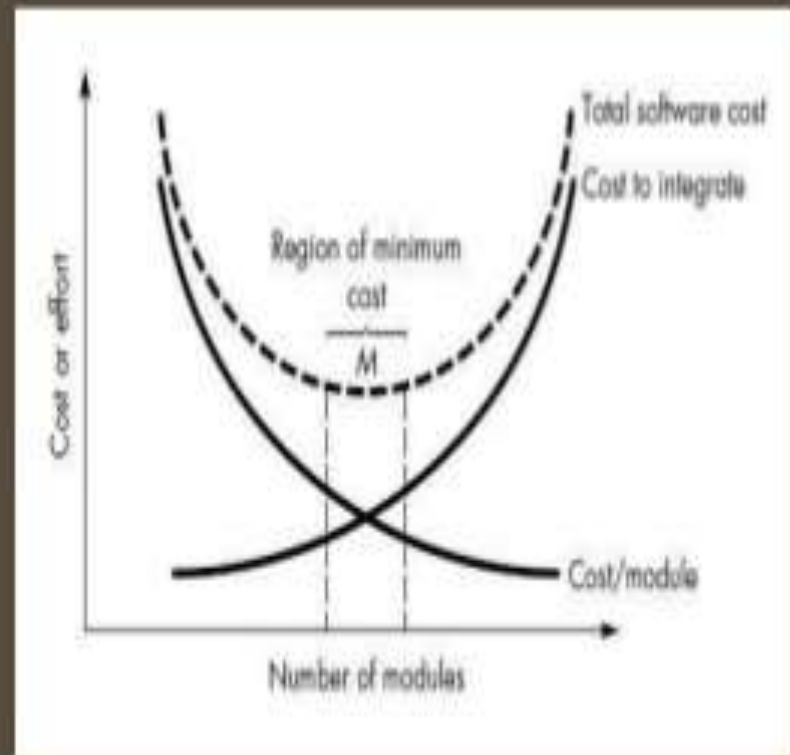
# Cont...

## ● 4)Modularity

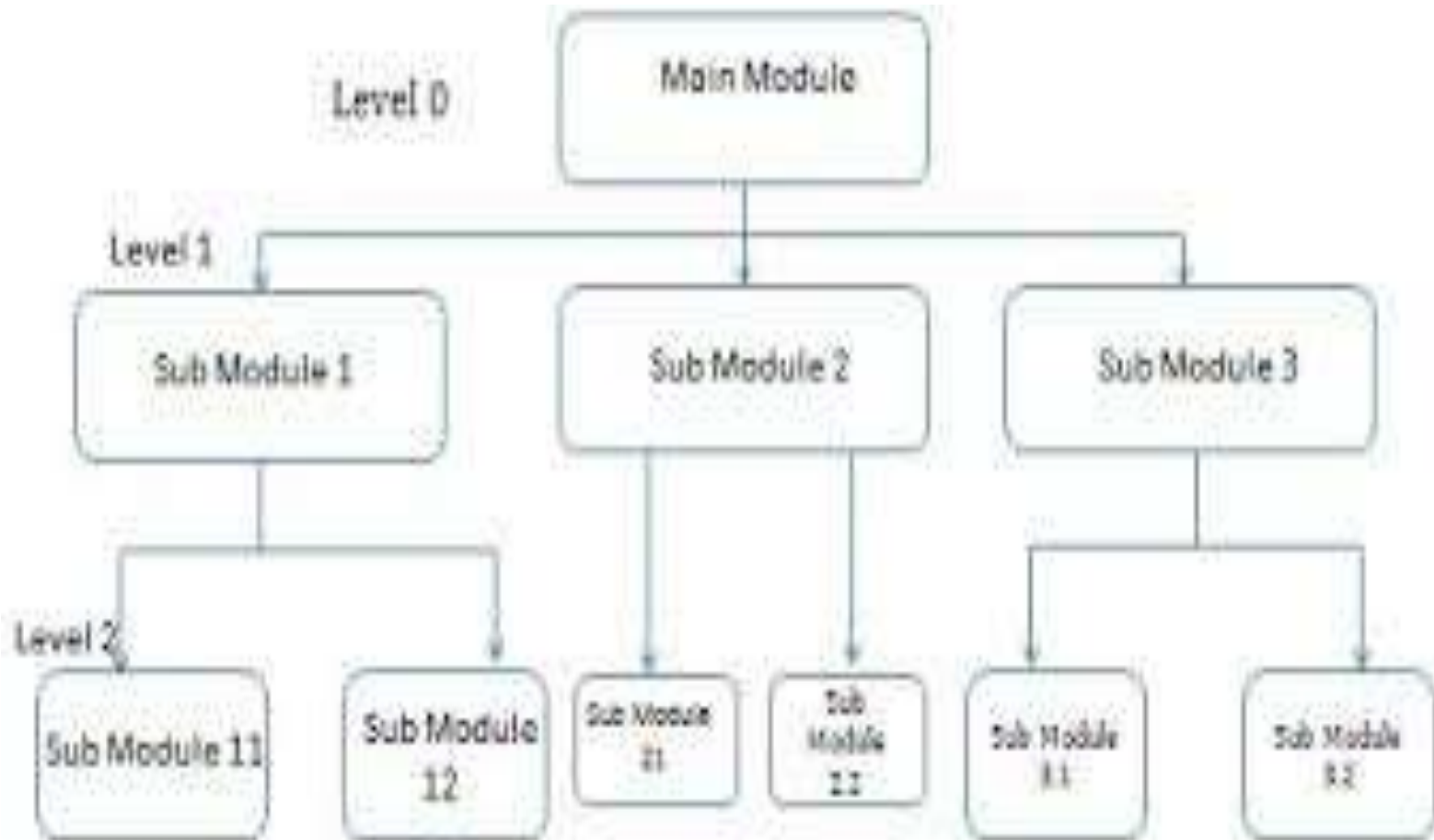
- s/w is divided into **some components** is called modules that are integrated to satisfy problem requirement.
- We modularize a design so that **development** can easily planned.
- s/w increments can be defined and delivered. **Changes** can be more easily accommodated.
- **Testing and debugging** can be conducted more efficiently
- and **long term maintenance** can be conducted without serious side effects.
- The effort (cost) to develop an individual software module does **decrease** as the total number of **modules increases**.

# MODULARITY

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.



# Ex:



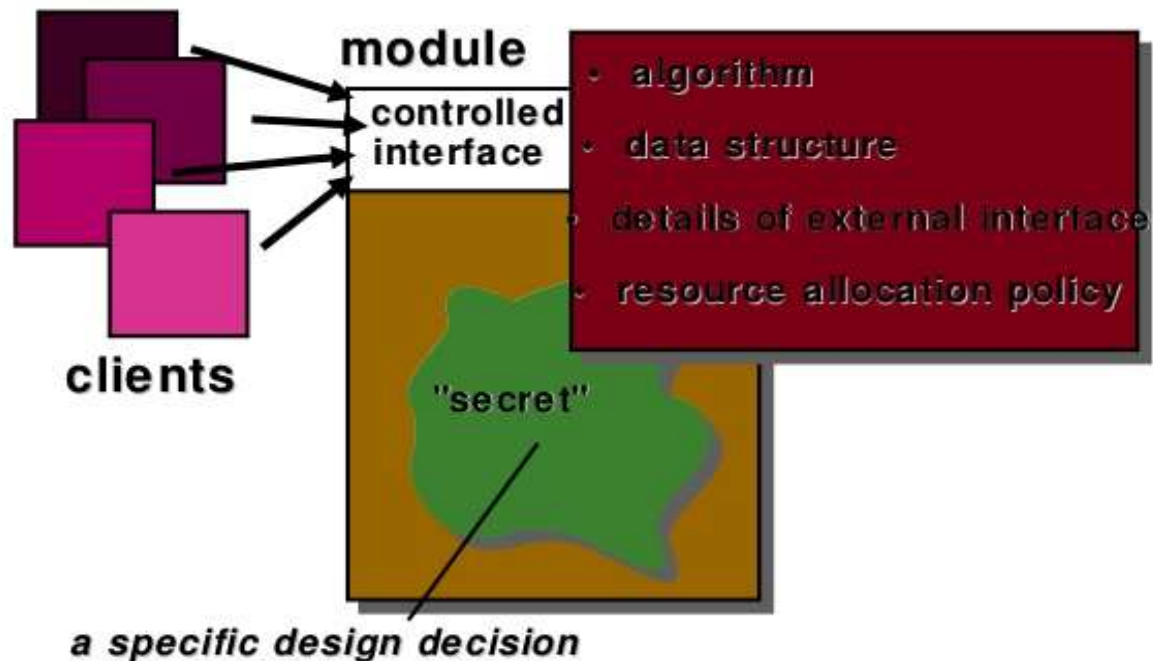
# Cont..

- 5)Information hiding

- The modules be “characterized by design decisions **that hides** from all others”
- ie modules should be specified and designed so that information contained within a module is **inaccessible to other modules** that have no need for such information.
- Hiding implies that **effective modularity** can be achieved by defining a set of **independent** modules that communicate with one another only that information necessary to achieve s/w function.
- Hiding prevents **error propagation** outside of a module
- Results in **higher quality** software

# Cont..

## Information Hiding



# cont..

- 6)Functional independence

- It is achieved by developing modules with “single minded” function and an “aversion to excessive interaction with other modules.
- Functional independence is key to good design, and design is the key to s/w quality.
- Independence is assessed using two qualitative criteria ie cohesion and coupling
- Cohesion is an indication of the relative functional strength of a module .
- High cohesion – a module performs only a single task
- Coupling is an indication of the relative interdependence among modules.
- Low coupling – a module has the lowest amount of connection needed with other modules

# Cont..

## Cohesion

Vs

## Coupling

Cohesion is the concept of intra module.

Coupling is the concept of inter module.

Cohesion represents the relationship within module.

Coupling represents the relationships between modules.

Increasing in cohesion is good for software.

Increasing in coupling is avoided for software.

Cohesion represents the functional strength of modules.

Coupling represents the independence among modules.

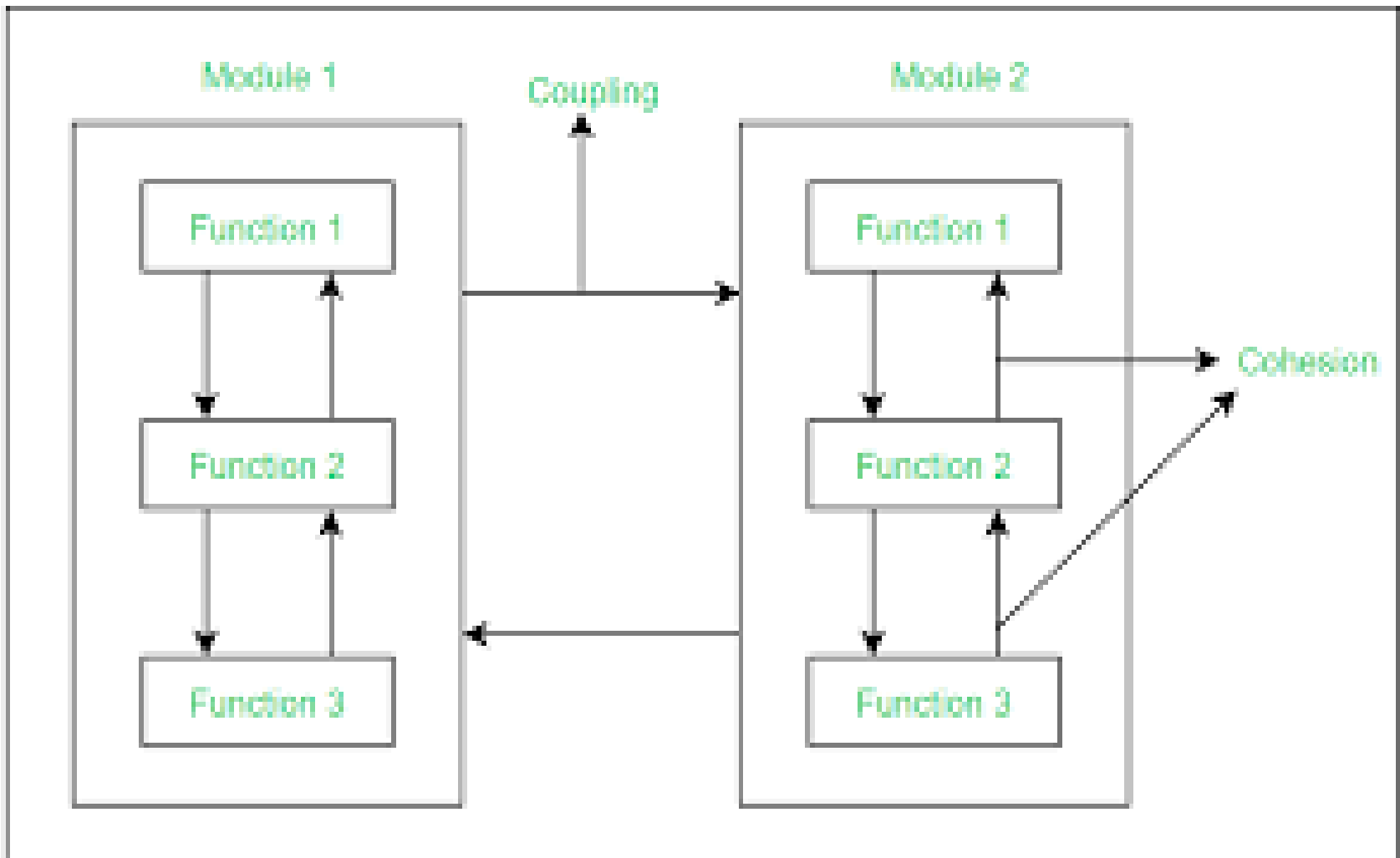
Highly cohesive gives the best software.

Where as loosely coupling gives the best software.

In cohesion, module focuses on the single thing.

In coupling, modules are connected to the other modules.

# Ex:

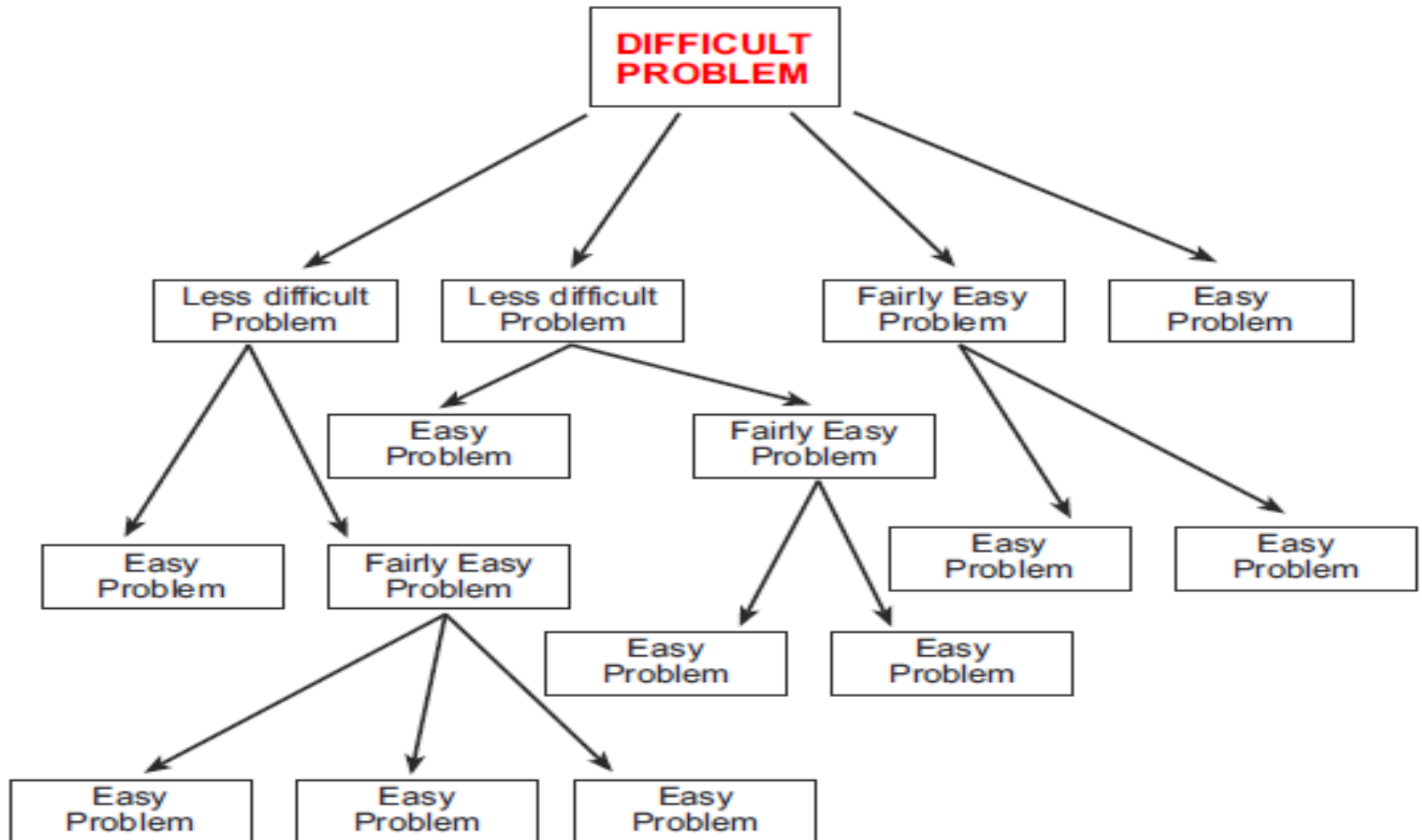




# Cont..

- 7)Stepwise refinement
  - Development of a program by successively refining levels of procedure detail
  - Stepwise refinement is top-down design strategy.
  - Refinement is actually a process of elaboration.
  - Refinement causes the designer to elaborate on the original statement , providing more and more detail as each successive refinement occurs.
  - Abstraction and refinement are complimentary concepts.
  - Refinement helps the designer to reveal low-level details as design progresses
  - Both concepts aid the designer in creating a complete design model as the design evolves .

# Cont..



# Cont..

- 8)Refactoring

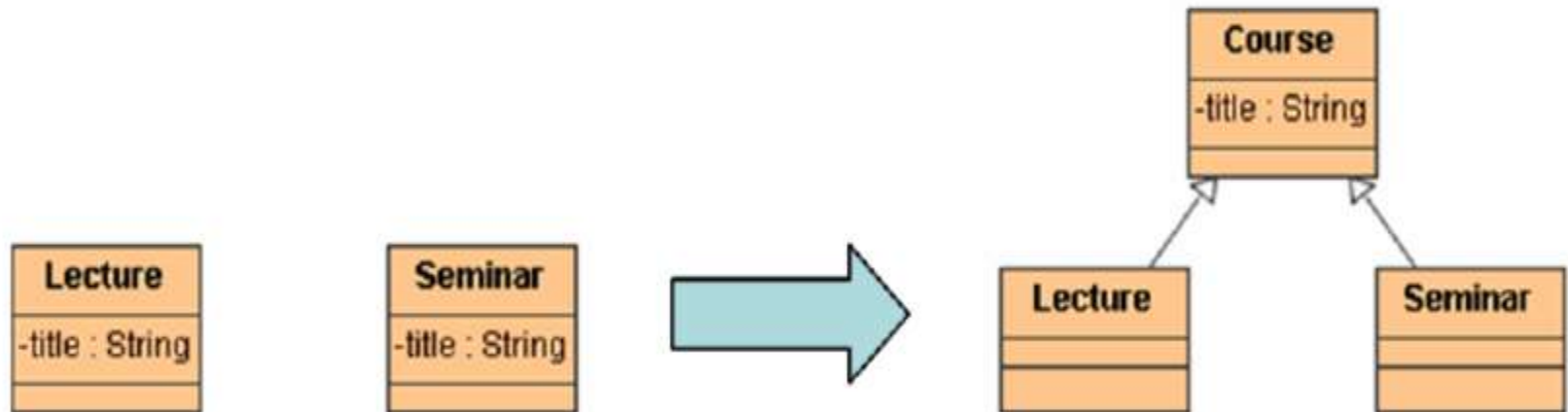
- A reorganization technique that **simplifies the design** (or internal code structure) of a component **without changing its function or external behavior**

- **Advantages:**

*Removes* - redundancy,

- unused design elements,
- inefficient or unnecessary algorithms,
- poorly constructed or
- inappropriate data structures,
- or any other design failures

# Ex for model Refactoring:

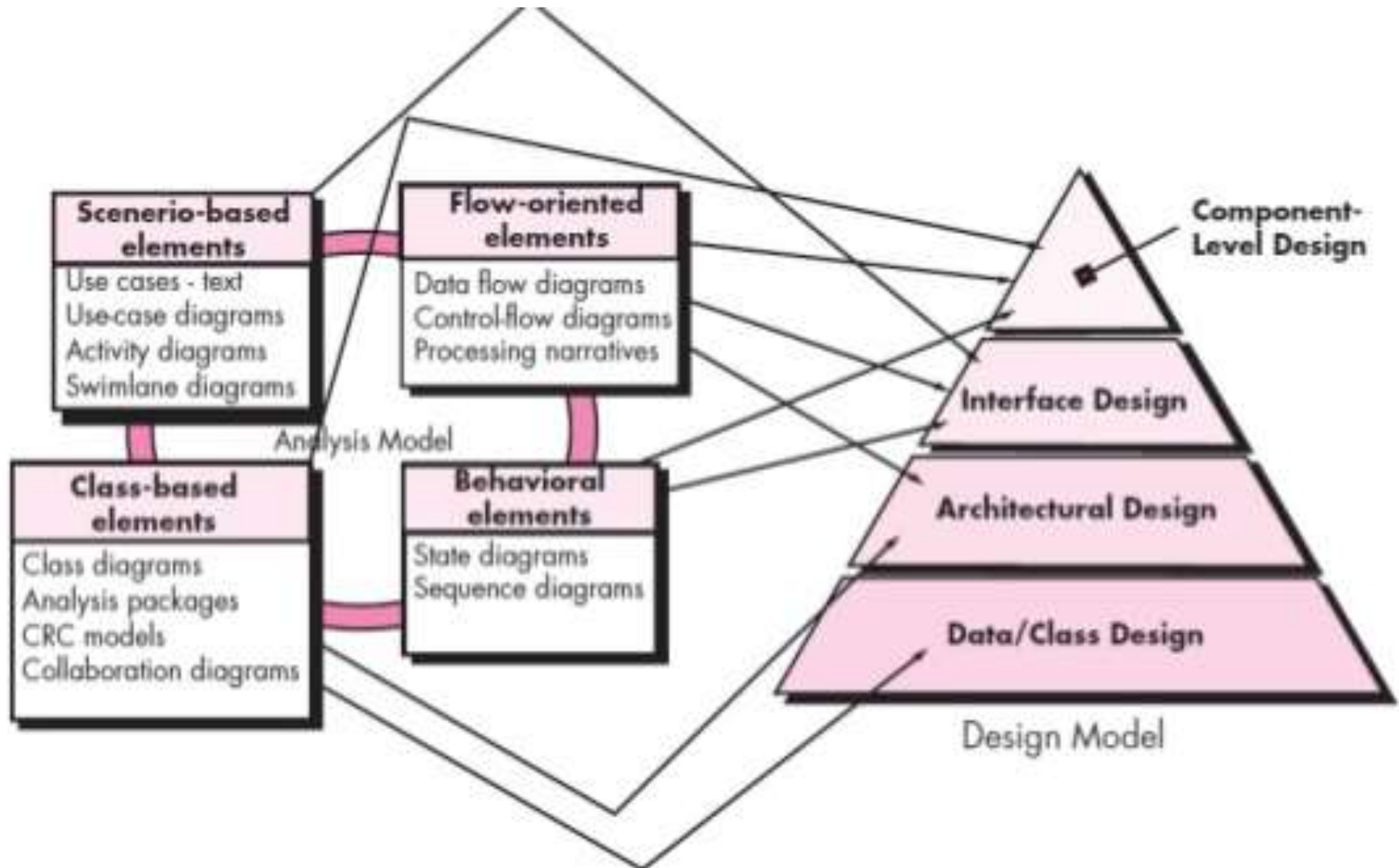


# Cont..

## Examples of refactoring

- Rename an entity
- Encapsulate part of the code as a function
  - opposite: expand a function in a place of call
- Move function into/out of class
- Merge and divide classes
  - factor out a base class, component class

Fig: Translating the Analysis model into the design model



# From Analysis Model to Design Model

- Each element of the **analysis model** provides information that is necessary to create the **four design** models
  - The **data/class design** transforms analysis classes into **design classes** along with the **data structures** required to implement the software
  - The **architectural design** defines the **relationship** between major structural elements of the software; **architectural styles and design patterns** help achieve the requirements defined for the system
  - The **interface design** describes how the software **communicates** with systems that **interoperate** with it and with humans that use it
  - The **component-level design** transforms structural elements of the software architecture into a **procedural description** of software components

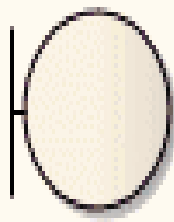
# Design Classes

- **Design Classes** - Detail needed to implement the classes and implement a software infrastructure that support the business solution (i.e., user interface classes, system classes..)
- There are 5 design classes
  - **User interface classes:**
    - These classes are designed for Human Computer Interaction(HCI).
    - These interface classes define all **abstraction** which is required for Human Computer Interaction(HCI).
  - **Business Domain classes:**
    - These classes are commonly refinements of the analysis classes.
    - The class identify the **attributes and services** (methods) that are required to implement the elements of the business domain



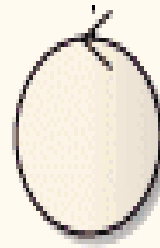
- **Process classes:** Implement lower-level business abstractions required to fully **manage the business** domain classes
- **Persistent classes:** represents **data stores** (ex: data bases) that will persist beyond the execution of the software
- **System classes:** Implement software management and control functions that are enable the **system to operate and communicate** within its computing environment and with the outside world

# Graphical Representations in UML



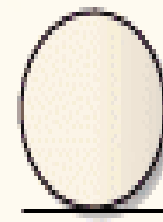
Boundary

A boundary often represents a User Interface (Screen)



Control

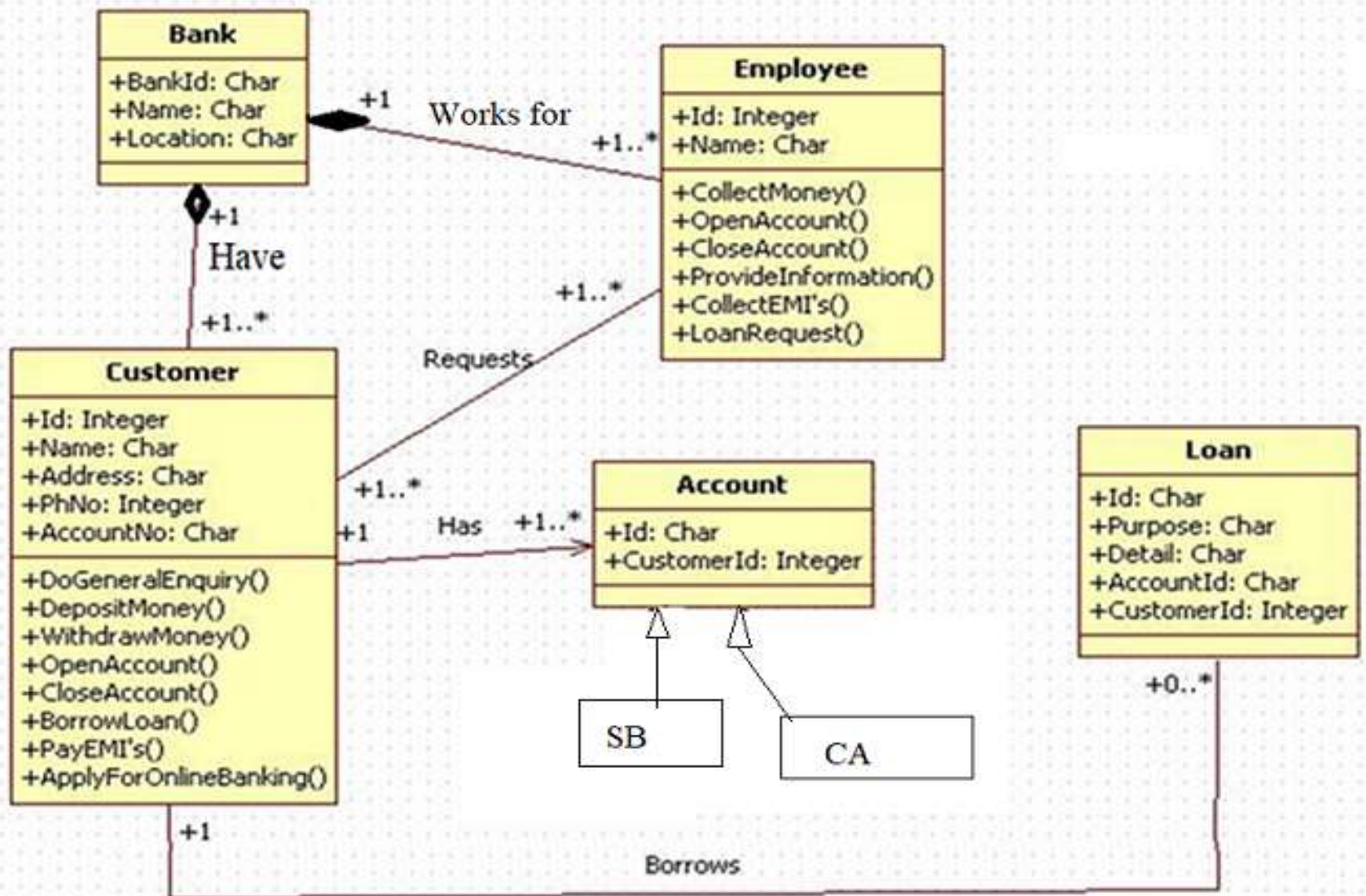
A controller is responsible for implementing business logic between the user interface and the database



Entity

An entity is a persistent (database) object

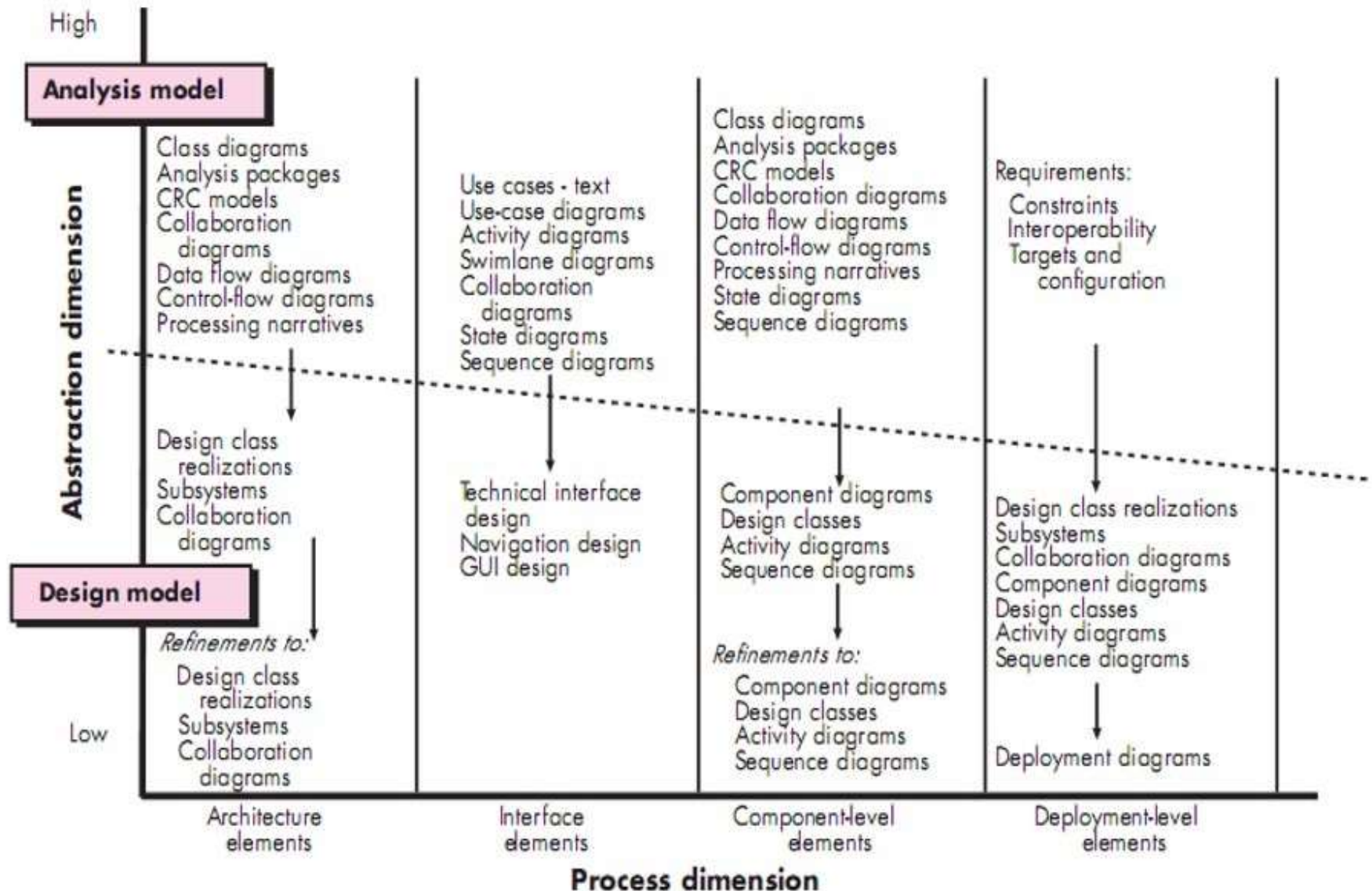
# Ex:



# THE DESIGN MODEL

- The evolution of the design model indicates **design tasks** are executed as a part of the software process
- Design model use many of the **UML diagrams** that were used in the analysis model.
- These diagrams are **refined and elaborated** as a part of design, implementation- specific, architectural style
- components that **reside within** the architecture, and the interface between the components and with the outside world are all emphasized.

## Dimensions of the design model



- The design model can be viewed in two different dimensions
  - (Horizontally) The process dimension indicates the evolution of the parts of the design model as each design task is executed
  - (Vertically) The abstraction dimension represents the level of detail as each element of the analysis model is transformed into the design model and then iteratively refined
- Design model elements are **not always** developed in a **sequential** fashion
- In most cases preliminary **architectural design** sets the stage and is followed by **interface design** and **component-level design**, which often occur in parallel.
- The deployment model usually **delayed until** the design has been fully developed.

# Cont.

## Design Model Elements

- Data Design Elements
  - Data model --> data structures
  - Data model --> database architecture
- Architectural Design Elements (*In detail later on*)
  - Application domain
  - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
  - Patterns and "styles"
- Interface elements
  - the user interface (UI)
  - external interfaces to other systems, devices, networks or other producers or consumers of information
  - internal interfaces between various design components.
- Component Level Design Elements
- Deployment Level Design Elements



# Data design elements

- It creates a model of data and/or information that is represented at a **high level of abstraction**.
- This data model is then **refined** into more implementation-specific and can be processed by the computer-based system.
- The **structure of data** has always been an important part of software design



- At **component level**, the data design considers the data structures that are required to **implement local data objects**
- At the **application level**, the data design focuses on **files or data bases**
- At the **business level**, the collection of information stored in disparate databases and reorganized into a “**data warehouse**” enables data mining or knowledge discovery that can have an impact on the success of the business itself.

- The *architectural design for software is the equivalent to the floor plan of a house(room ,size,shape...)*

*The architectural model is derived from three sources*

- Information about the **application domain** for the software to be built
- Specific **analysis model elements** such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand
- The availability of **architectural patterns**

## Architectural design elements

## Interface design elements:

- The interface design for software is the equivalent to a **set of detailed drawings** for the doors, windows, and external utilities of a house.
- The interface design elements for software tell how **information flows into and out** of the system
- How it is **communicated** among the components defined as part of the architecture.

# There are 3 important elements of interface design

- The **user interface(UI)** Connectivity between components in the system is UI
- **External interfaces** to other systems, devices, networks, or other produces or consumers of information
- **Internal interfaces** between various design components
- These interface design elements allow the software to communicated externally and enable internal communication and collaboration among the components that populate the software architecture.

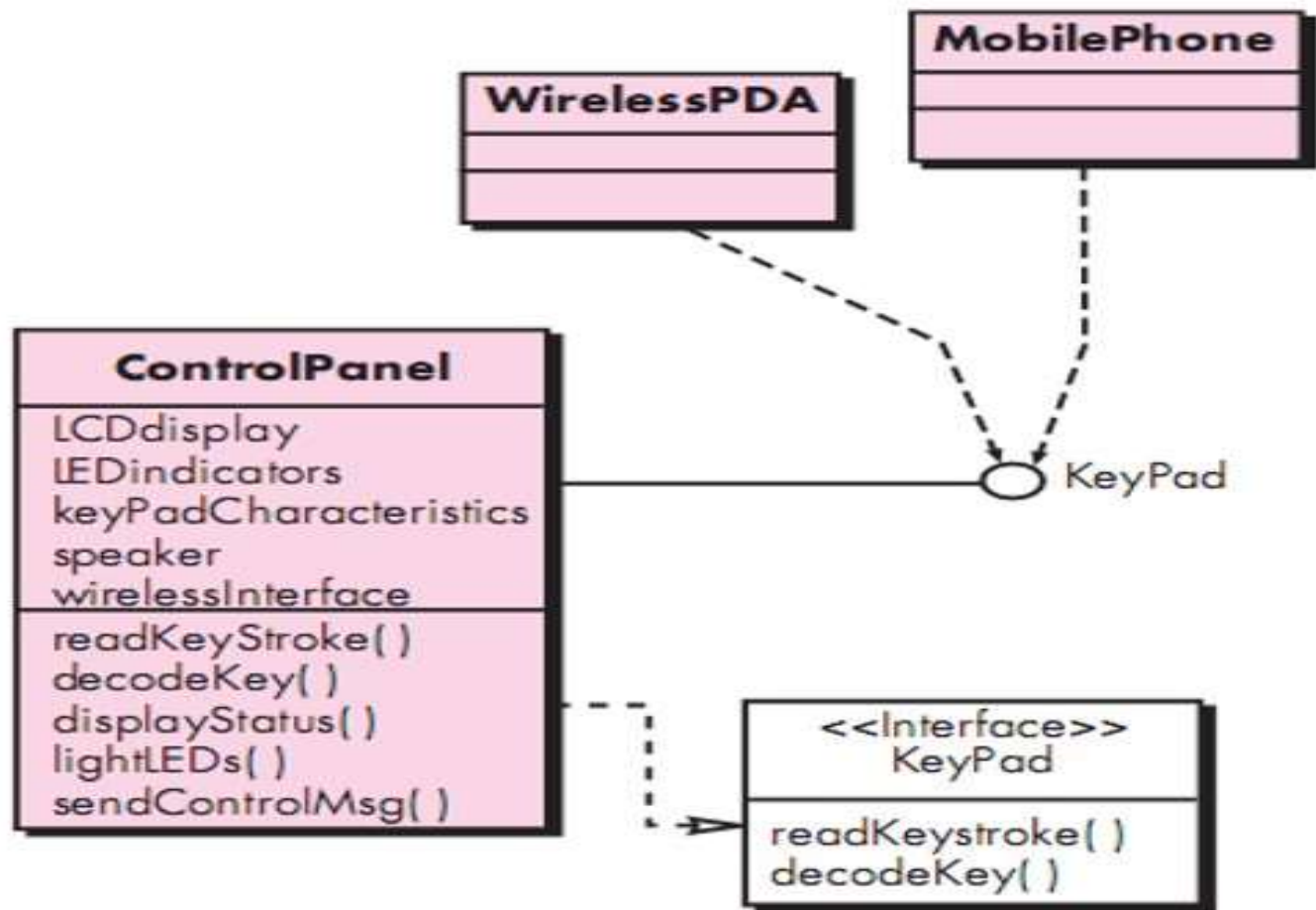
## UI design is a major software engineering action

- The design of a UI incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms..)
- Ergonomic elements (e.g., information layout and placement, UI navigation...)
- Technical elements (e.g., UI patterns, reusable components).
- In general, the UI is a unique subsystem within the overall application architecture.
- Ex:-Control panel function via wireless PDA or mobile phone

# Ex:GUI



# Interface representation of control panel

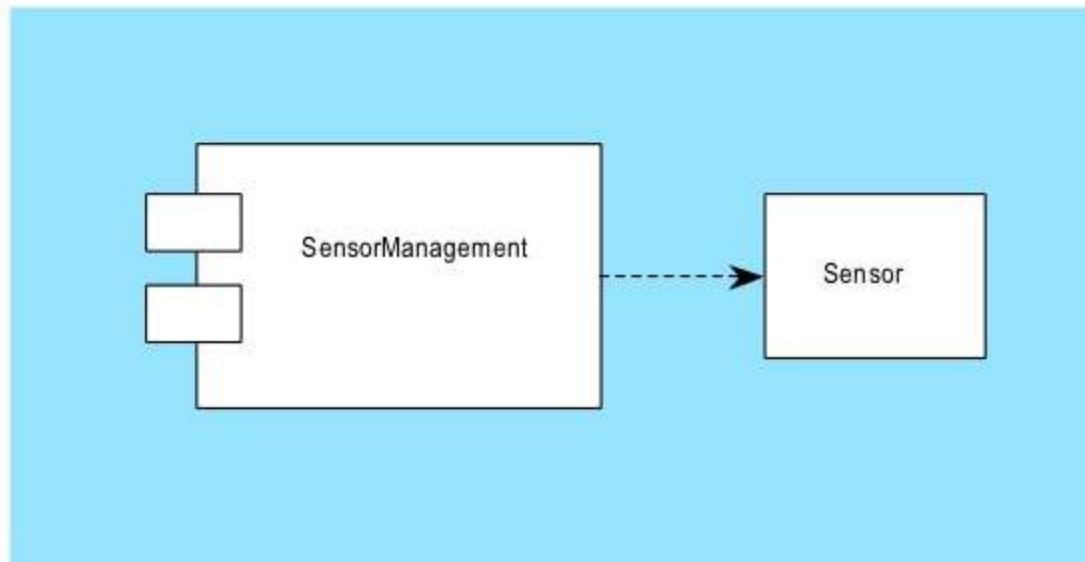


# Component- level design elements

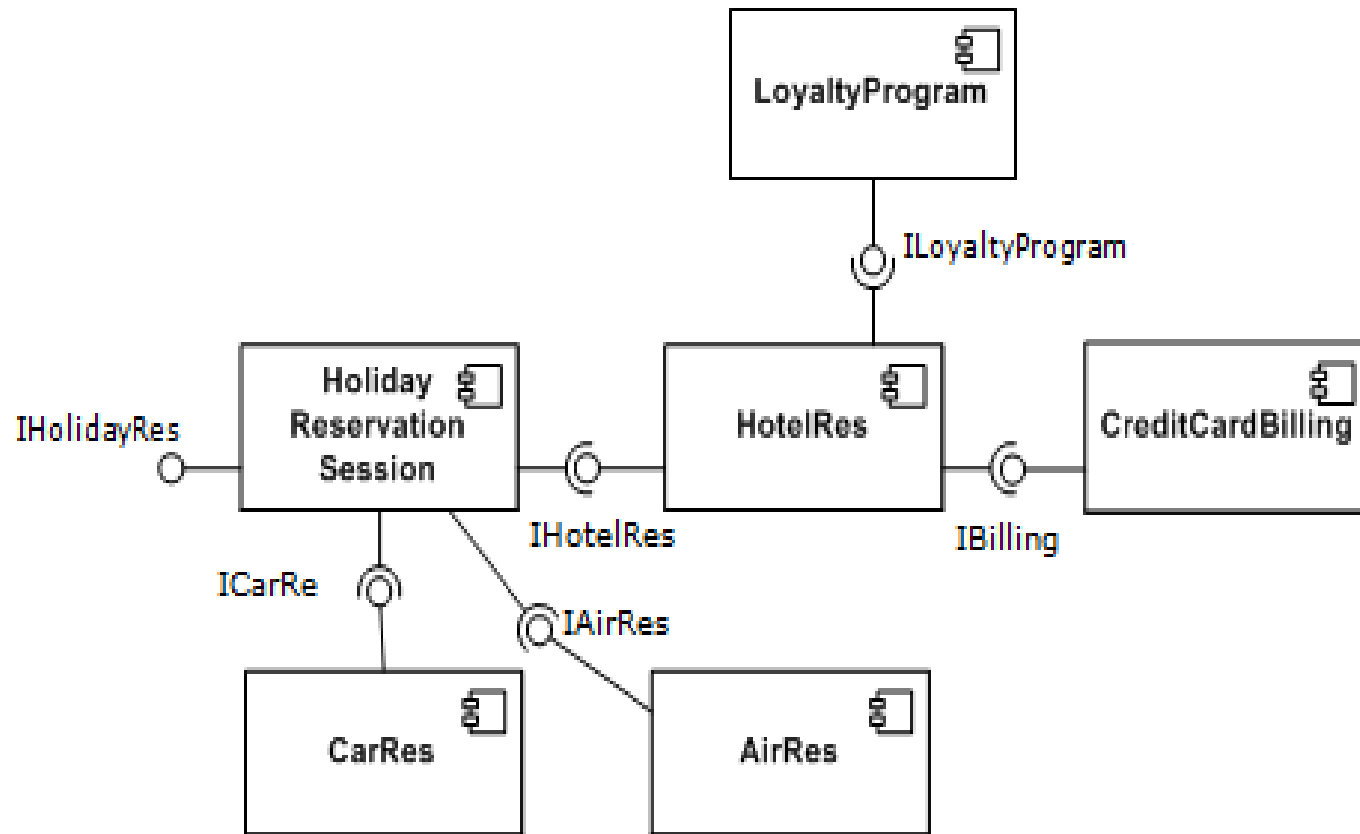
- The component-level design for software is equivalent to a set of **detailed drawings**.
- The component-level design for software fully describes the **internal detail** of each software component.
- It defines the **data structures, algorithms**, interface characteristics, and communication mechanisms allocated to each **component** for the system development.



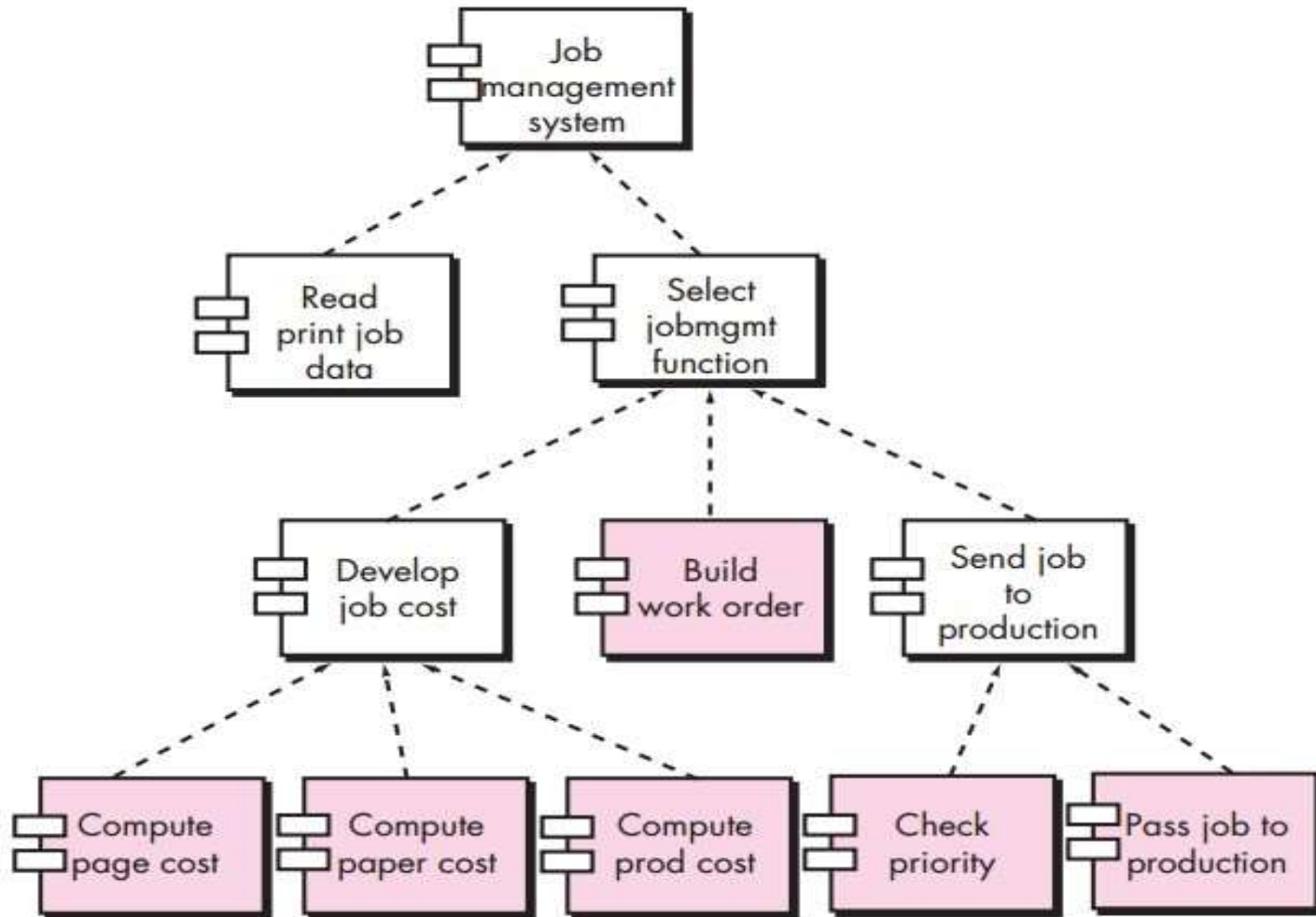
## Component Level Design Elements



## Ex2:

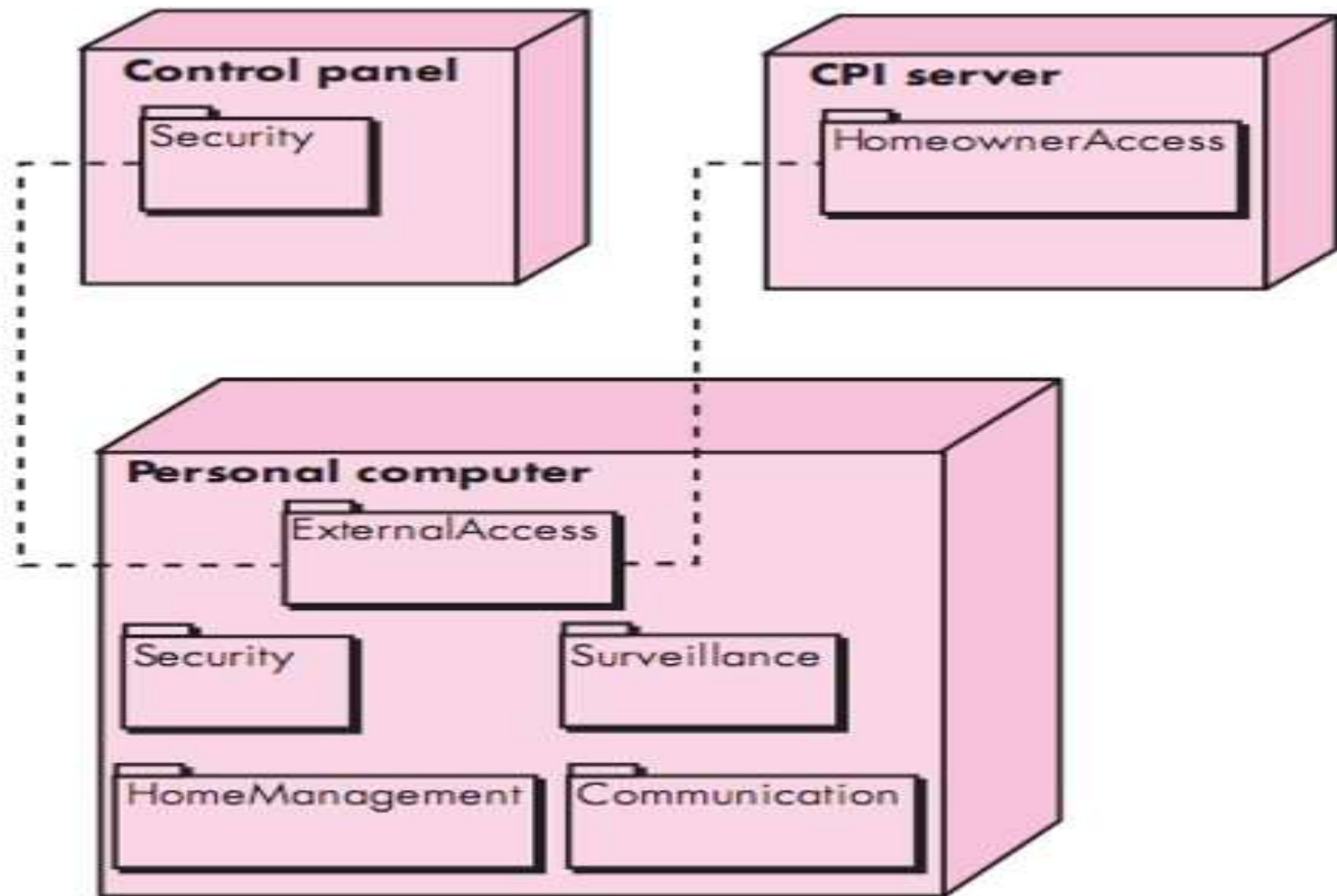


## Ex 3.



# Deployment-level design elements

- Deployment-level design elements indicated how software **functionality and subsystems** will be allocated within the physical computing environment that will support the software
- The **Deployment-level Design** creates a model that shows the physical architecture of the hardware and **software** of the system
- **Deployment** diagrams are used to visualize the **physical hardware** on which the software system **will execute**.



# Ex:Deployment diagram for ATM

