Unit II- SOFTWARE DESIGN

SOFTWARE DESIGN

- Abstraction, Modularity, Cohesion & Coupling,
- Scenario based modeling : it identifies the primary use cases for the proposed software system or application, to which later stages of requirements **modeling** will refer.
- SSAD (ER diagram, Control Flow Diagram CFD , Data Flow Diagram DFD),
- OOAD (Unified Modeling Language UML)- Modeling Language (UML)- Introduction to all Diagrams.

Software Design

- Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- Software design usually involves problem solving and planning a software solution. This includes both a low-level component and algorithm design and a high-level, architecture design.
- It’s the next step after RE

Software Design Levels

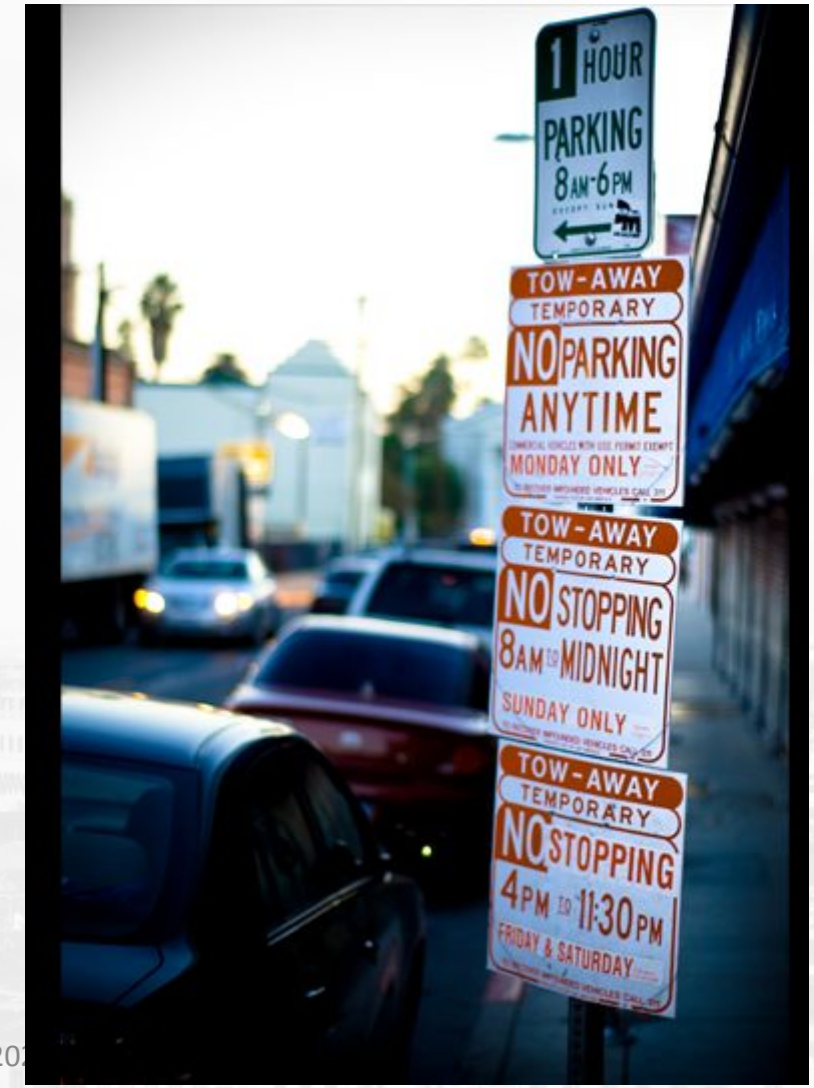
- Software design yields **three** levels of results:

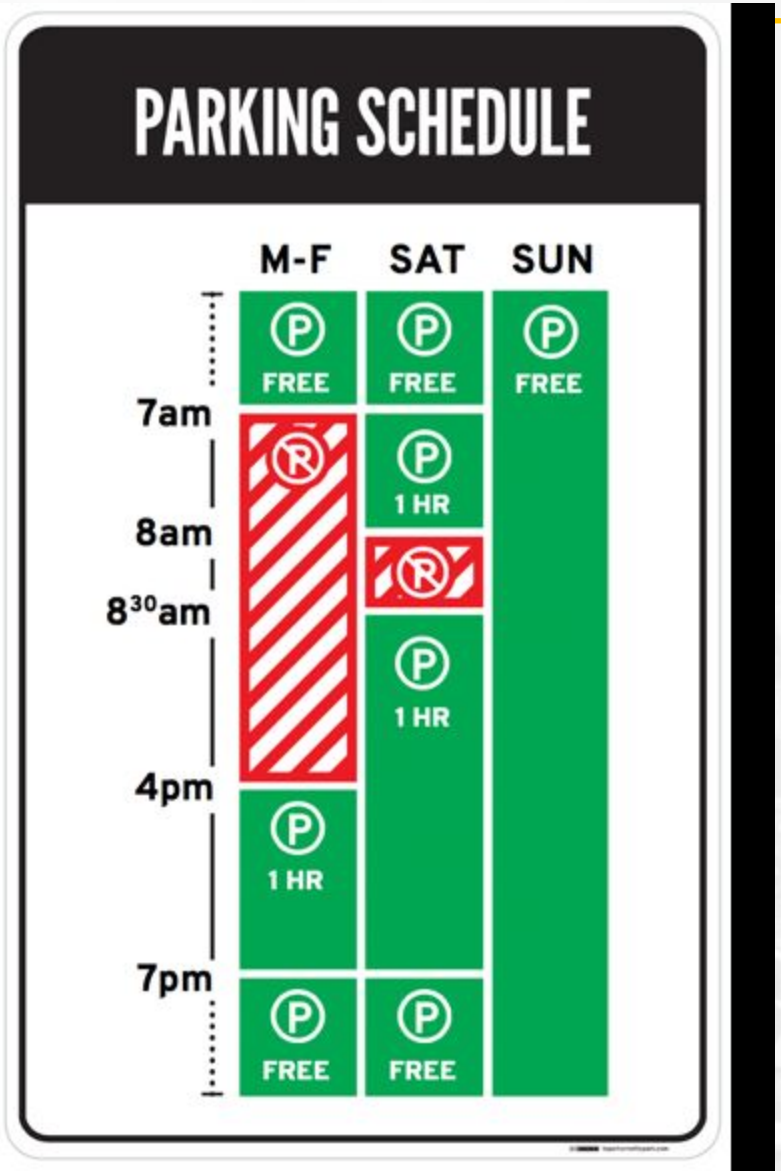
1. **Architectural Design -**
   i. is the highest abstract version of the system and identifies the software as a system with many components interacting with each other.
   ii. "the process of defining a collection of **hardware and software components** and their interfaces to establish the framework for the development of a computer system."
2. **High-level Design-**
   i. Breaks the Architectural Design **into less-abstracted view of sub-systems and modules** and depicts their **interaction** with each other.
3. **Detailed Design-**
   i. deals with the **implementation** part of what is seen as a system and its sub-systems
   ii. It defines **logical structure** of each module and their interfaces to communicate with other modules.

Software Design Levels

1. **Architectural Design**
2. **High-level Design-**
3. **Detailed Design-**

Good design and Bad Design





Good Design vs Bad Design

Good Design	Bad Design
1. Meets all technical requirements	1. Meets only some technical requirements
2. Works all the time	2. Works initially but stops working after a short time
3. Meets cost requirements	3. Costs more than it should
4. Requires little or no maintenance	4. Requires frequent maintenance
5. Is safe	5. Poses a hazard to users
6. Creates no ethical dilemma	6. Raises ethical questions

What is a good design?

- A **good design** is focused.
- A **good design** is effective and efficient in fulfilling its purpose.
- It relies on as few external factors and inputs as possible, and these are easy to measure and manipulate to achieve an expected other output.
- A **good design** is always the simplest possible working solution.

Abstraction

- The principle of abstraction requires:
  - lower-level modules do not invoke functions of higher level modules.
  - Also known as layered design.
- High Level Design: High-level design maps functions into modules such that:
  - Each module has high cohesion
  - Coupling among modules is as low as possible
  - Modules are organized in a neat hierarchy

Modularity

- Modularity is a fundamental attributes of any good design.
  - Decomposition of a problem cleanly into modules:
  - Modules are almost independent of each other
  - Divide and conquer principle.
- If modules are independent:
  - Modules can be understood separately,
  - Reduces the complexity greatly.
  - To understand why this is so,
  - *Remember that it is very difficult to break a bunch of sticks but very easy to break the sticks individually.*

Modularity

- A module ideally should have **high cohesion** and **low coupling**:
- **Functionally independent** of other modules to have minimal interaction with other modules.
- Improves understandability and good design:
- Complexity of design is reduced as different modules are understood in isolation:
- A functionally independent module:
  - Can be easily taken out and reused in a different program.
  - Each module does some well-defined and precise function
  - The interfaces of a module with other modules is simple and minimal.

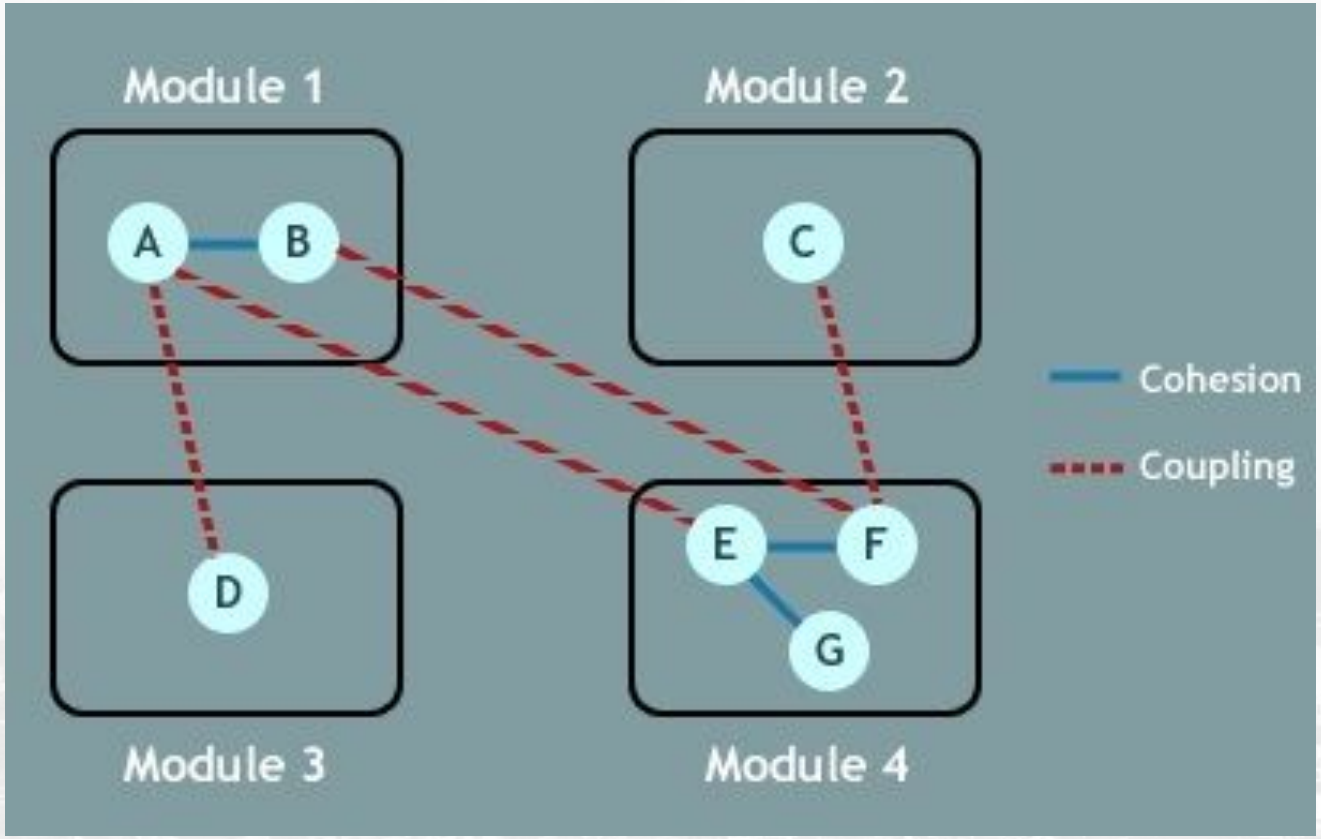
Modularity

- In technical terms, modules should display:
  - high cohesion
  - low coupling.
- Cohesion is a measure of:
  - Functional strength of a module.
  - A cohesive module performs a single task or function.
- Coupling between two modules:
  - A measure of the degree of interdependence or interaction between the two modules.

Cohesion and 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.

Cohesion and Coupling



Ideally good Software Design will promote Tight cohesion and Low coupling
- Changes in the system

Transition from Analysis to Design

- **Analysis:** A process of extracting and organizing **user requirements** and establishing an **accurate model of the problem domain**.(WHAT)

- **Design:** Process of mapping requirements to a **system implementation** that conforms to desired cost, performance, and quality parameters.(HOW)

Transition from Analysis to Design

- Blurred line between analysis & design
- Process of design leads to better understanding of requirements
- Can be performed iteratively
- No direct & obvious mapping exists between structured analysis and structured design.

Two Approaches for Analysis & Design

- Traditional Approach: SSAD(System Structured Analysis and Design)
  - Structured Analysis (SA), resulting in a logical design, drawn as a set of **data flow diagrams**
  - Structured Design (SD) transforming the logical design into a program structure drawn as a set of **structure charts**

- OOAD(Object Oriented Analysis and Design)
  - Designing systems using self-contained objects and object classes

Structured Analysis

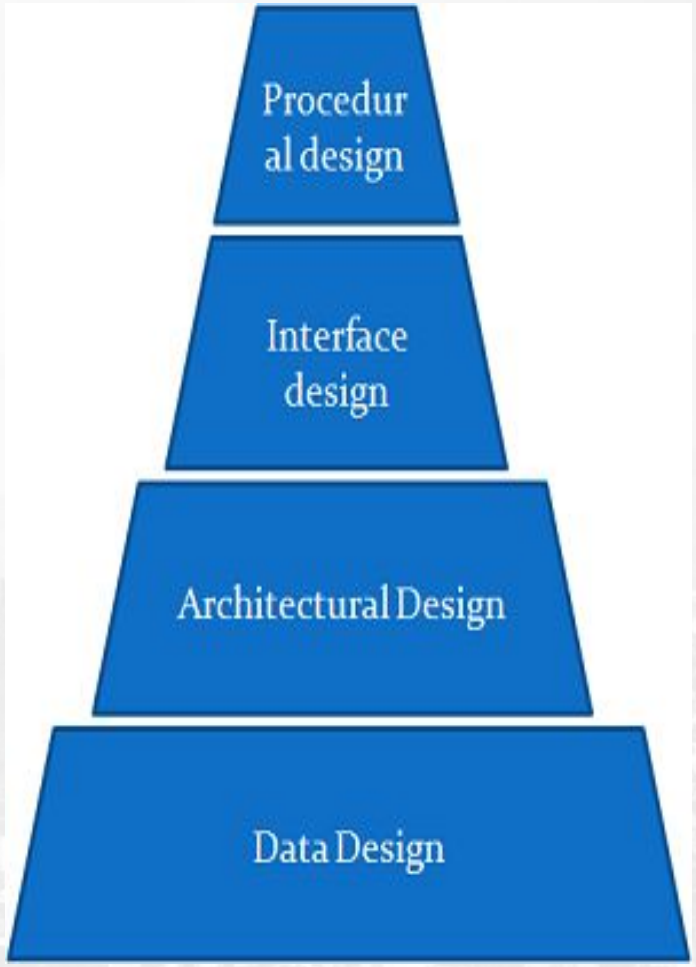
- Is a development method to understand the system and its activities in a logical way.
- It is a systematic approach that analyze and refine objectives of an system
- It has following attributes −
  - It is graphic which specifies the presentation of application.
  - It divides the processes so that it gives a clear picture of system flow.
  - It is logical rather than physical
  - It is an approach that works from high-level overviews to lower-level details.
- Structured Analysis Tools and techniques used for system development:
  - **Data Flow Diagrams**
  - **Entity Relation Diagram**
  - Data Dictionary
  - Decision Trees
  - Structured English
  - Pseudocode

Structured Analysis

- Define what system needs to do (processing requirements)
- Define data system needs to store and use (data requirements)
- Define inputs and outputs
- Define how functions work together to accomplish tasks
- Data flow diagrams and entity relationship diagrams show results of structured analysis

Structured Analysis

- **Data design** transforms ERD (Entity Relationship Diagram) in to data structures
- **Architectural design** - higher-level structure of the system that depicts the relationship between the modules of the system
- **Interface design** defines interaction of people with the system
- **Procedural design** transforms the modules of the system architecture in to Pseudo-code or Flow Chart



Control Flow Diagrams

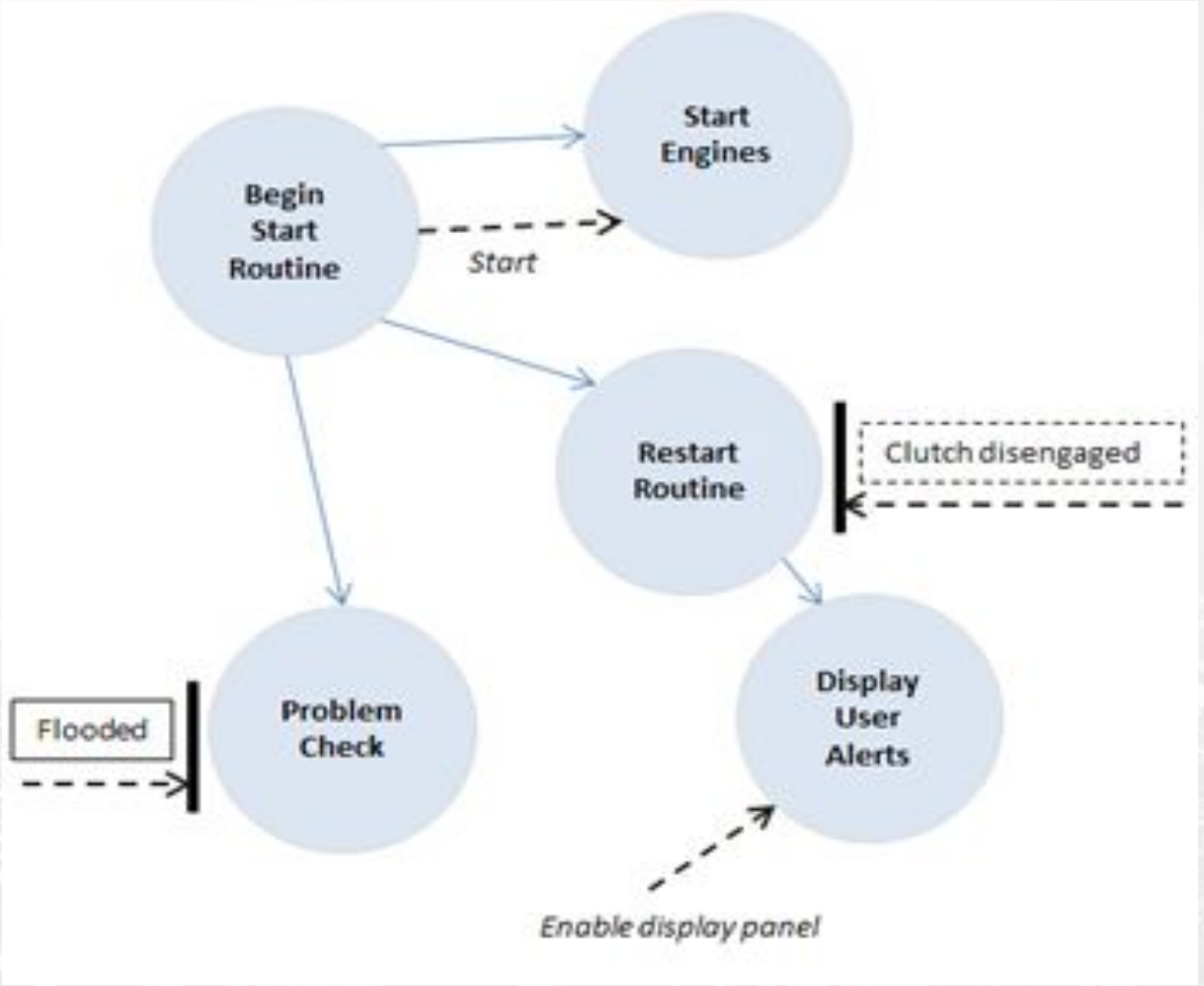
- A control flow diagram helps to understand the detail of a process.
- It shows us where control starts and ends and where it may branch off in another direction, given certain situations.
- *Let's say you are working on software to start a machine. What happens if the engine is flooded, or a spark plug is broken? Control then changes the flow to other parts of the software.*
- We can represent these branches with a diagram. The flow diagram is helpful because it can be understood by both stakeholders and systems professionals.
- Although some of the symbols might not be fully understood by the layperson, they can still grasp the general concept.

Control Flow Diagrams

- Even to the non-IT person, it is fairly clear that the software will attempt to start engines.
- If errors occur, then it will branch off into different directions.
- The general flow should make sense
  - Try to start the engines
  - Run a problem check
  - Display alerts if it can't start
  - Disengage the clutch
  - Try restarting
  - But what about the vertical lines and dashed arrows?

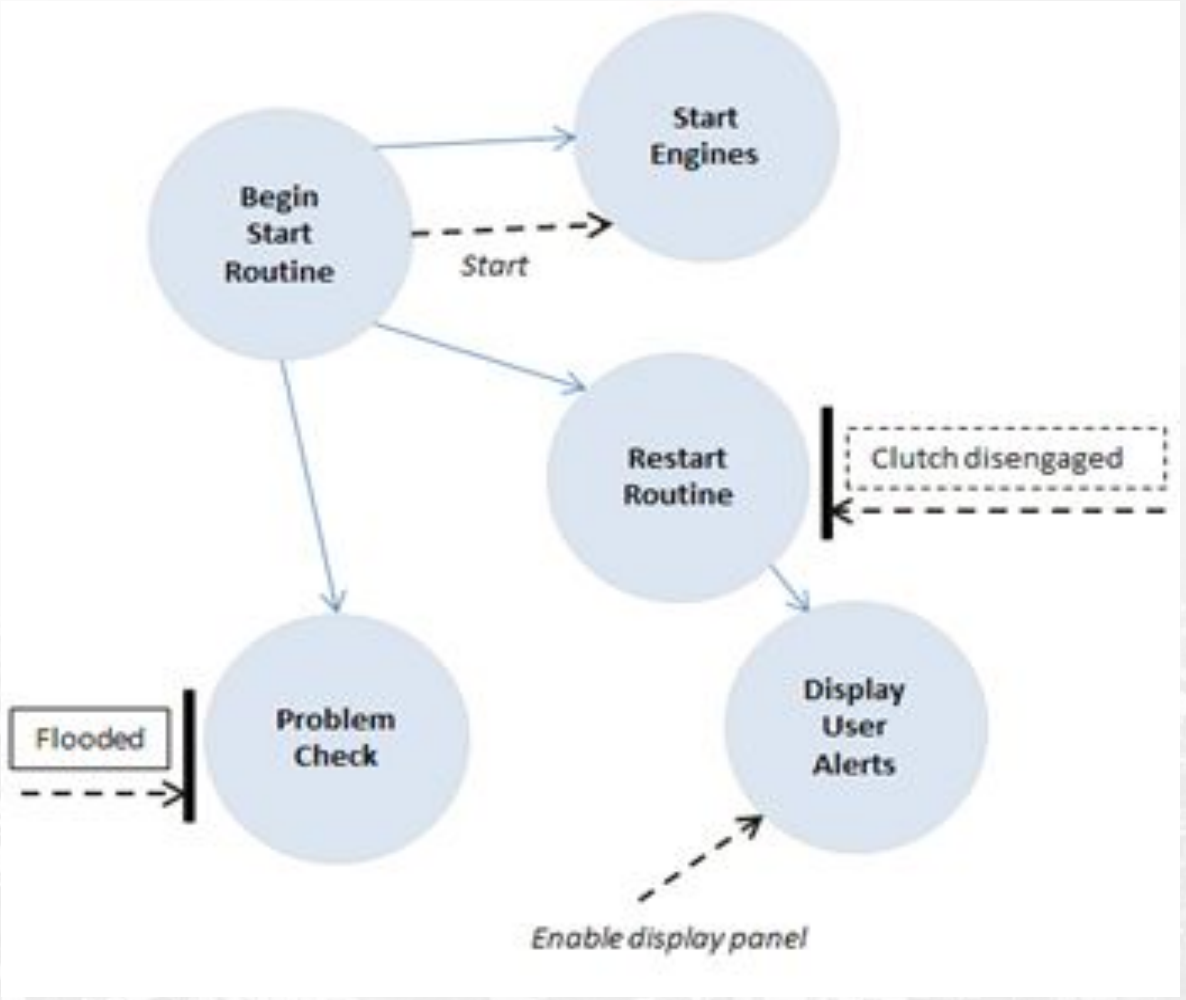
Control Flow Diagram

- Keeping with the idea of an engine-start program, the following shows flow through the software:
- **Vertical Bar**
  - The vertical bar indicates input or output from the current control specification.
  - Think of the bar as a window INTO or OUT OF the entire specification/flow. Additional notes, including an arrow, indicate further detail of the flow through that window.



Control Flow Diagram

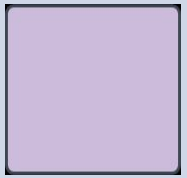
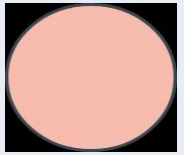
- **Dashed Arrow**
  - The input to the problem check step is *Flooded*
  - The input into the restart routine is *clutch disengaged*



Data Flow Diagrams

- The DFD (also known as a bubble chart) is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among these functions.
- Each function is considered as a processing station (or process) that consumes some input data and produces some output data.
- **The system is represented in terms of the input data to the system, various processing carried out on these data, and the output data generated by the system.**

DFD Notations

S.No	Name	Description	Notation Yourdon and Coad	Gane and Sarson
1	**External Entity**	An outside system that sends or receives data, communicating with the system being diagrammed. They are the sources and destinations of information entering or leaving the system eg. an outside organization or person, a computer system or a business system. They are also known as terminators, sources and sinks or actors. They are typically drawn on the edges of the diagram.		
2	**Process**	Any process that changes the data, producing an output. It might perform computations, or sort data based on logic, or direct the data flow based on business rules		
3	**Data store**	Files or repositories that hold information for later use, such as a database table or a membership form.		
4	**Data flow**	The route that data takes between the external entities, processes and data stores.		

DFD Rules and Tips

- Each process should have at least one input and an output.
- Each data store should have at least one data flow in and one data flow out.
- Data stored in a system must go through a process.
- All processes in a DFD go to another process or a data store.

DFD

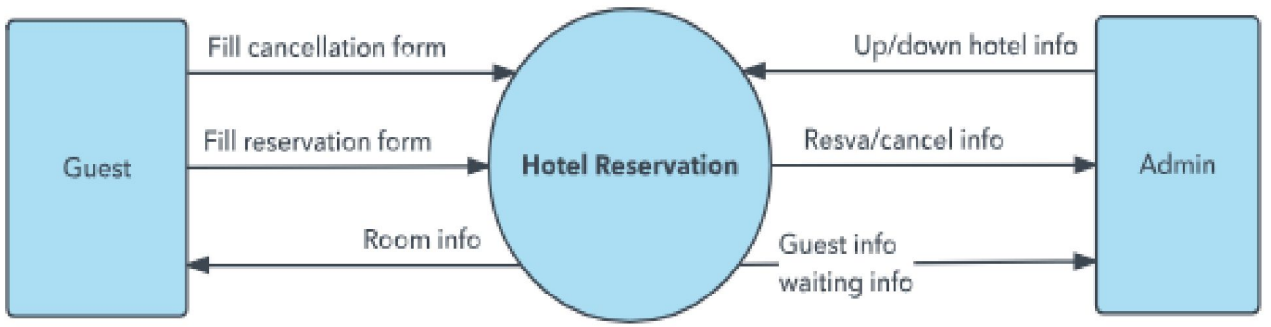
- A data flow diagram can dive into progressively more detail by using levels and layers, zeroing in on a particular piece. DFD levels are numbered 0, 1 or 2, and occasionally go to even Level 3 or beyond.
  - Level 0
  - Level 1
  - Level 2

Data-processing models

- Data flow diagrams are used to model the system’s data processing
- These show the processing steps as data flows through a system
- Intrinsic part of many analysis methods
- Simple and intuitive notation that customers can understand
- Show end-to-end processing of data

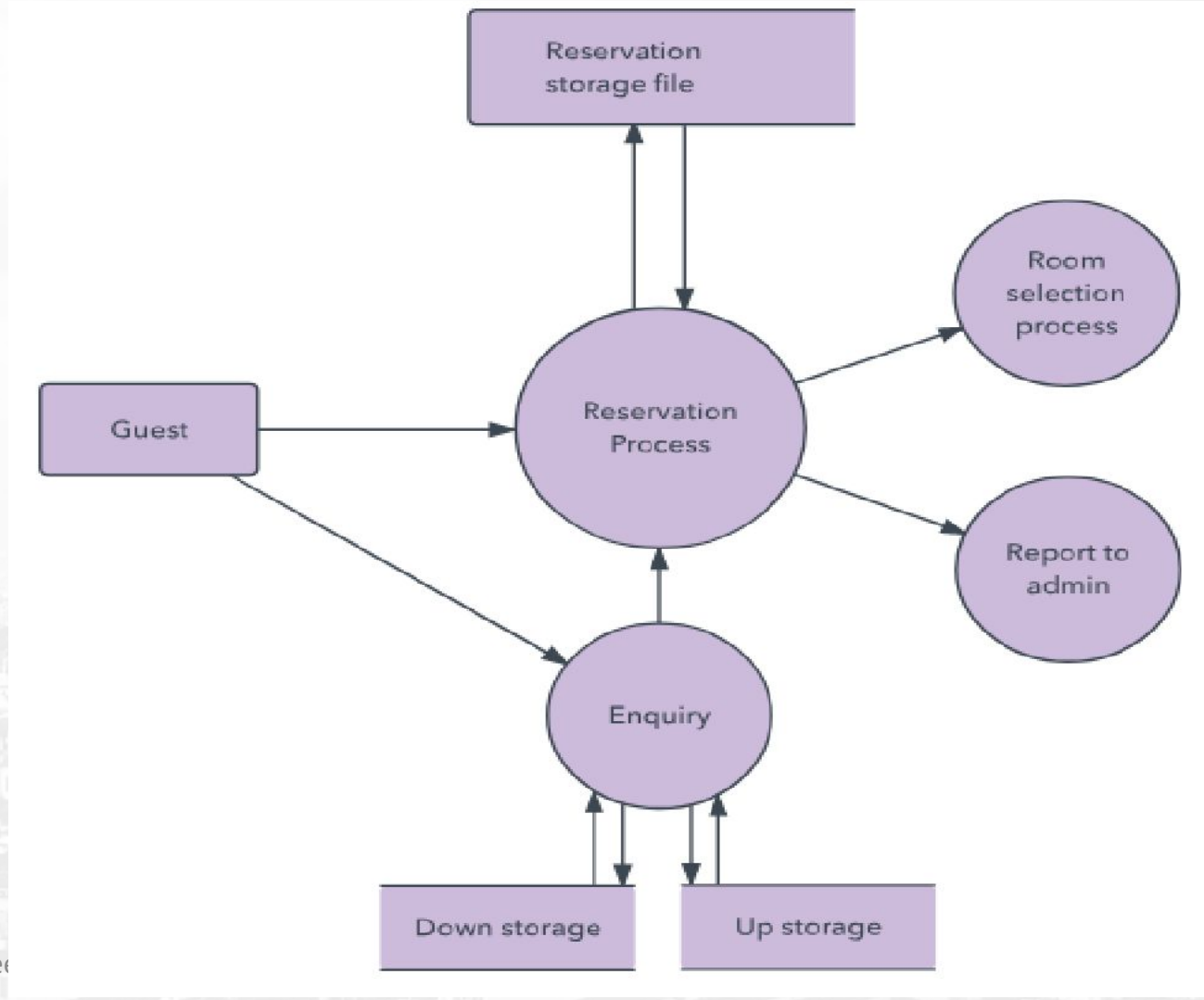
DFD- Level 0

- DFD Level 0 is also called a Context Diagram.
- It's a basic overview of the whole system or process being analyzed or modeled.
- It's designed to be an at-a-glance view, showing the system as a single high-level process, with its relationship to external entities.
- It should be easily understood by a wide audience, including stakeholders, business analysts, data analysts and developers.



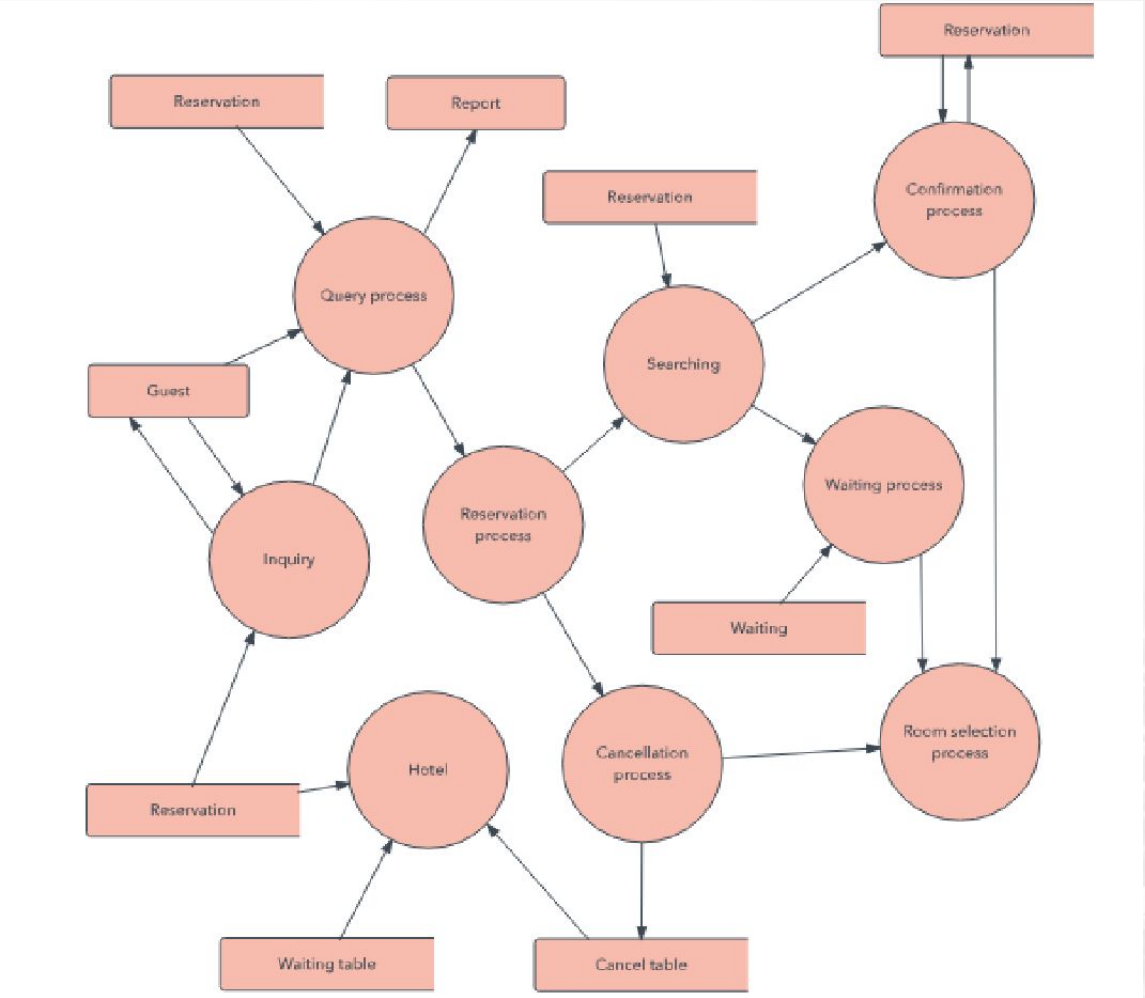
DFD- Level 1

- DFD Level 1 provides a more detailed breakout of pieces of the Context Level Diagram.
- You will highlight the main functions carried out by the system, as you break down the high-level process of the Context Diagram into its subprocesses.



DFD- Level 2

- DFD Level 2 then goes one step deeper into parts of Level 1. It may require more text to reach the necessary level of detail about the system's functioning.

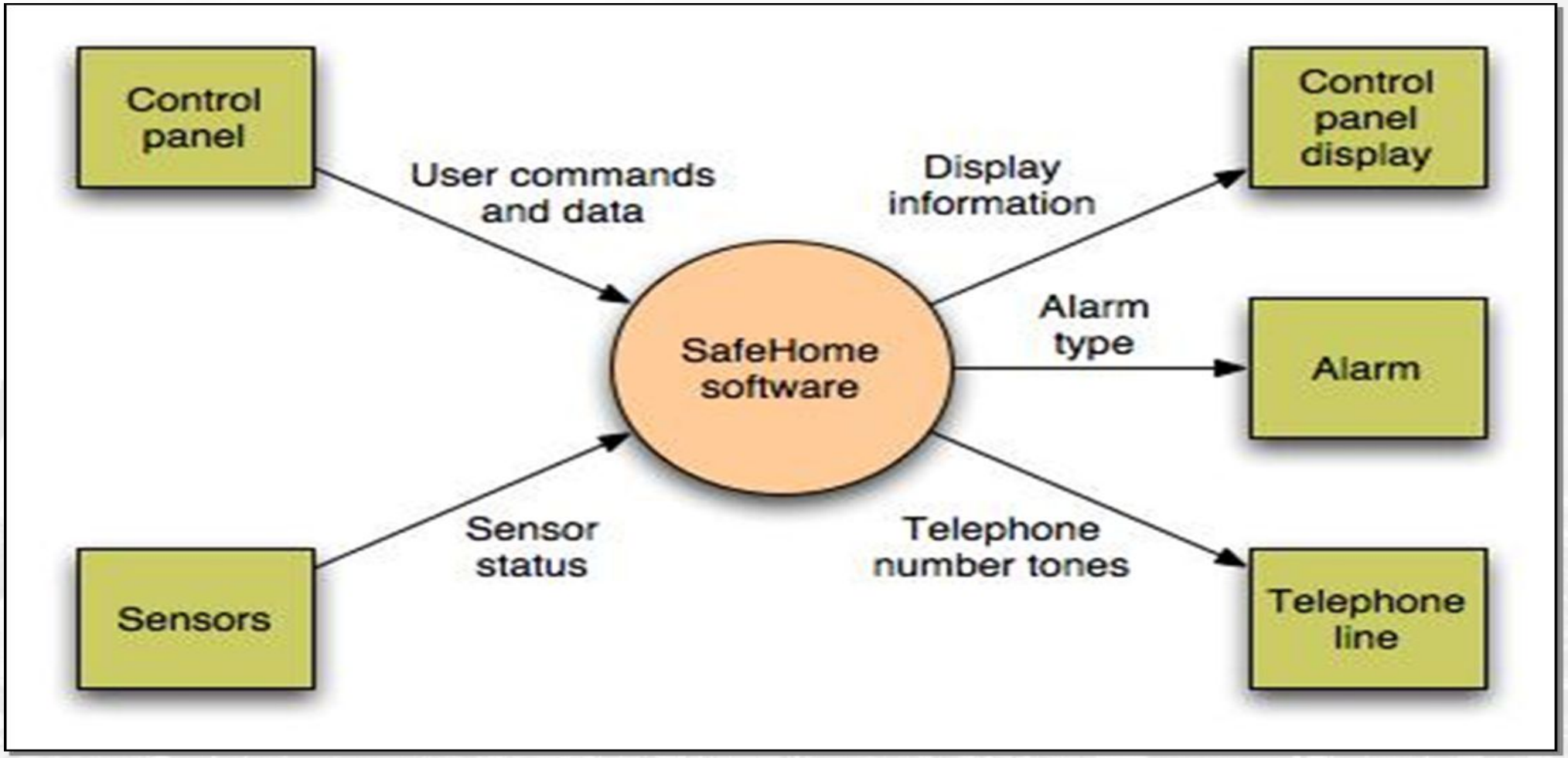


Creating Data Flow Diagrams

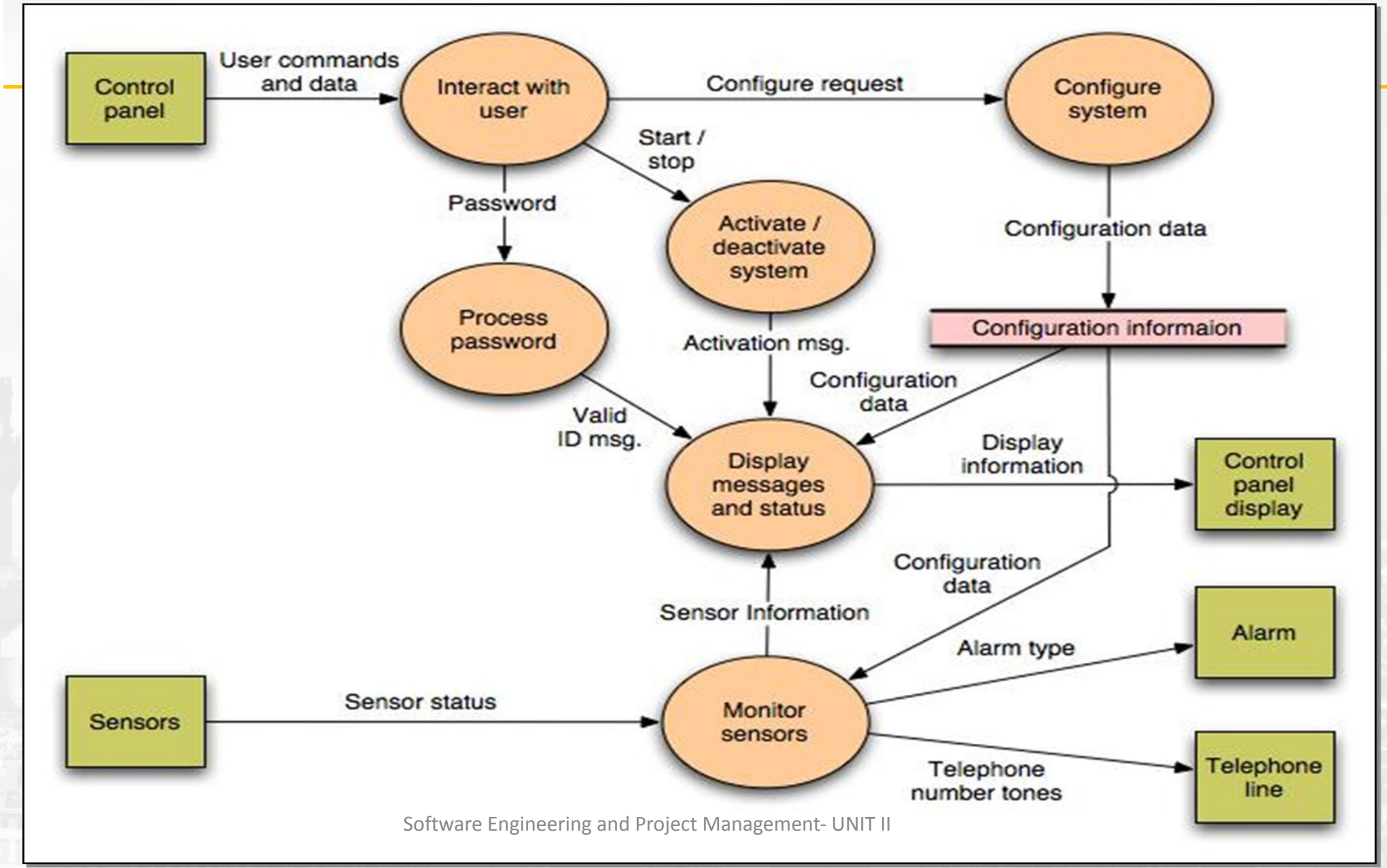
Steps:

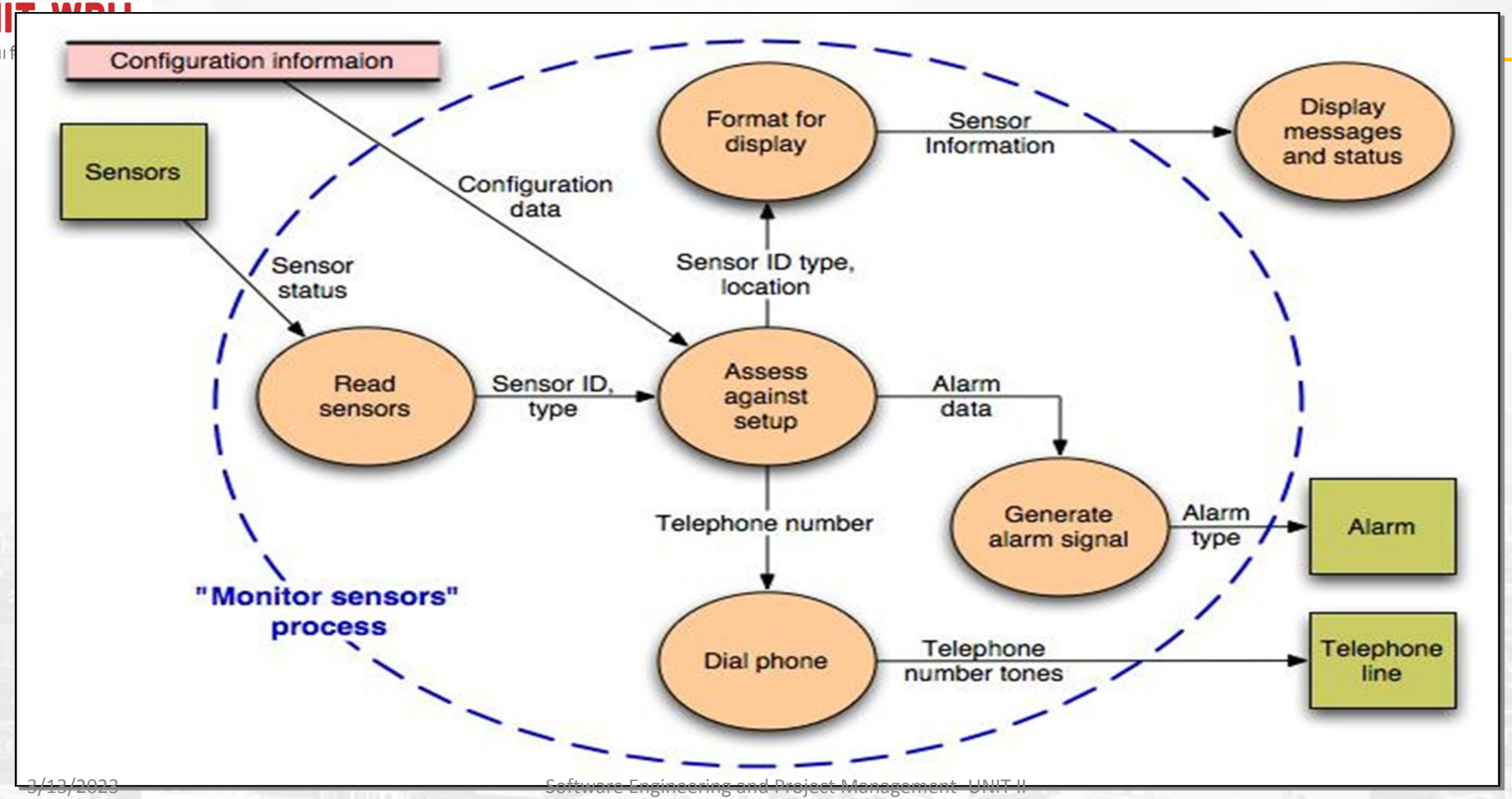
1. Create a list of activities
2. Construct Context Level DFD
   (identifies external entities and processes)
3. Construct Level 0 DFD
   (identifies manageable sub process)
4. Construct Level 1- n DFD
   (identifies actual data flows and data stores)
5. Check against rules of DFD

- The ***SafeHome*** **security function** enables the **homeowner** to configure the **security system** when it is installed, monitors all **sensors** connected to the security system, and interacts with the homeowner through the **Internet**, a **PC**, or a **control panel**.
- During **installation**, the *SafeHome* PC is used to program and configure the system. Each sensor is assigned a **number** and **type**, a **master password** is programmed for arming and disarming the system, and **telephone number(s)** are input for dialing when a **sensor event** occurs.
- When a sensor event is recognized, the software invokes an audible **alarm** attached to the system. After a **delay time** that is specified by the homeowner during system configuration activities, the software *dials* a telephone number of a **monitoring service**, provides information about the **location**, reporting the nature of the event that has been detected. The telephone number will be redialed every 20 seconds until a **telephone connection** is obtained.
- The homeowner receives **security information** via a control panel, the PC, or a browser, collectively called an **interface**. The interface displays prompting **messages** and system **status information** on the control panel, the PC, or the browser window. Homeowner interaction takes the following form...



Context-level DFD for *SafeHome* security function


Control panel
User commands and data
Interact with user
Configure request
Configure system
Start / stop
Password
Activate / deactivate system
Configuration data
Process password
Configuration informaion
Activation msg.
Configuration data
Valid ID msg.
Display messages and status
Display information
Control panel display
Configuration data
Sensor Information
Alarm
Alarm type
Sensors
Sensor status
Monitor sensors
Telephone number tones
Telephone line


Configuration informaion
Sensors
Format for display
Sensor Information
Display messages and status
Configuration data
Sensor status
Sensor ID type, location
Read sensors
Sensor ID, type
Assess against setup
Alarm data
Generate alarm signal
Alarm type
Alarm
Telephone number
"Monitor sensors" process
Dial phone
Telephone number tones
Telephone line

Example 2 : Lemonade Stand

Creating Data Flow Diagrams

Example

Group these activities in some logical fashion, possibly functional areas.

1. Create a list of activities

Customer Order
Serve Product
Collect Payment

Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

Pay for Labor

2. Construct Context Level DFD (identifies sources and sink)

Creating Data Flow Diagrams

Example

Create a level 0 diagram identifying the logical subsystems that may exist.

Customer Order
Serve Product
Collect Payment

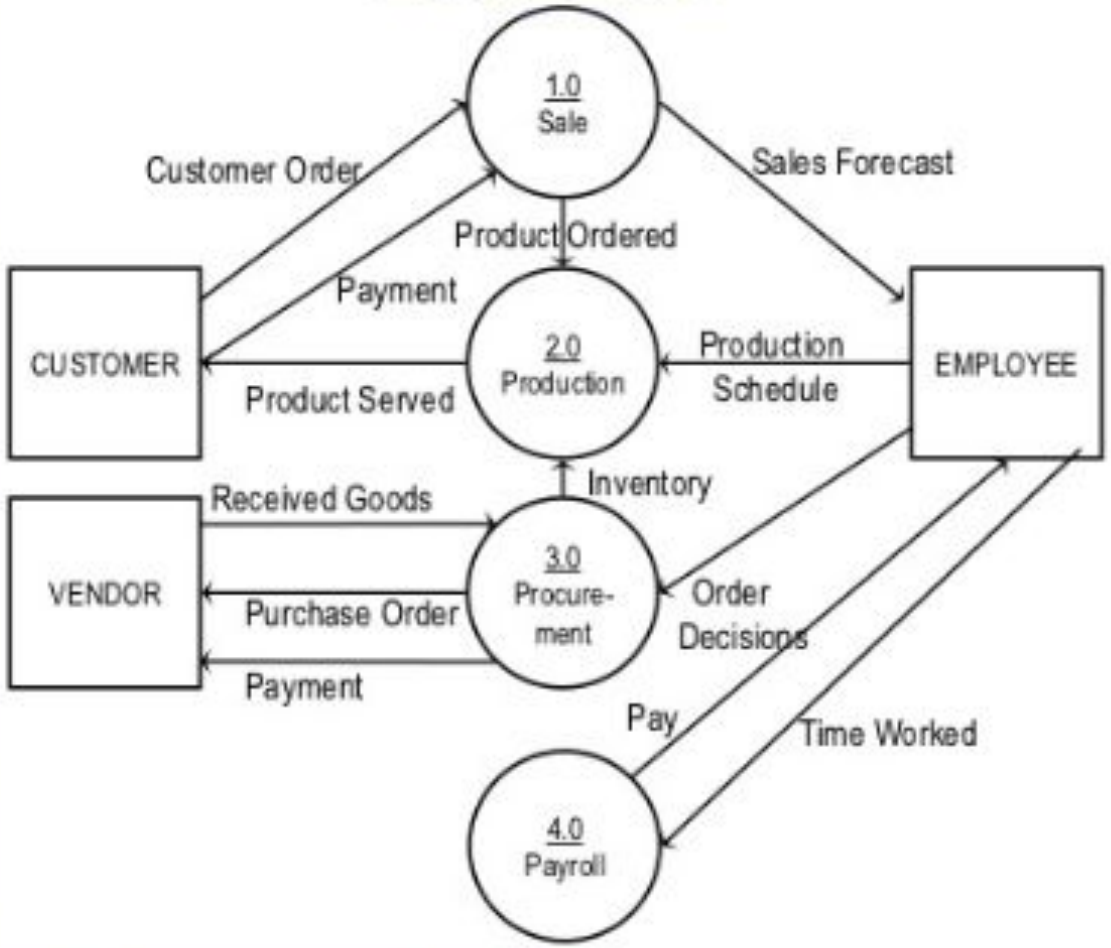
Produce Product
Store Product

Order Raw Materials
Pay for Raw Materials

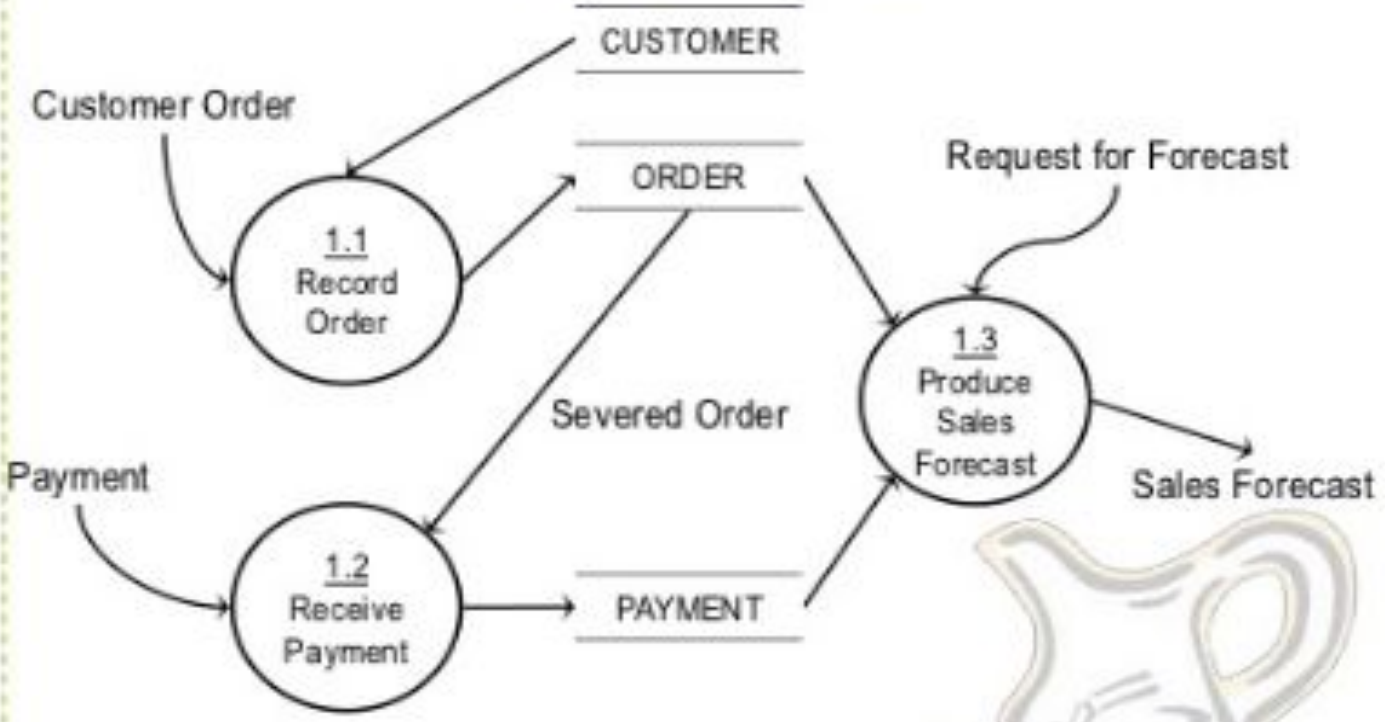
Pay for Labor

3. Construct Level 0 DFD
(identifies manageable sub processes)

Level 0 DFD

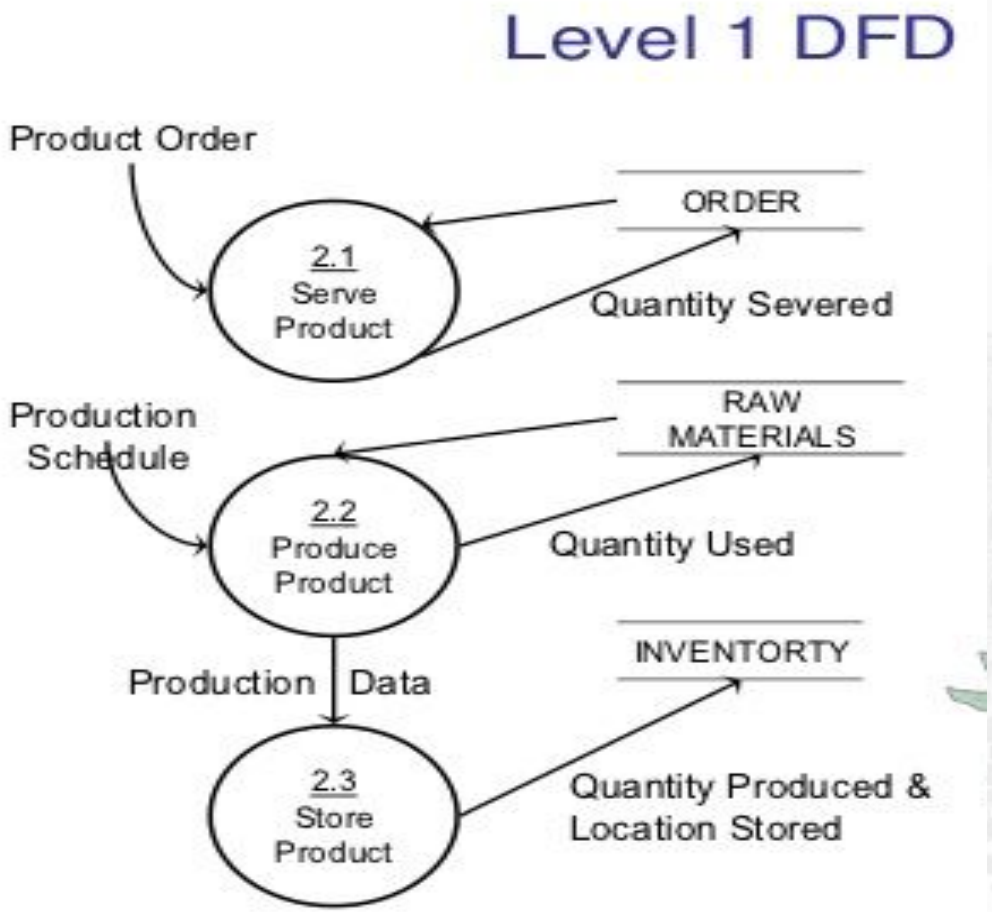


4. Construct Level 1- n DFD
(identifies actual data flows and data stores)



Level 1

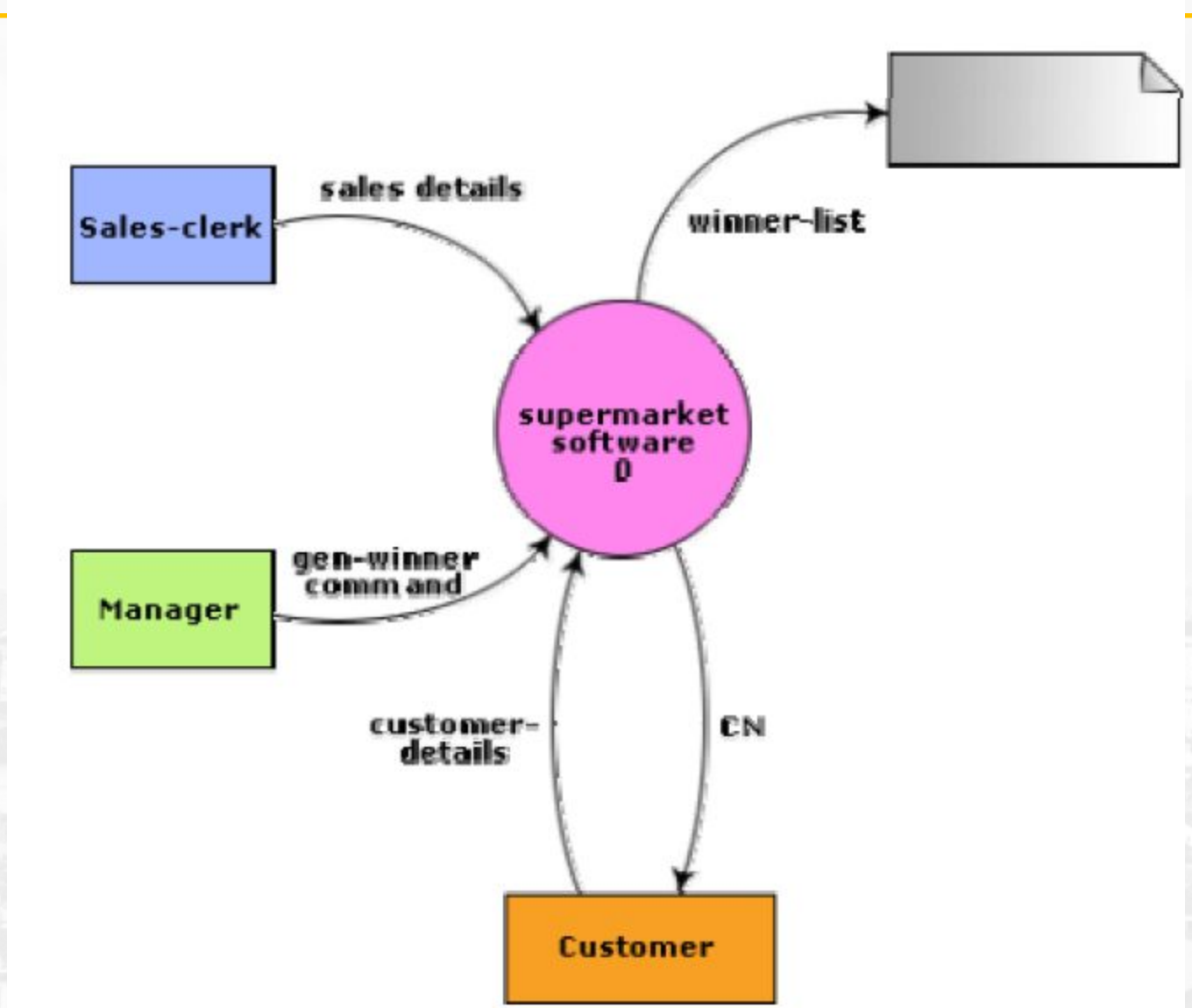
4. Construct Level 1 (continuec



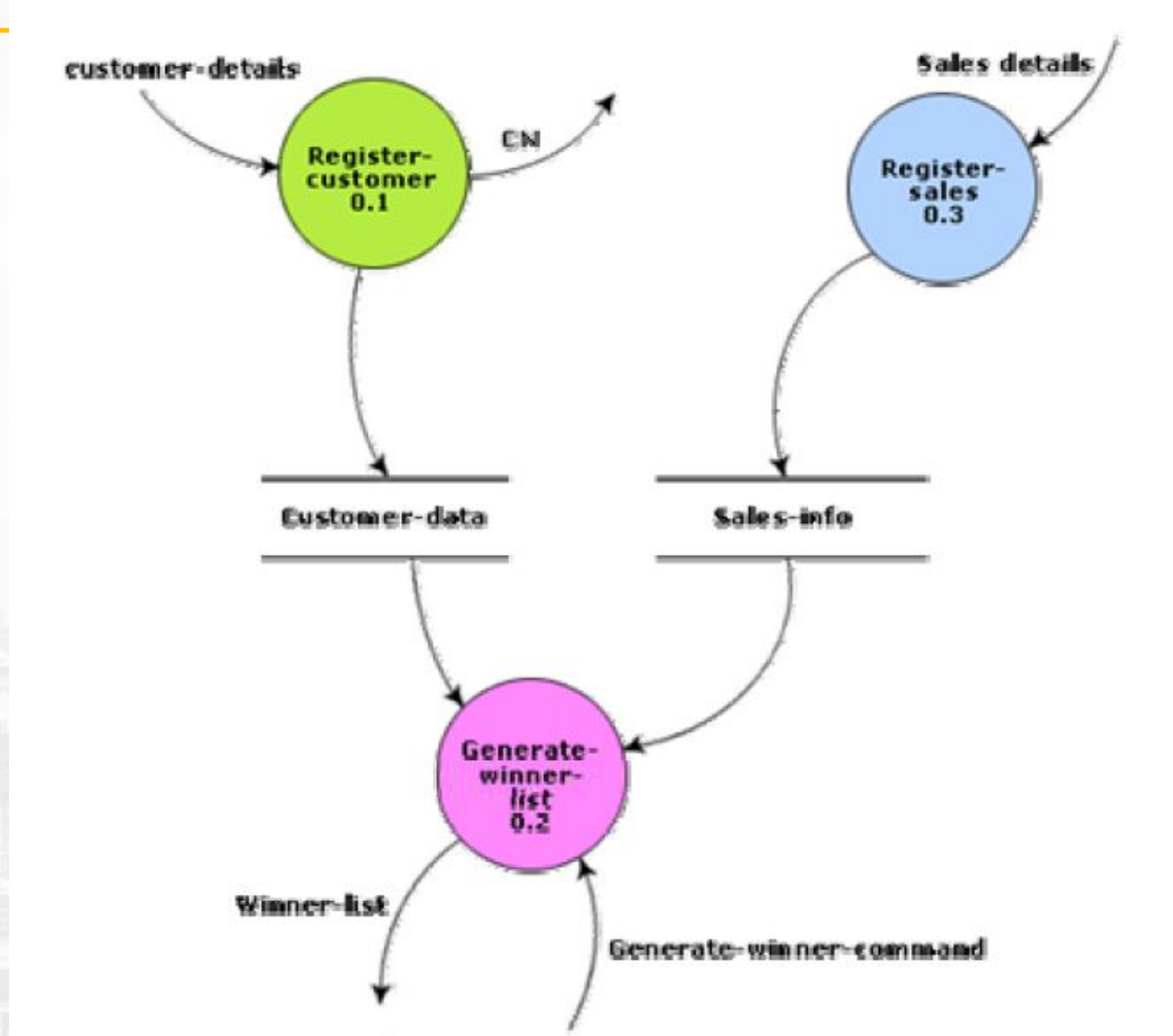
Example 3

- A supermarket needs to develop the following software to encourage regular customers. For this, the customer needs to supply his/her residence address, telephone number, and the driving license number. Each customer who registers for this scheme is assigned a unique customer number (CN) by the computer. A customer can present his CN to the check out staff when he makes any purchase. In this case, the value of his purchase is credited against his CN. At the end of each year, the supermarket intends to award surprise gifts to 10 customers who make the highest total purchase over the year. Also, it intends to award a 22 caret gold coin to every customer whose purchase exceeded Rs.10,000. The entries against the CN are the reset on the day of every year after the prize winners' lists are generated.

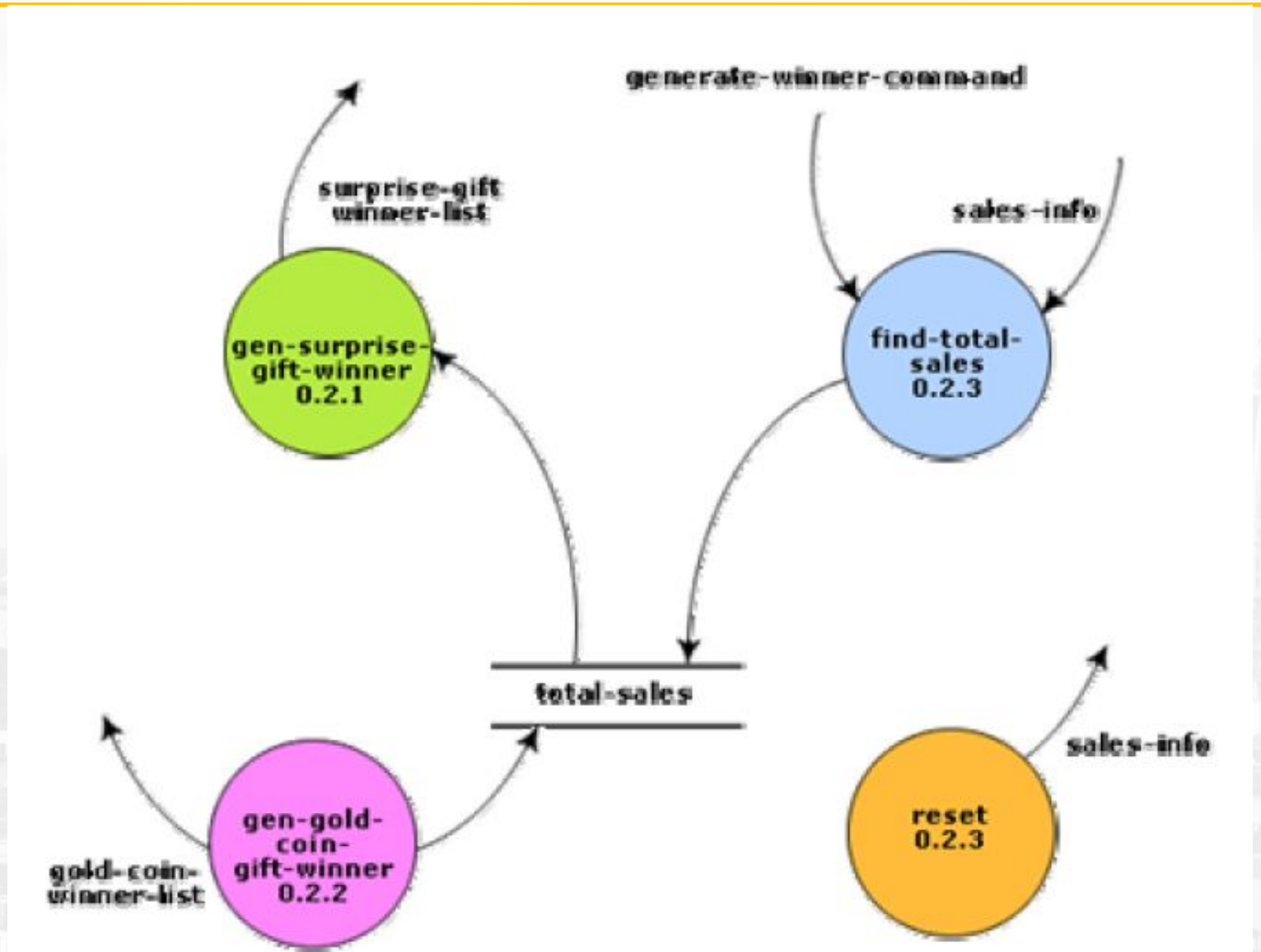
Level 0


Sales-clerk
sales details
winner-list
supermarket software 0
Manager
gen-winner command
customer-details
CN
Customer

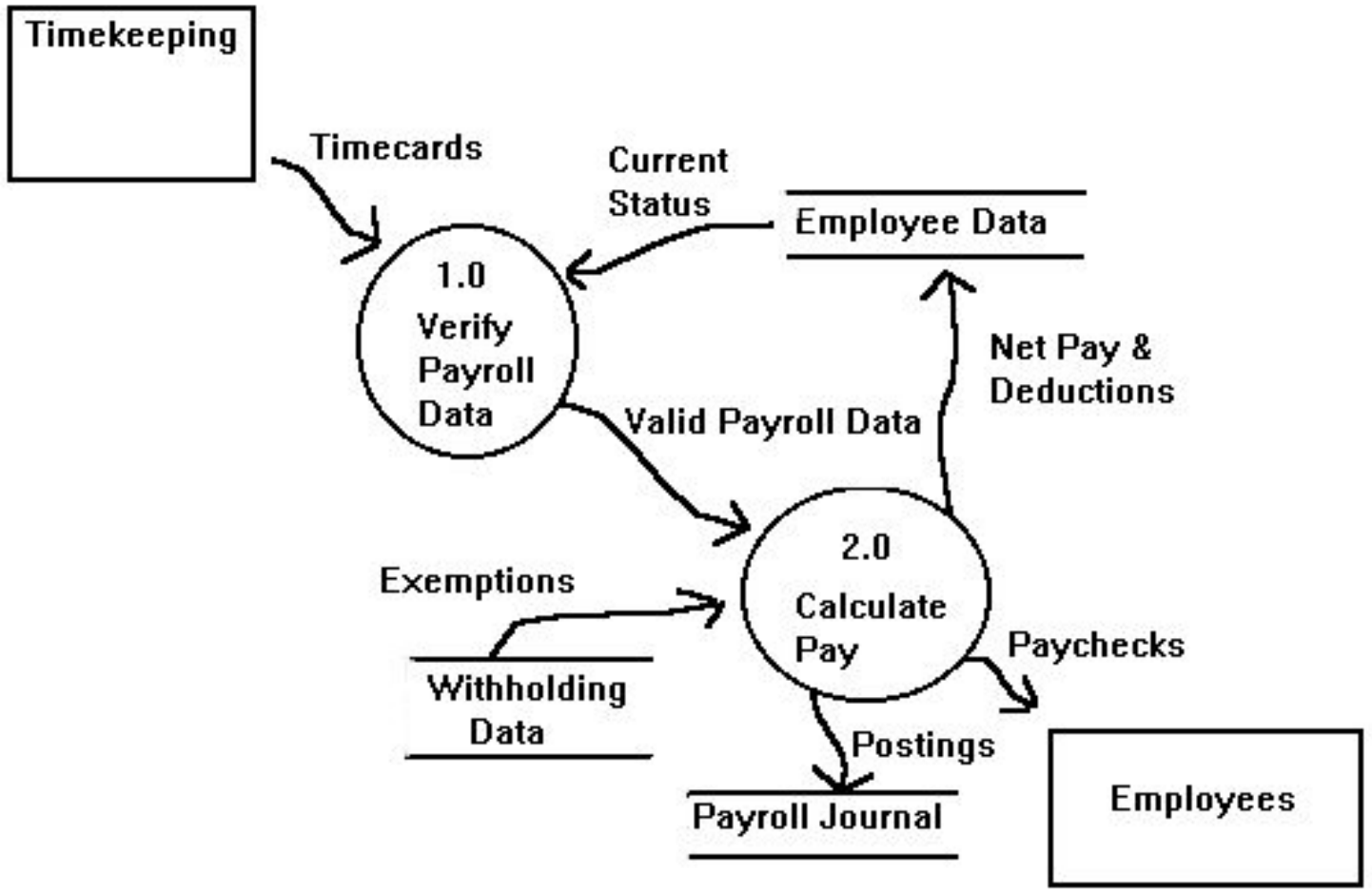
Level 1


customer-details
Register-customer 0.1
CN
Sales details
Register-sales 0.3
Customer-data
Sales-info
Generate-winner-list 0.2
Winner-list
Generate-winner-command

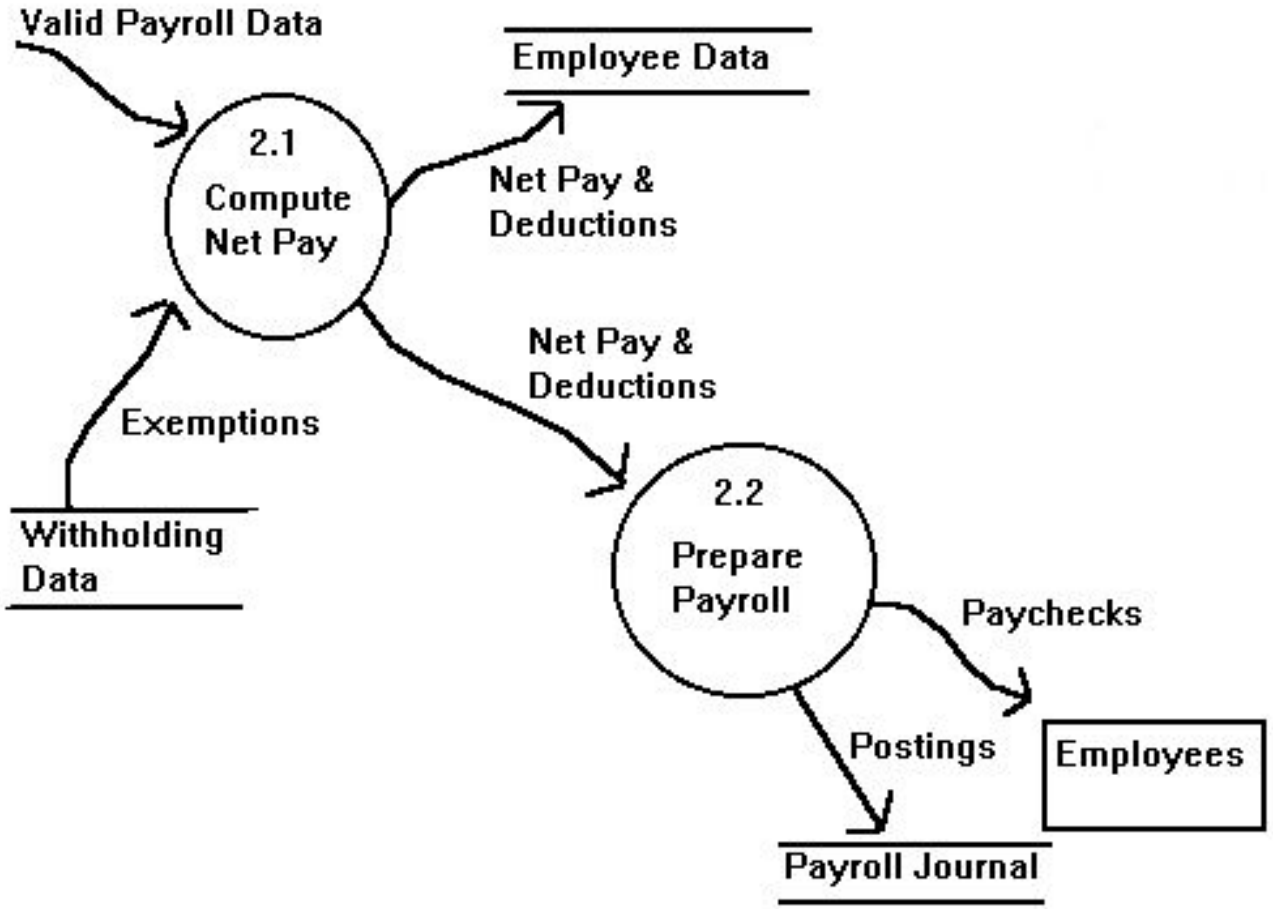
Level 2


generate-winner-command
surprise-gift winner-list
sales-info
gen-surprise-gift-winner 0.2.1
find-total-sales 0.2.3
total-sales
gold-coin-winner-list
gen-gold-coin-gift-winner 0.2.2
reset 0.2.3
sales-info

Level 1 :Payroll System



Level 2 :Payroll System

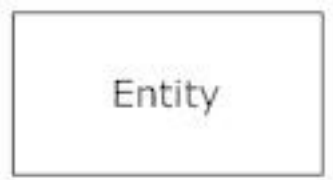


Entity Relationship Diagrams

- An ERD shows the relationships of entity sets stored in a database.
  - An entity in this context is an object, a component of data.
  - An entity set is a collection of similar entities.
  - These entities can have attributes that define its properties.
- ER diagram illustrates the logical structure of databases.
- ER diagrams are used to sketch out the design of a database.

ER Diagram Notations

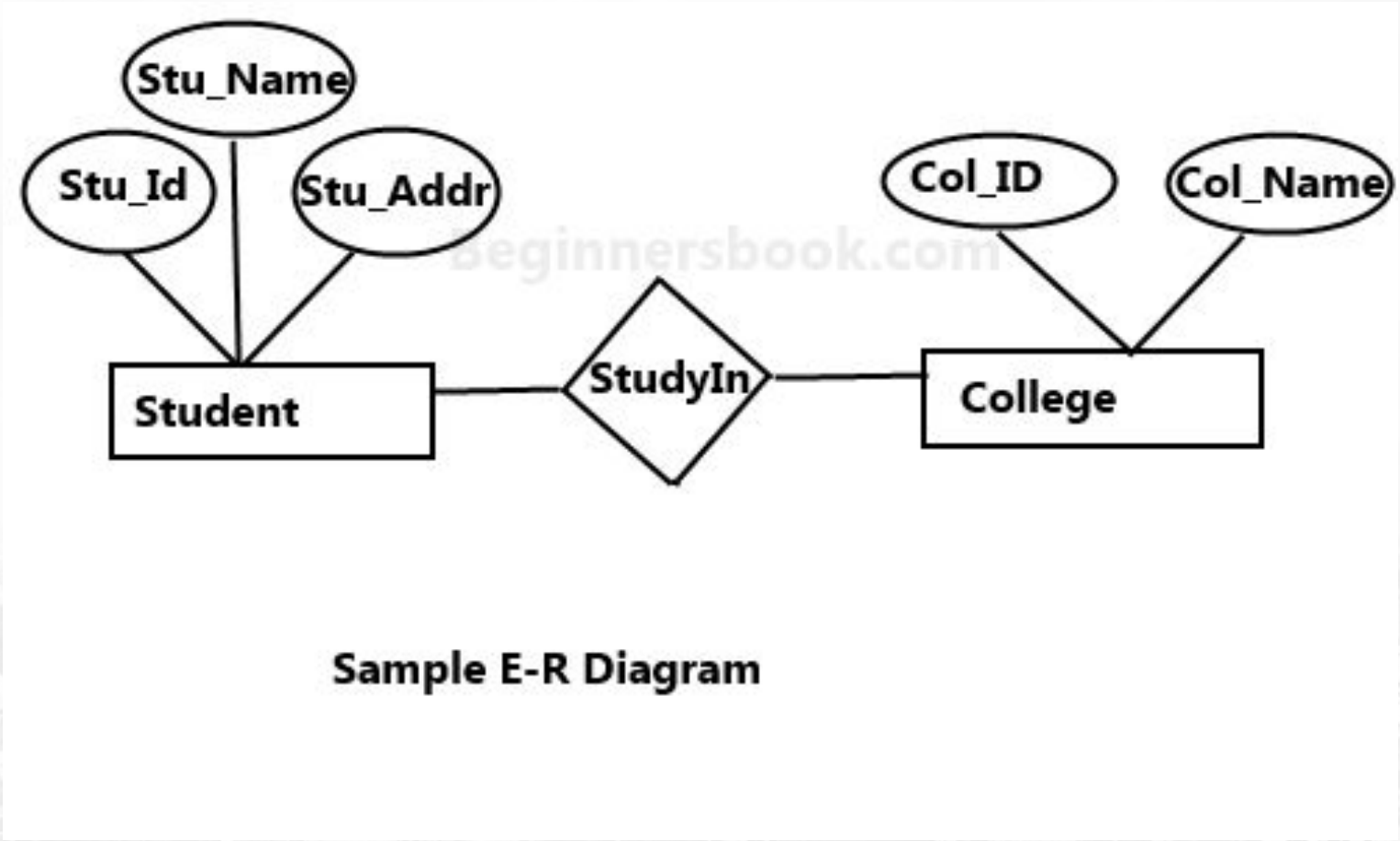
- Entities : is an object or concept about which you want to store information.



- Actions, which are represented by diamond shapes, show how two entities share information in the database.

- Attributes, which are represented by ovals. A key attribute is the unique, distinguishing characteristic of the entity.
  - A multivalued attribute can have more than one value. For example, an employee entity can have multiple skill values.
  - A derived attribute is based on another attribute. For example, an employee's monthly salary is based on the employee's annual salary.

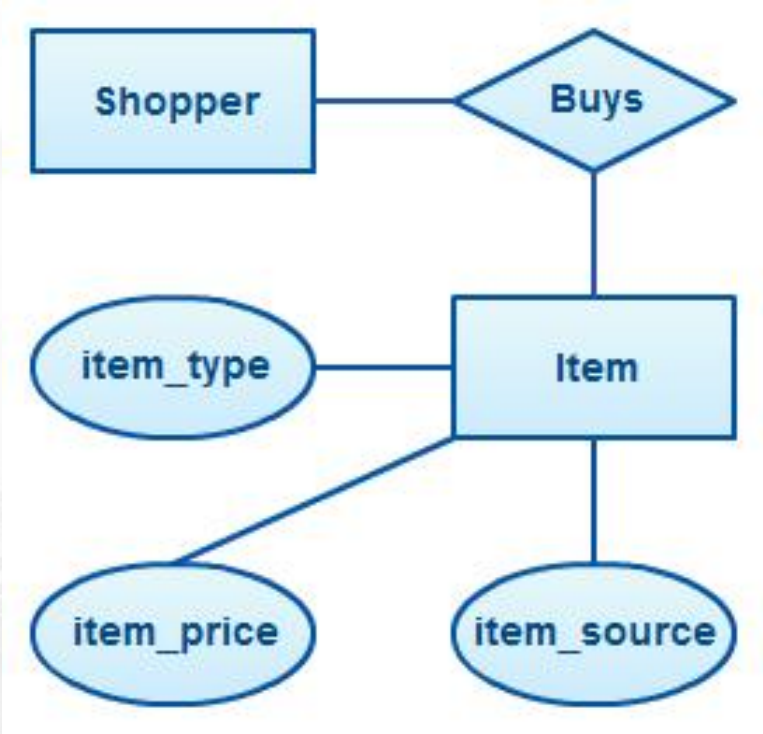
- Connecting lines, solid lines that connect attributes to show the relationships of entities in the diagram.



Sample E-R Diagram

Example 1

- Draw an ER diagram to show that shopper buys items.

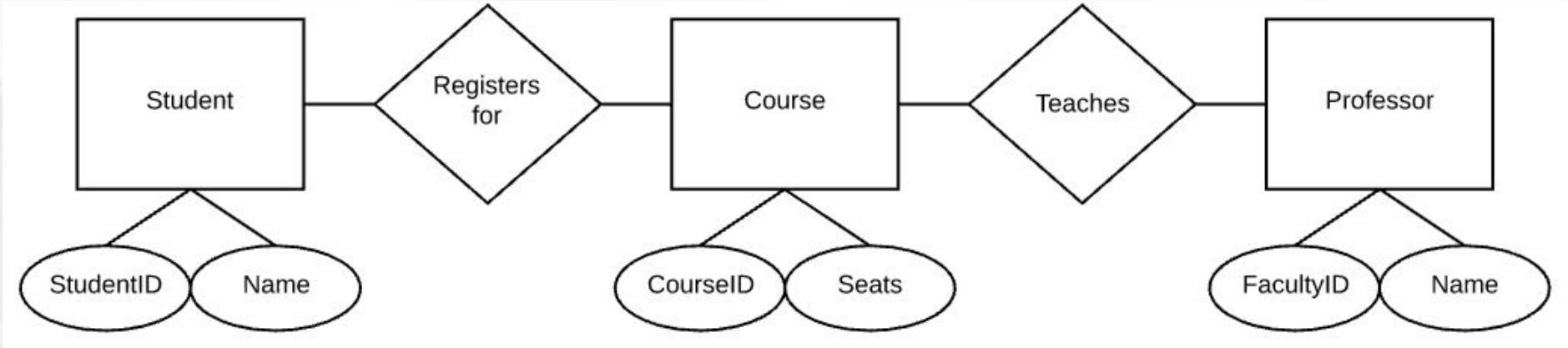


Example 2

- Draw an ER diagram to show that a student registers for a course. The course is taught by an Professor.

Example 2

- Draw an ER diagram to show that a student registers for a course. The course is taught by an Professor.

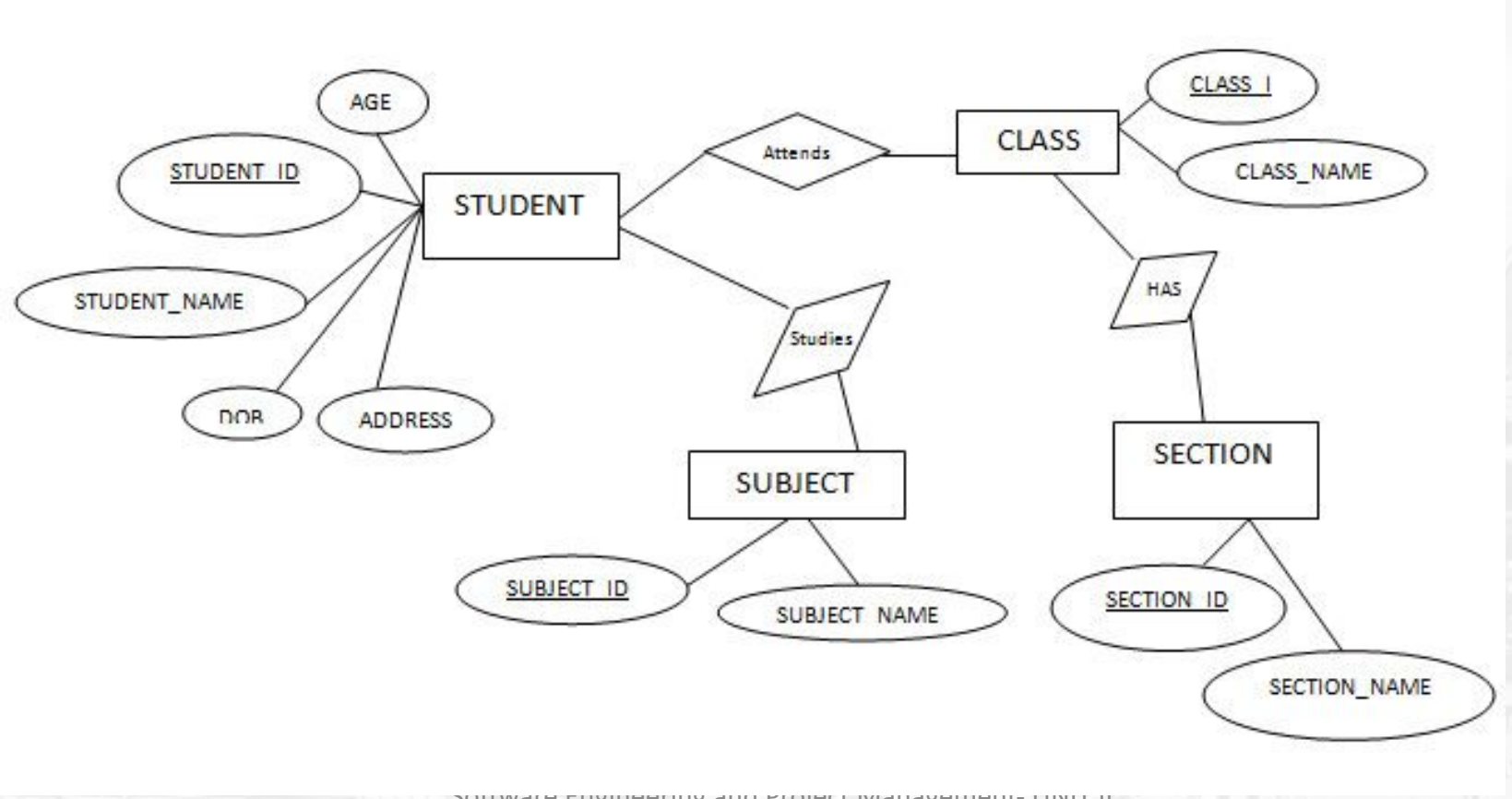


Example 3

- Draw ER diagram for : A student attends classes. Each class has many sections. The student takes up a subject.

Example 3

- Draw ER diagram for : A student attends classes. Each class has many sections. The student takes up a subject.



OOAD : Object Oriented Analysis and Design

- Object-Oriented Analysis : Is the procedure of identifying requirements and developing specifications by using interacting objects.
- It groups items that interact with one another, typically by class, data or behavior, to create a model that accurately represents the system
- The focus is on capturing the structure and behavior of systems into small modules that combines both data and process
- The primary tasks in object-oriented analysis (OOA) are −
  - Identifying objects
  - Organizing the objects by creating object model diagram
  - Defining the internals of the objects, or object attributes
  - Defining the behavior of the objects, i.e., object actions
  - Describing how the objects interact

OOAD

- Object-Oriented Design: Involves implementation of the conceptual model produced during object-oriented analysis
- The implementation details generally include –
  - Restructuring the class data (if necessary),
  - Implementation of methods, i.e., internal data structures and algorithms,
  - Implementation of control, and
  - Implementation of associations.

OOAD methods

Three major steps:

1. Identify the objects

2. Determine their attributes and services

3. Determine the relationships between objects

Object-Oriented Approach

- Views information system as collection of interacting objects that work together to accomplish tasks
    - OOA
    - OOD
    - OOP
- Object-oriented analysis (OOA)
  - Defines types of objects that do work of system
  - Shows how objects interact with users to complete tasks

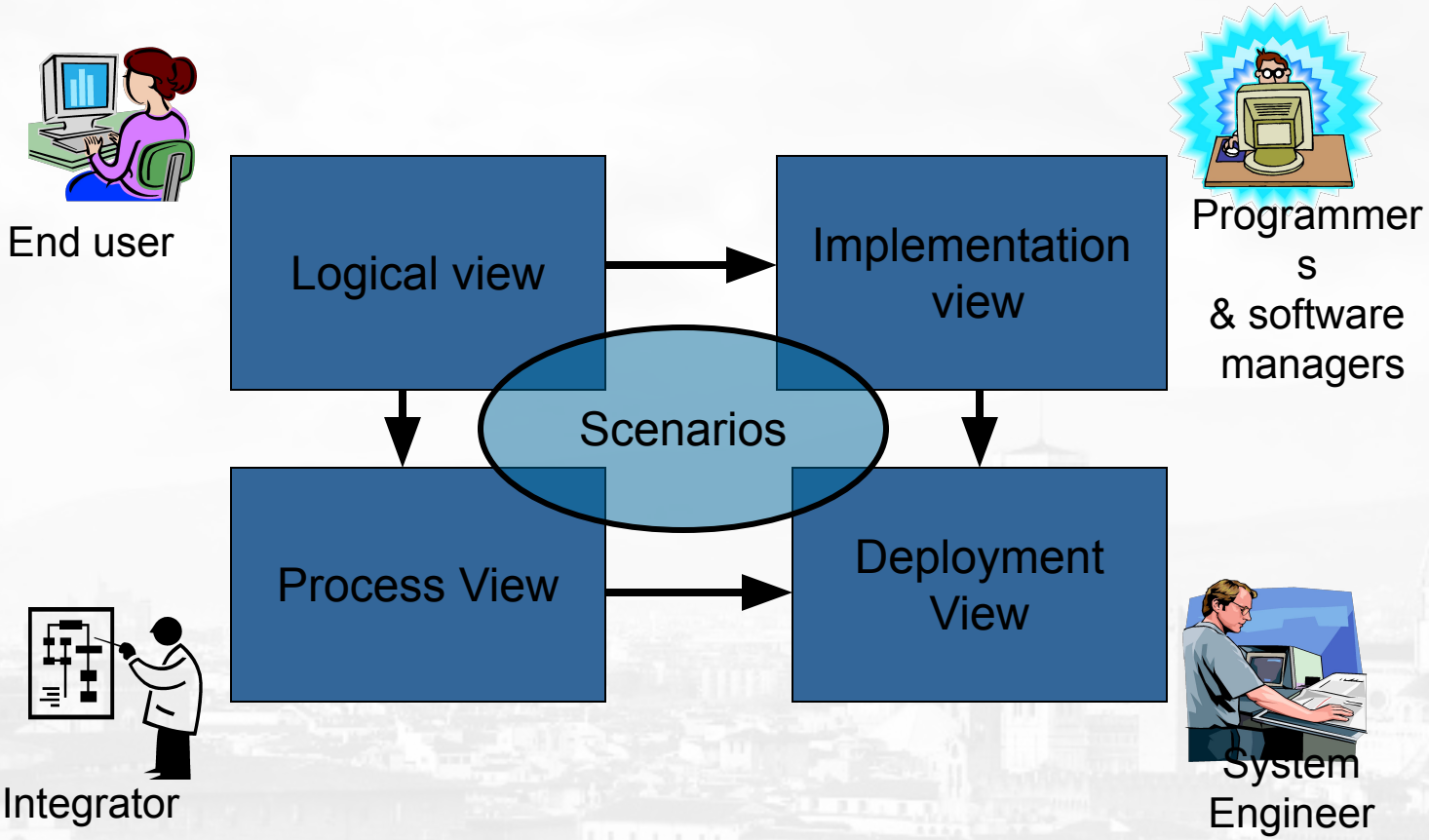
4+1 view Architecture:

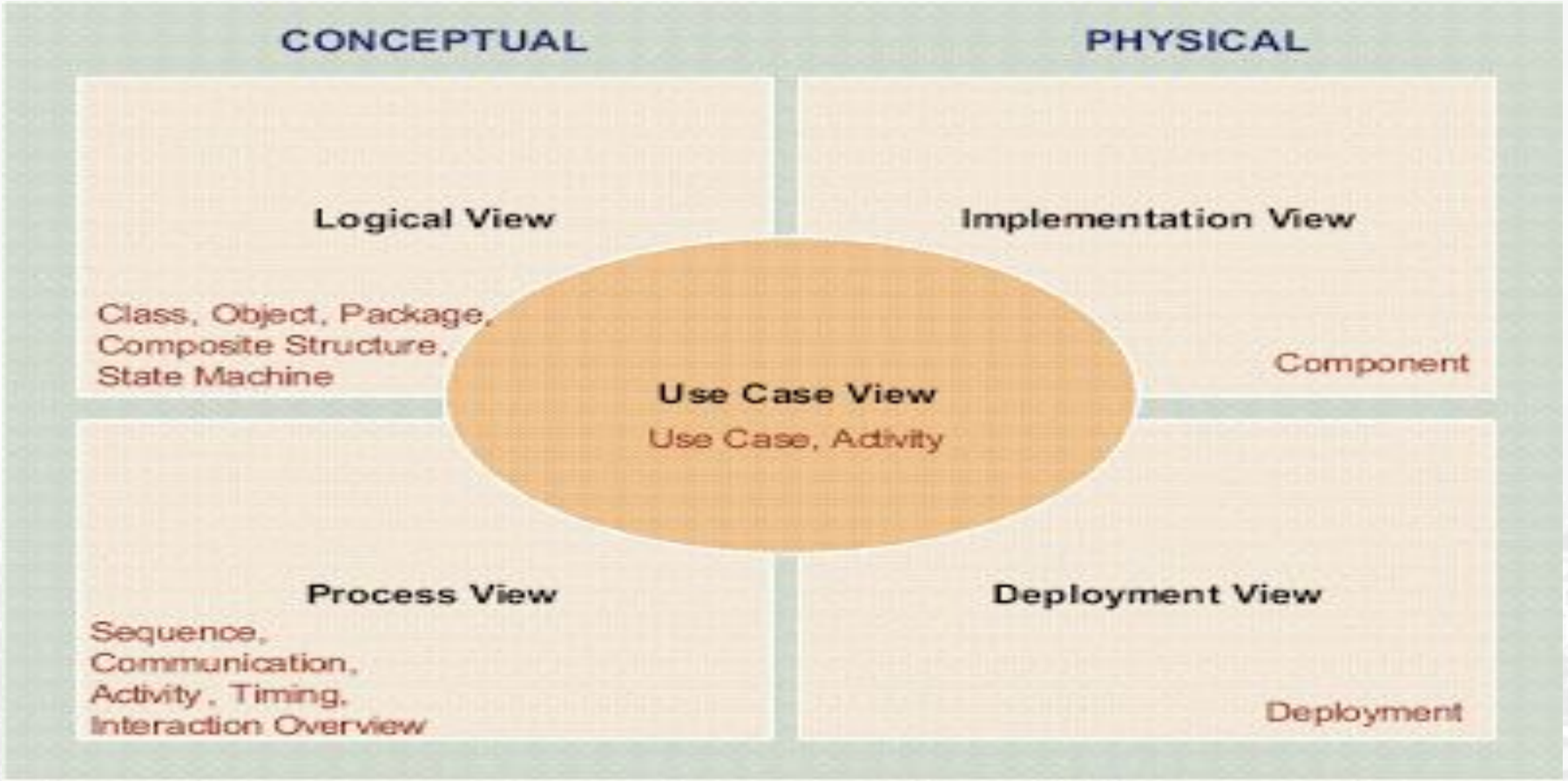
1. Arch. documents over-emphasize an aspect of development (i.e. team organization) or do not address the concerns of all stakeholders

2. Various stakeholders of software system: end-user, developers, system engineers, project managers

3. Software engineers struggled to represent more on one blueprint, and so arch. documents contain complex diagrams

4+1 View Model of Architecture




CONCEPTUAL
PHYSICAL
Logical View
Class, Object, Package,
Composite Structure,
State Machine
Implementation View
Component
Use Case View
Use Case, Activity
Process View
Sequence,
Communication,
Activity, Timing,
Interaction Overview
Deployment View
Deployment

Logical View

- **The logical view**, which is the object model of the design (when an object-oriented design method is used)

Viewer: End-user

considers: Functional requirements- What are the services must be provided by the system to the users.

Process View

The process view, which captures the concurrency and synchronization aspects of the design**(The process decomposition).**

viewer: Integrators

considers: Non - functional requirements (scalability, concurrency, and performance)

Implementation View

The *Implementation* view, which describes the **static** organization of the software in its development environment.

Viewer: Programmers and Software Managers

considers: software module organization.
(Hierarchy of layers, software management, reuse, constraints of tools).

Deployment View

the Deployment view, which describes the mapping(s) of the software onto the hardware and reflects its distributed aspect.

Viewer: System Engineers

Considers: Non-functional requirement (reliability, availability and performance). regarding to underlying hardware.

There may be two architecture:

- Test and development
- deployment

Scenarios

(Putting all “4 views” together)

Viewer: All users and Evaluators.

Considers: System consistency and validity

Notation: Similar to logical view

General Advantages

- Understandable
  - maps the “real-world” objects more directly
  - manages complexity via abstraction and encapsulation
- Practical
  - successful in real applications
  - suitable to many, but not all, domains
- Productive
  - experience shows increased productivity over life-cycle
  - encourages reuse of model, design, and code
- Stable
  - changes minimally perturb objects

Unified Modeling Language (UML)

- **UML** (Unified Modeling Language) is a modeling language **used** by software developers.
- **UML** can be **used** to develop diagrams and provide users with ready-to-**use**, expressive modeling examples.
- Some **UML** tools generate program language code from **UML**.
- It's a rich language to model software solutions, application structures, system behavior and business processes
- **UML** can be **used** for modeling a system independent of a platform language.

What is UML?

- UML (Unified Modeling Language)
  - An emerging standard for modeling object-oriented software.
  - Resulted from the convergence of notations from three leading object-oriented methods:
    - OMT (James Rumbaugh)
    - OOSE (Ivar Jacobson)
    - Booch (Grady Booch)
- Reference: “The Unified Modeling Language User Guide”, Addison Wesley, 1999.
- Supported by several CASE tools
    - Rational ROSE
    - TogetherJ

Systems, Models and Views

- A ***model*** is an abstraction describing a subset of a system
- A ***view*** depicts selected aspects of a model
- A ***notation*** is a set of graphical or textual rules for depicting views
- Views and models of a single system may overlap each other

Examples:

- System: Aircraft
- Models: Flight simulator, scale model
- Views: All blueprints, electrical wiring, fuel system

UML Concepts

- UML can be used to support your entire life cycle
  - UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
  - The interaction of your application with the outside world (use case diagram)
  - Visualize object interaction (sequence & collaboration diagrams)
  - The structure of your system (class diagram)
  - View the system architecture by looking at the defined package.
  - The components in your system (component diagram)

What is UML

- Unified modeling language (UML) for visualizing, specifying, constructing, documenting of artifact of a software system
- The blueprint of a system is written in it
- UML is also used for modeling non-software system
- It is standard for building object oriented and component based software system
- UML is a notation system though which we can visualize a model of a system
- It describe only design or structure of system

UML 2.0 Diagrams

- UML is divided in to two General set of Diagrams
- Structured modeling diagrams
  - It depicts the static view of the model
- Behavioral modeling diagram
  - Behavior diagrams depicts the varieties of interaction within a model as it 'executes' over time

Structural Modeling Diagram(Static Design)

- Class Diagram
- Object Diagram
- Component Diagram
- Package Diagram
- Composite Structure
- Deployment Diagram

Behavioral Modeling Diagram(Dynamic Design)

- Use Case Diagram
- Activity Diagram
- State Machine Diagram
- Communication Diagram
- Sequence Diagram
- Timing Diagram
- Interaction Over view Diagram

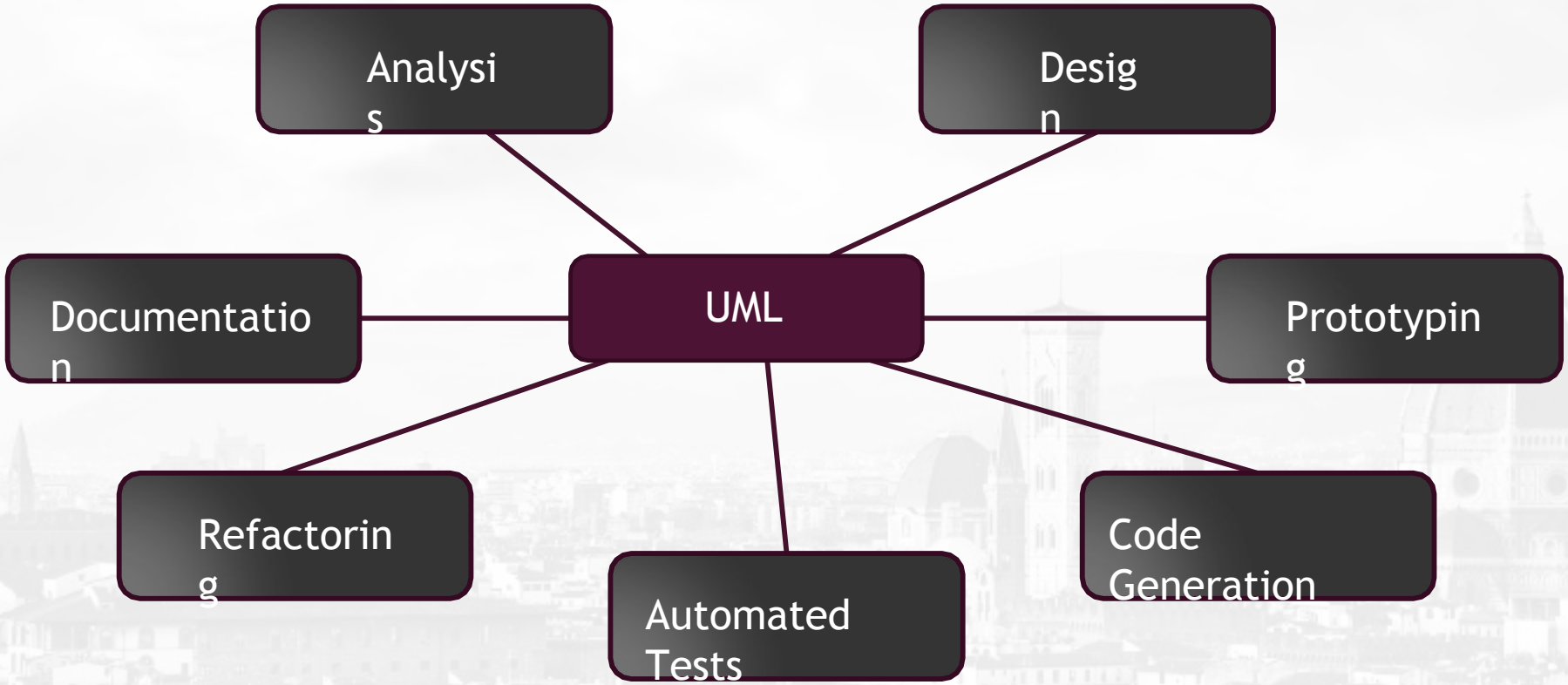
Introduction to UML and Building Blocks

- Introduction to UML Diagram.
- A conceptual model of UML
- building blocks of UML
  - Things
  - Relationships
  - Diagrams

INTRODUCTION TO UML DIAGRAM

- UML stands for **U**nified **M**odelling **L**anguage.
- UML is a standard language for specifying, visualizing, constructing, and documenting a system in which software represents the most significant part.
- UML is different from the other common programming languages like C++, Java, COBOL etc.
- UML is a pictorial language used to make software blue prints.
- UML can serve as a central notation for software development process. Using UML helps project teams communicate, explore potential designs, and validate the architectural designs of software.
- UML diagrams are made using notation of things and relationships.

INTRODUCTION TO UML DIAGRAM



A CONCEPTUAL MODEL OF UML

Conceptual model: A conceptual model can be defined as a model which is made of concepts and their relationships.

- As UML describes the real time systems it is very important to make a conceptual model and then proceed gradually. Conceptual model of UML can be better understood by learning the following three major elements :
  - *UML building blocks*
  - *Rules to connect the building blocks Common mechanisms of UML*

BUILDING BLOCKS OF UML

The building blocks of UML can be defined as:

- Things
- Relationshi
- Diagrams

Things: Things are the most important building blocks of UML. Things can be:

- Structural
- Behavioral
- Grouping
- Annotations
- I

STRUCTURAL THINGS

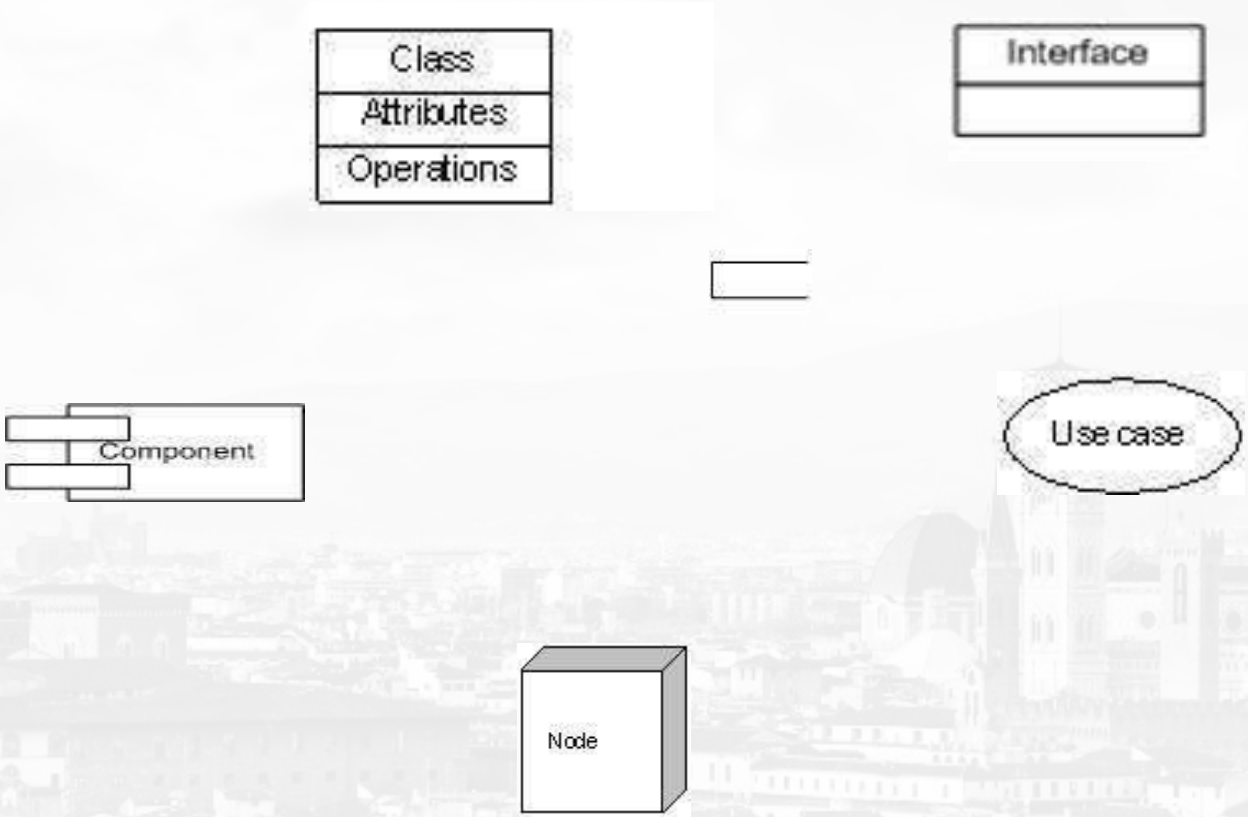
The **Structural things** define the static part of the model. They represent physical and conceptual elements. Following are the brief descriptions of the structural things.

- **Class:** Class represents set of objects having similar responsibilities.
- **Interface:** Interface defines a set of operations which specify the responsibility of a class.

ts

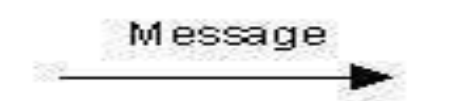
- **Collaboration:** Collaboration defines interaction between elements
- **Use case:** Use case represents a set of actions performed by a system for a specific goal.
- **Component:** Component describes physical part of a system.
- **Node:** A node can be defined as a physical element that exists at run

Continued…


Class
Attributes
Operations
Interface
Component
Use case
Node

BEHAVIORAL THING

- A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things:
  - Interaction: Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



  - State machine: State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change.

RELATIONSHIP

Relationship is another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application.

Continued

- Dependency: Dependency is a relationship between two things in which change in one element also affects the other one.
- Association: Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.
- Generalization: Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.
- Realization: Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.

Why an extension mechanism?

- Although UML is very well-defined, there are situations in which it needs to be customized to specific problem domains

- UML extension mechanisms are used to extend UML by:
  - adding new model elements,
  - creating new properties,
  - and specifying new semantics

- There are three extension mechanisms:
  - stereotypes, tagged values, constraints and notes

Stereotypes

- Stereotypes are used to extend UML to create new model elements that can be used in specific domains
- E.g. when modeling an elevator control system, we may need to represent some classes, states etc. as
  - «hardware»
  - «software»
- Stereotypes should always be applied in a consistent way

Stereotypes (cont.)

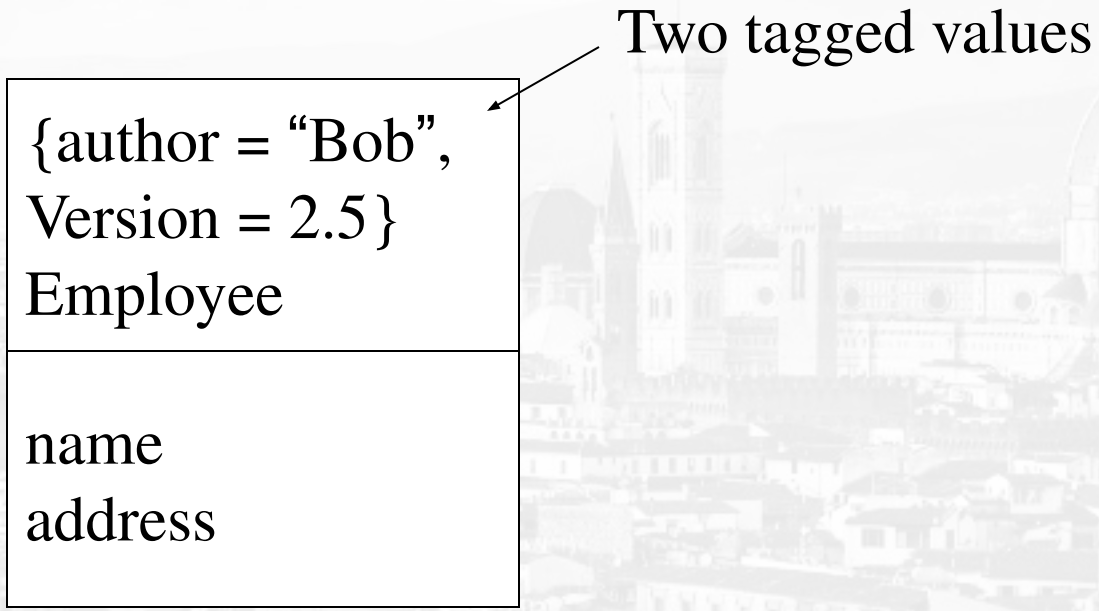
- Ways of representing a stereotype:
  - Place the name of the stereotype above the name of an existing UML element (if any)
    - The name of the stereotype needs to be between «» (e.g. «node»)
    - Don't use double '<' or '>' symbols, there are special characters called open and close
  - Create new icons

Tagged Values

- Tagged values
  - Define additional properties for any kind of model elements
  - Can be defined for existing model elements and for stereotypes
  - Are shown as a tag-value pair where the tag represent the property and the value represent the value of the property
- Tagged values can be useful for adding properties about
  - code generation
  - version control
  - configuration management
  - authorship
  - etc.

Tagged Values (cont.)

- A tagged value is shown as a string that is enclosed by brackets {} and which consists of:
  - the tag, a separator (the symbol =), and a value



Constraints

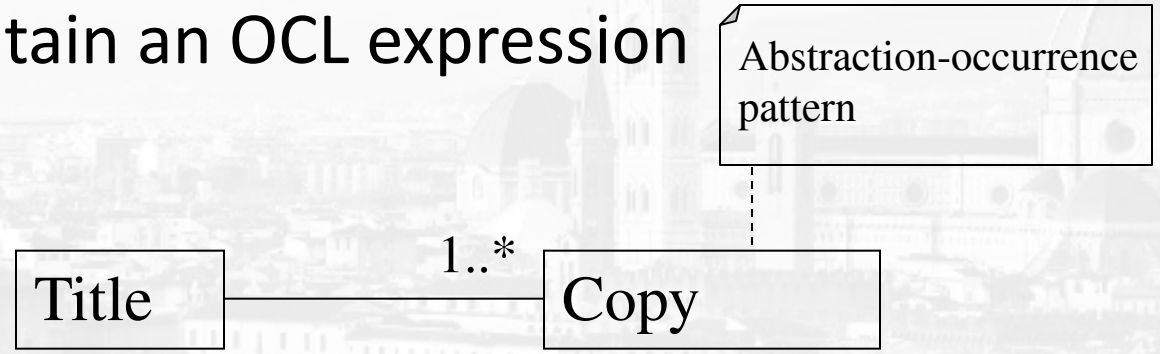
- Constraints are used to extend the semantics of UML by adding new rules, or modifying existing ones.
- Constraints can also be used to specify conditions that must be held true at all times for the elements of a model.
- Constraints can be represented using the natural language or OCL (Object Constraint Language)
  - We will learn more about OCL in future lectures

Comments

- Comments are used to help clarify the models that are being created
  - e.g. comments may be used for explaining the rationale behind some design decisions

- A comment is shown as a text string within a note icon.

- A note icon can also contain an OCL expression



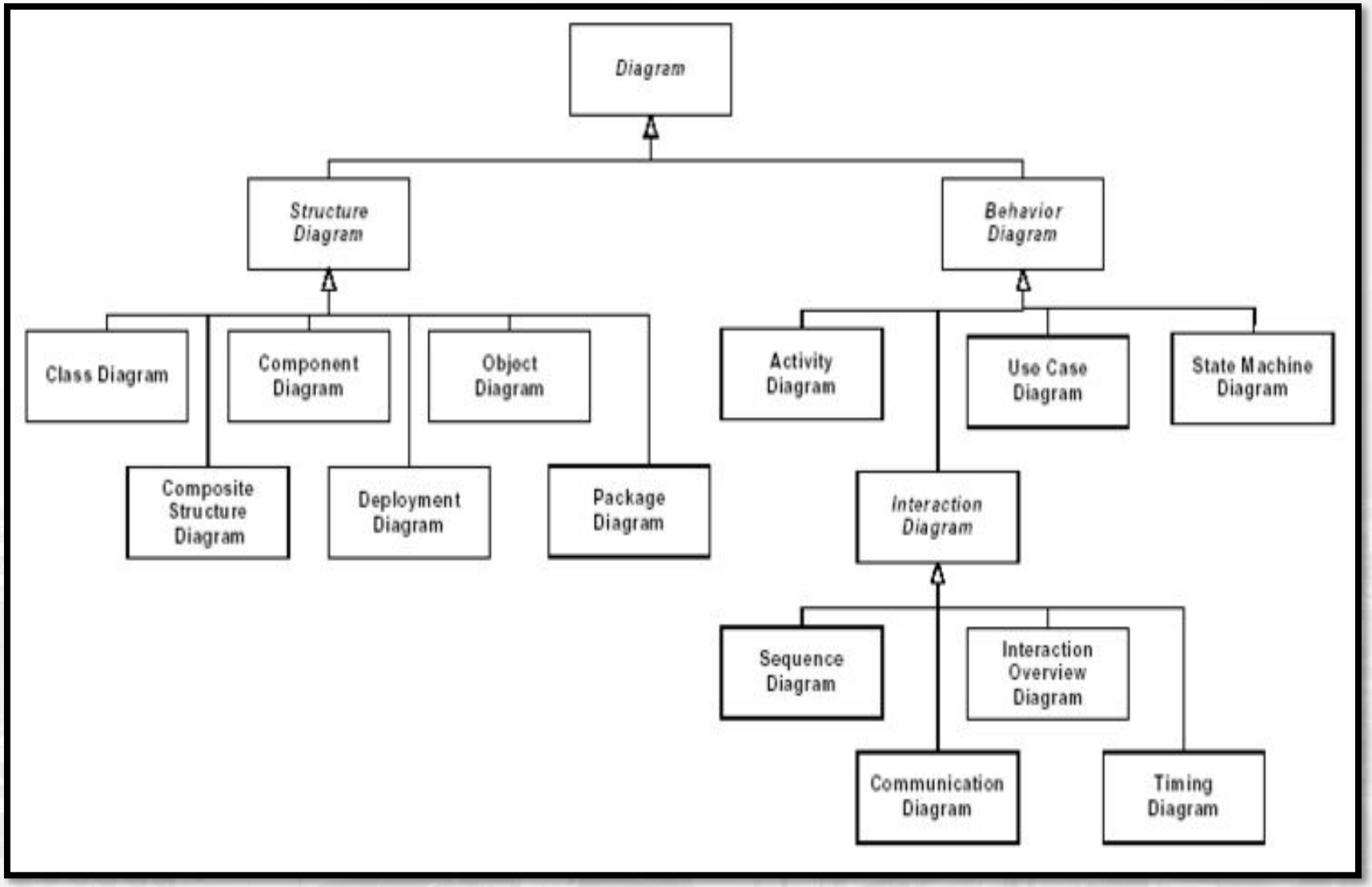
UML

- UML diagrams
  - **Static modeling**

(Structure Diagrams)

  - **Dynamic modeling**

(Behavior Diagrams)



Static Modeling(static Design)

- Class diagrams
- Object diagrams
- Composite structure diagrams
- Package diagrams
- Component Diagram - Interfaces and Components
- Deployment Diagram

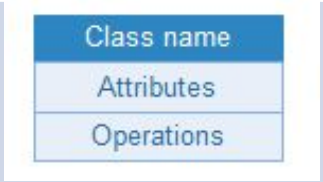
Class Diagram

- A class diagram depicts classes and their interrelationships
- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are useful for software developers

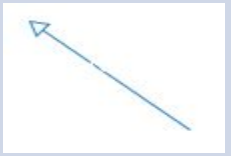
CLASS DIAGRAMS

- Class diagrams are one of the most fundamental diagram types in UML.
- They are used to capture the **static** relationships of your software; in other words, how things are put together
- A class represents a **group of things** that have common state and behavior.
- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Detailed class diagrams are used for developers

Class Diagram

S.No	Name	Description	Notation
1	Classes and interface	They are used to show the different objects in a system, their attributes, their operations and the relationships among them.	
2	Object	An object is an instance or occurrence of a class	
3	Aggregation	An aggregation describes a group of objects and how you interact with them.	
4	Composition	Composition represents whole-part relationships and is a form of aggregation.	
5	Dependency	Dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier.	

Class Diagram

S.No	Name	Description	Notation
3	Generalization	Generalization is a relationship in which one model element (the child) is based on another model element (the parent).	
4	Association	Association is a relationship between two classifiers, such as classes or use cases, that describes the reasons for the relationship and the rules that govern the relationship.	
5	Multiplicity		**Multiplicity** Symbol — Meaning 1 — One and only one 0..1 — Zero or one M..N — From M to N (natural language) * — From zero to any positive integer 0..* — From zero to any positive integer 1..* — From one to any positive integer

Drawing a Class Diagram ?

- Identify and model classes—Which classes do we need?
- Identify and model associations—How are the classes connected?
- Define attributes—What do we want to know about the objects?

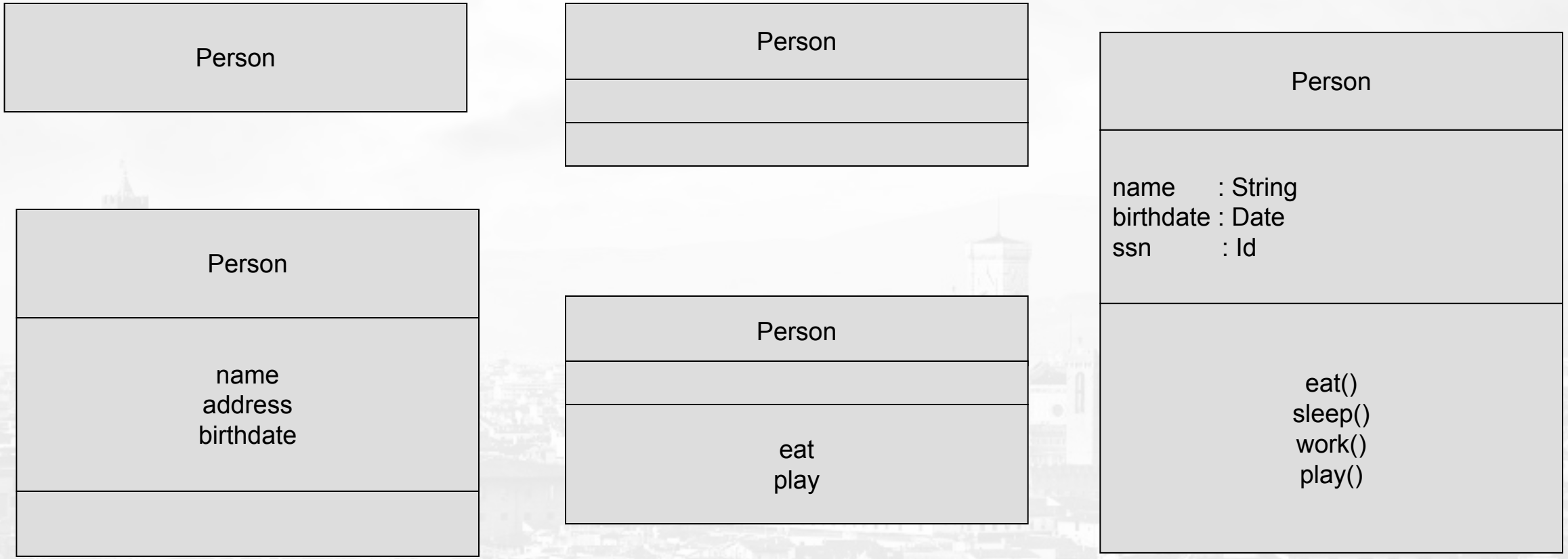
A Single Class

Rectangle
- width: int - height: int / area: double
+ Rectangle(w: int, h: int) + distance(r: Rectangle): double

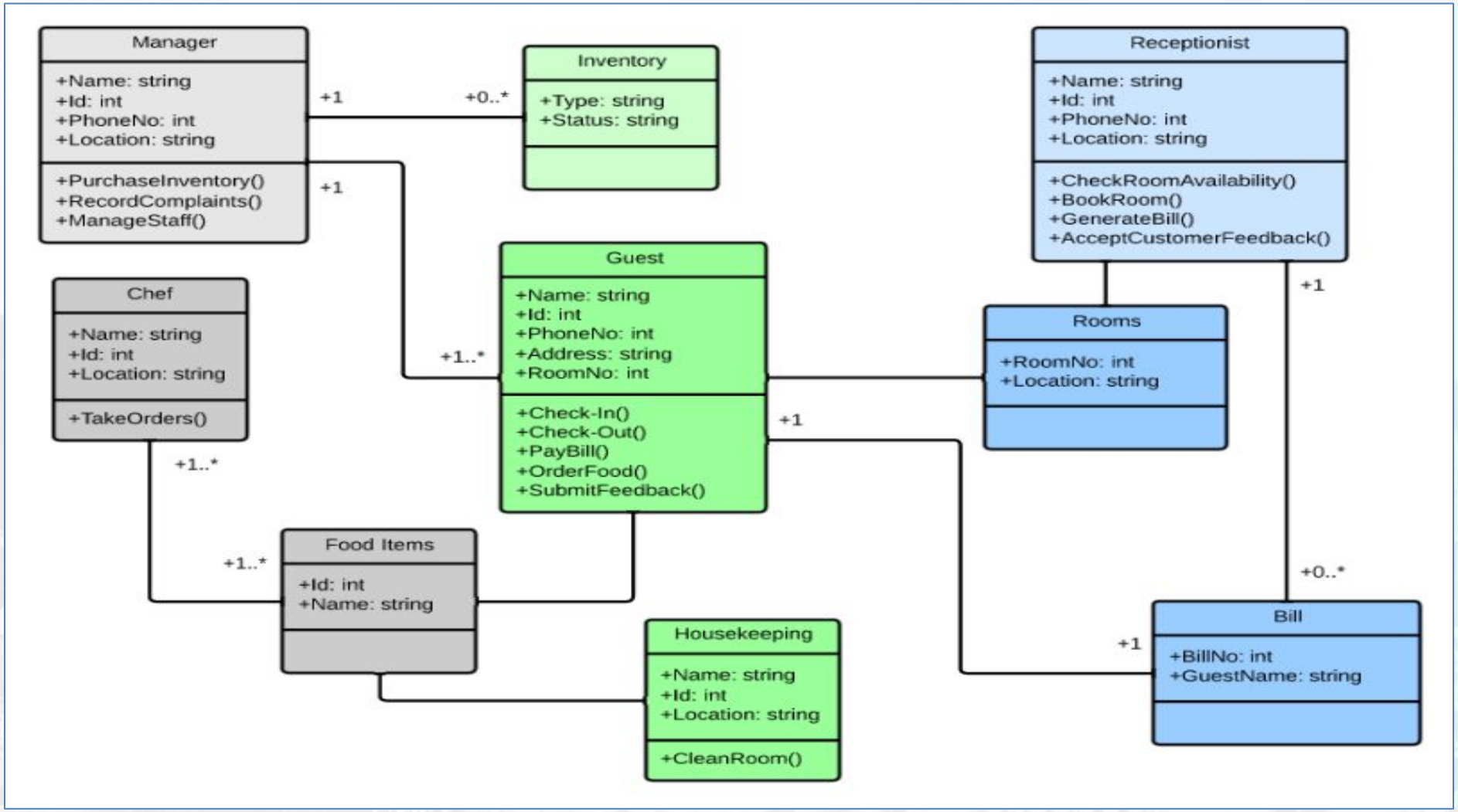
Student
- name: String - id: int - totalStudents: int
getID(): int ~ getEmail(): String

Depicting Classes

- When drawing a class, you needn't show attributes and operation in every diagram.



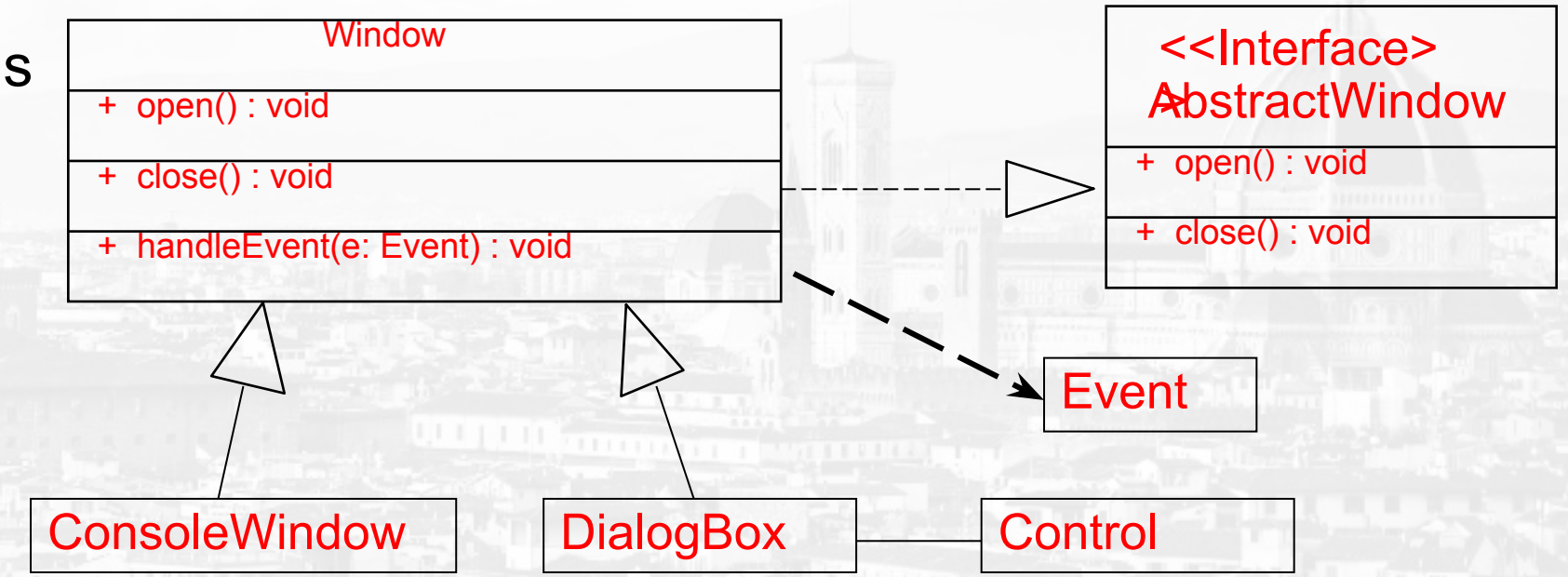
Class Diagram : Hotel Management System



Relationships

In UML, object interconnections (logical or physical), are modeled as relationships.
There are four kinds of relationships in Class Diagram:

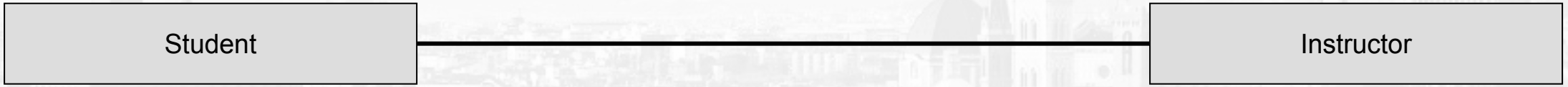
- Association
- Generalizations
- Dependency
- Realization



Association Relationships

- If two classes in a model need to communicate with each other, there must be link between them.

- An association denotes that link.

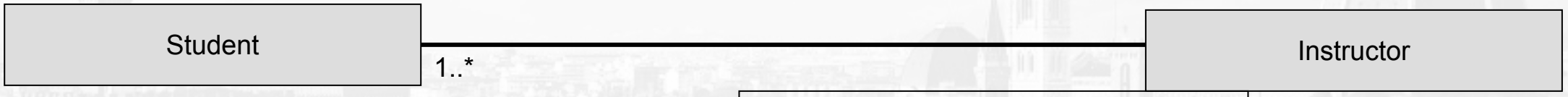


Association Relationships (Cont'd)

- We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association.
- Instance of one class interacting with the instance of another class

- The example indicates that a Student has one or more Instructors:

The example indicates that every *Instructor* has one or more *Students*:



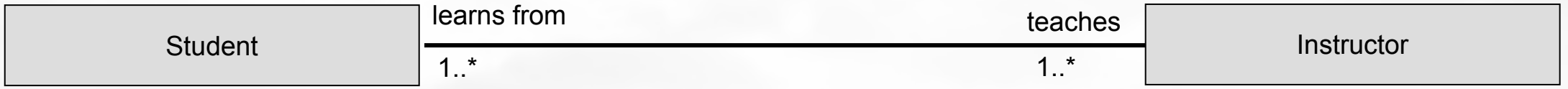
- **Optional (0 or 1) 0..1**
- **Exactly one 1= 1..1**
- **Zero or more 0..*= ***
- **One or more 1..***
- **A range of values 2..6**

Multiplicity

Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Association Relationships (Cont'd)

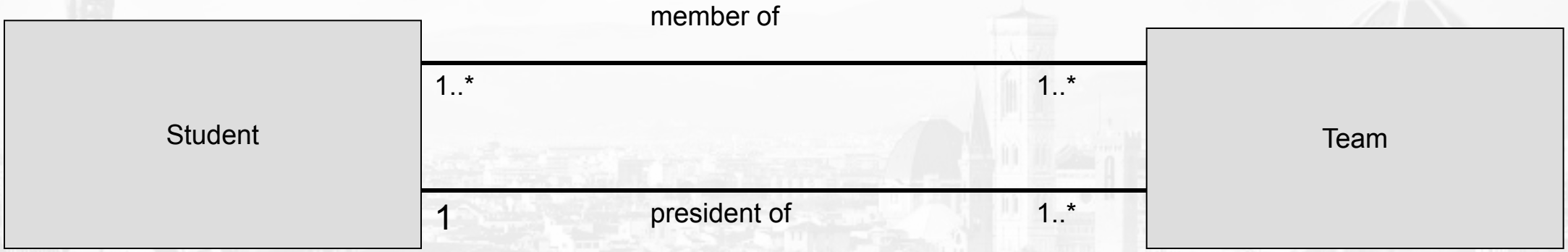
- We can also indicate the behavior of an object in an association (*i.e.,* the *role* of an object) using *rolenames.*



We can also name the association.

Association Relationships (Cont'd)

- We can specify dual associations.



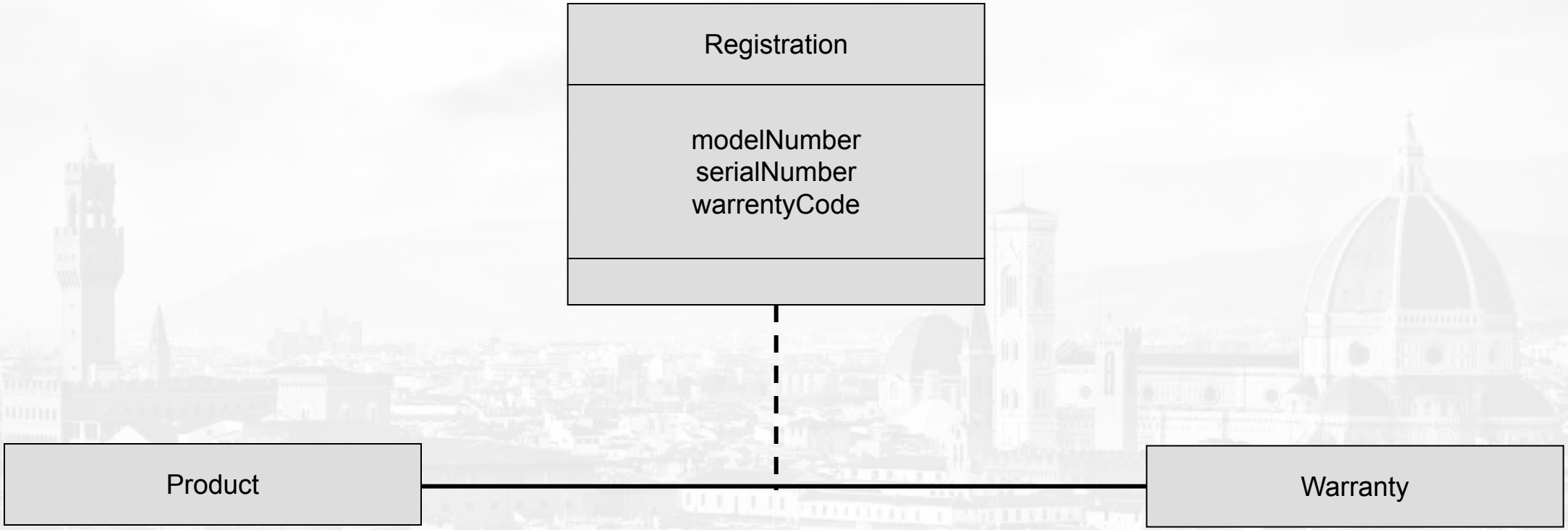
Association Relationships (Cont'd)

- Navigability of the association.

- Here, a Router object requests services from a DNS object by sending messages to (invoking the operations of) the server.
- The direction of the association indicates that the server has no knowledge of the Router.

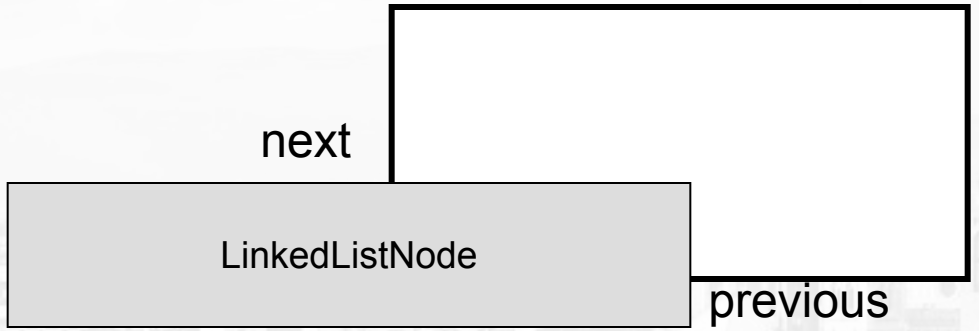
Association Relationships (Cont'd)

- Associations can also be objects themselves, called as association classes.

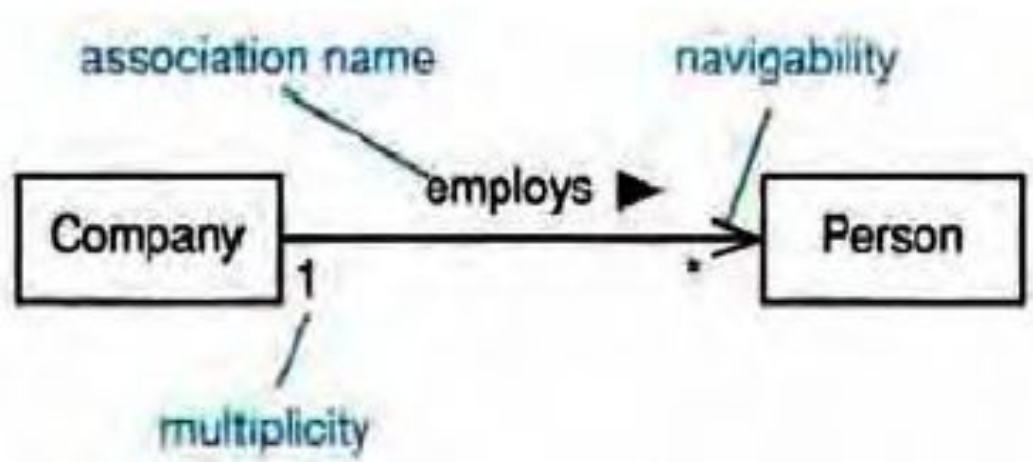


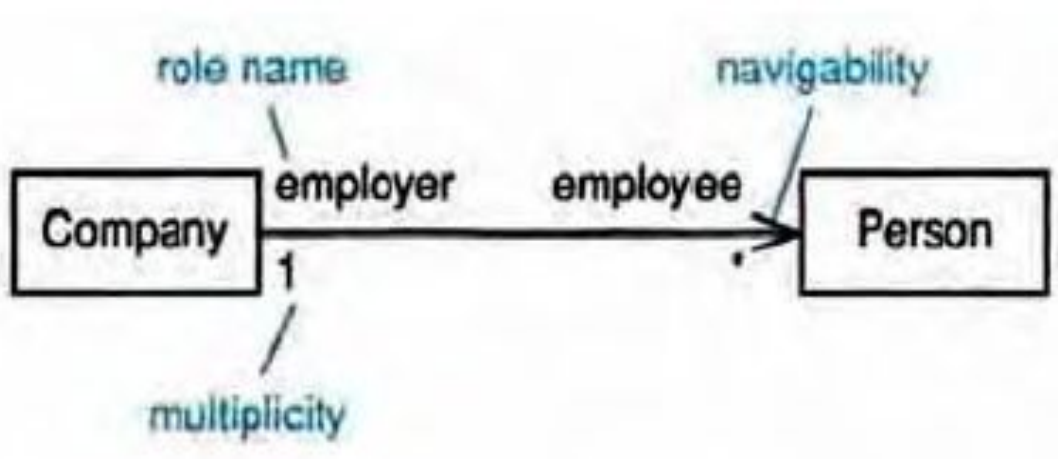
Association Relationships (Cont'd)

- A class can have a self association.

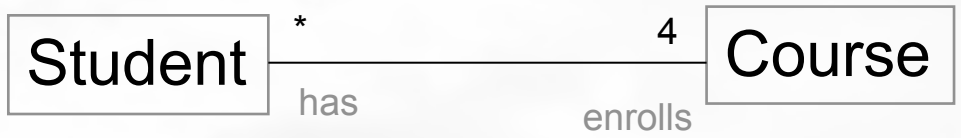


Association examples





Association: Model to Implementation




```
Class Student {
    Course enrolls[4];
}

Class Course {
     Student have[];
}
```

Association - Aggregation

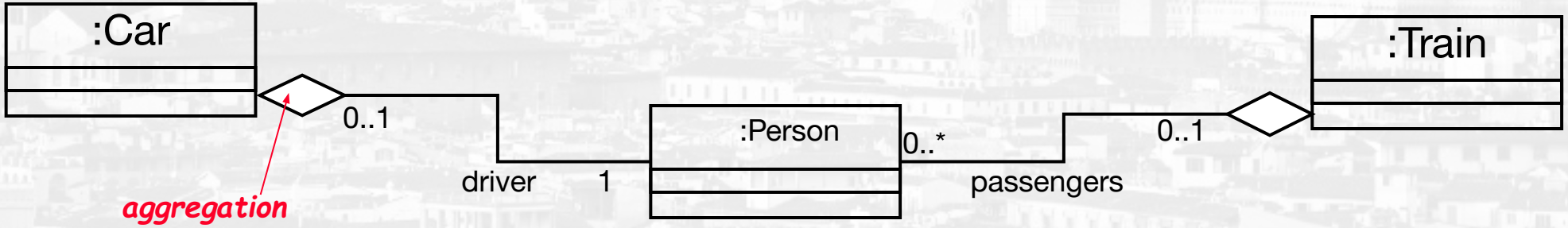
- A special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.

- **Aggregation** implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

Aggregation and Composition

Aggregation

- This is the "Has-a" or "Whole/part" relationship



115

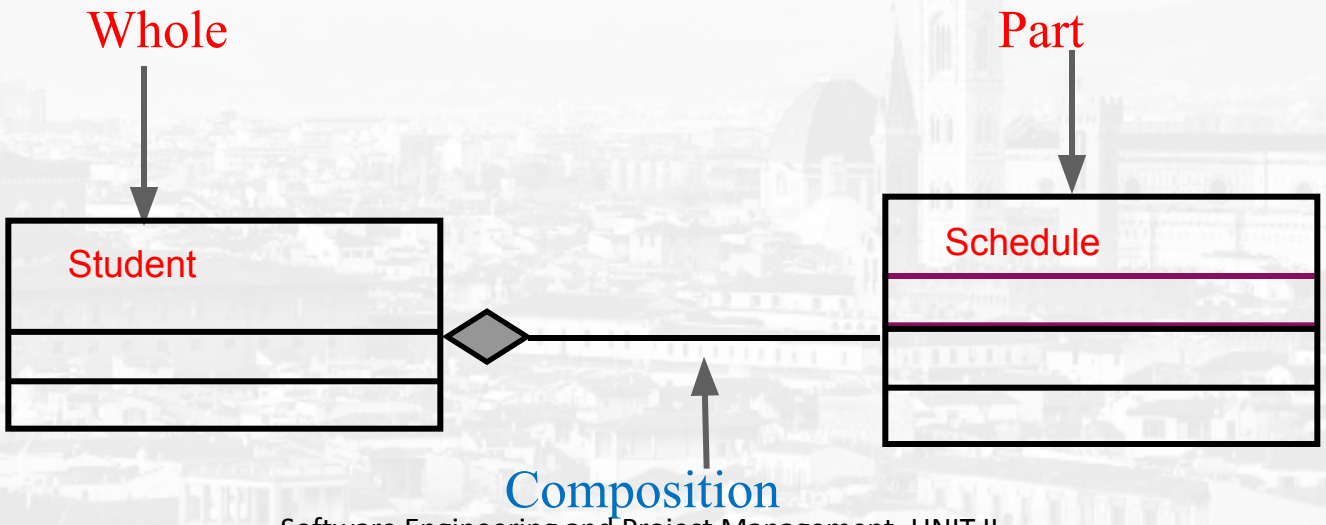
Aggregation and Composition

- **Composition**
  - **Strong form of aggregation that implies ownership:**
    - **if the whole is removed from the model, so is the part.**
    - **the whole is responsible for the disposition of its parts**

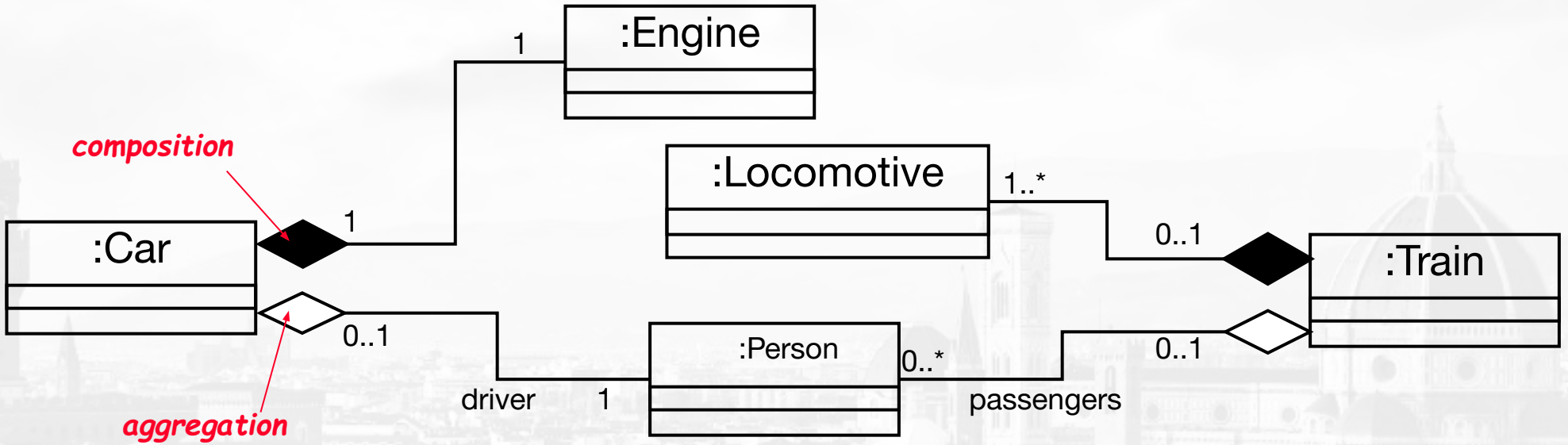
Association - Composition

- A form of aggregation with strong ownership and coincident lifetimes
  - The parts cannot survive the whole/aggregate

Composition implies a relationship where the child cannot exist independent of the parent.

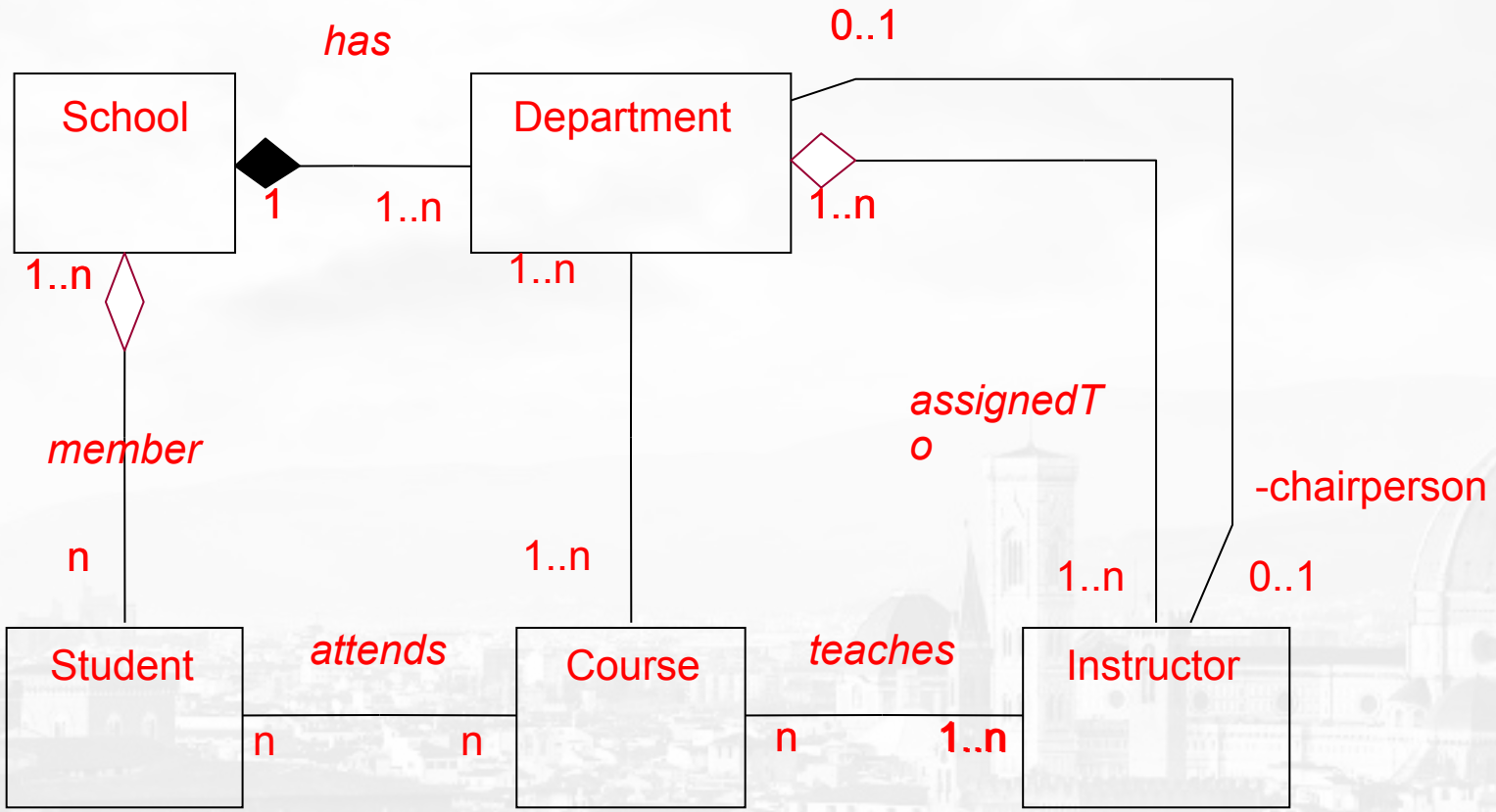


Aggregation and Composition

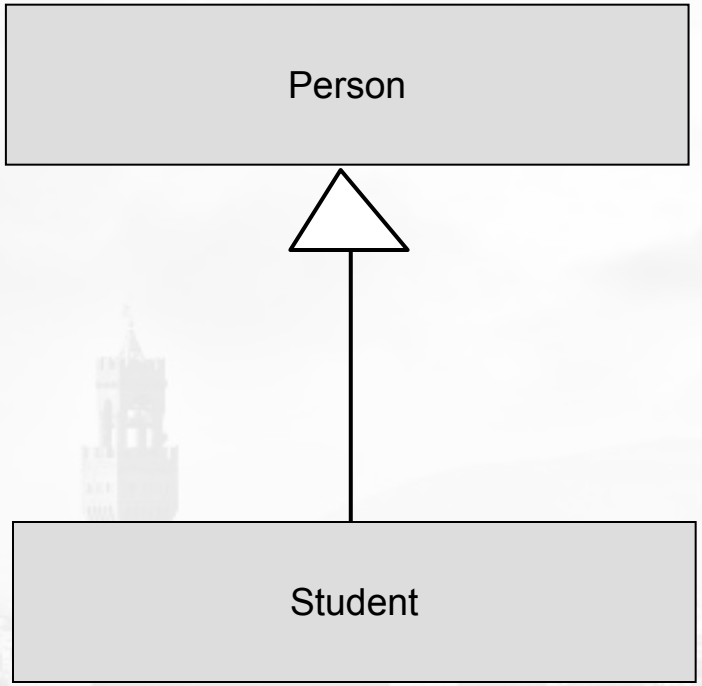


118

Example


has
0..1
School
Department
1
1..n
1..n
1..n
1..n
member
assignedTo
-chairperson
n
1..n
1..n
0..1
Student
attends
Course
teaches
Instructor
n
n
n
1..n

Generalization Relationships



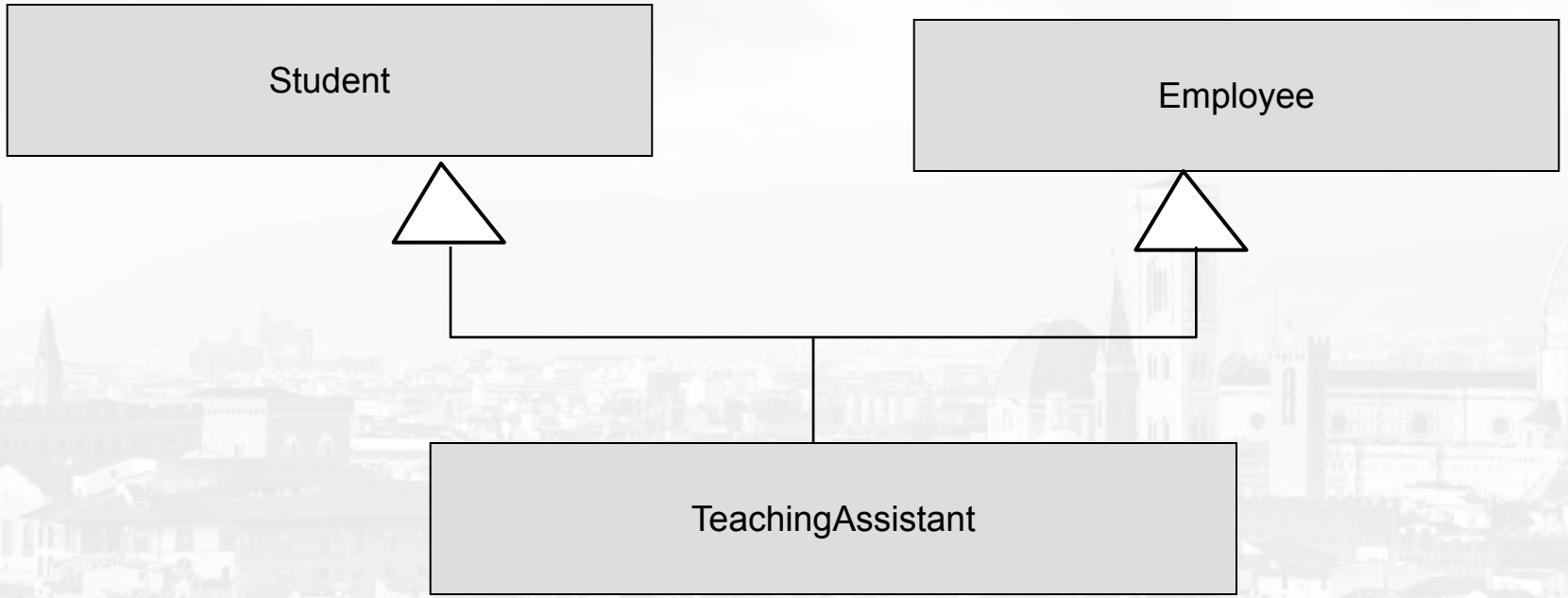
- A *generalization* connects a subclass to its superclass.

It denotes an

- Inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

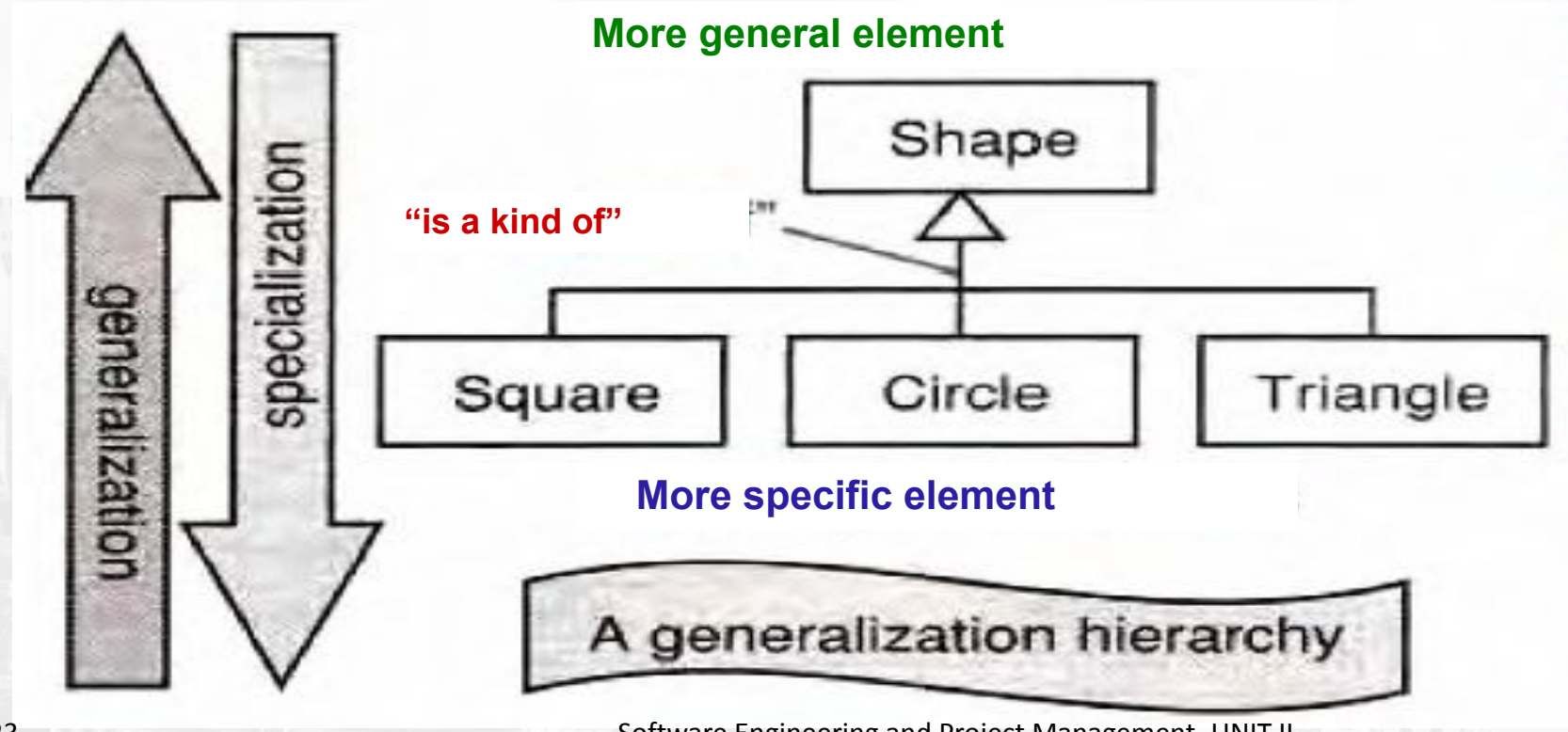
Generalization Relationships (Cont'd)

- UML permits a class to inherit from multiple superclasses, although some programming languages (*e.g.,* Java) do not permit multiple inheritance.



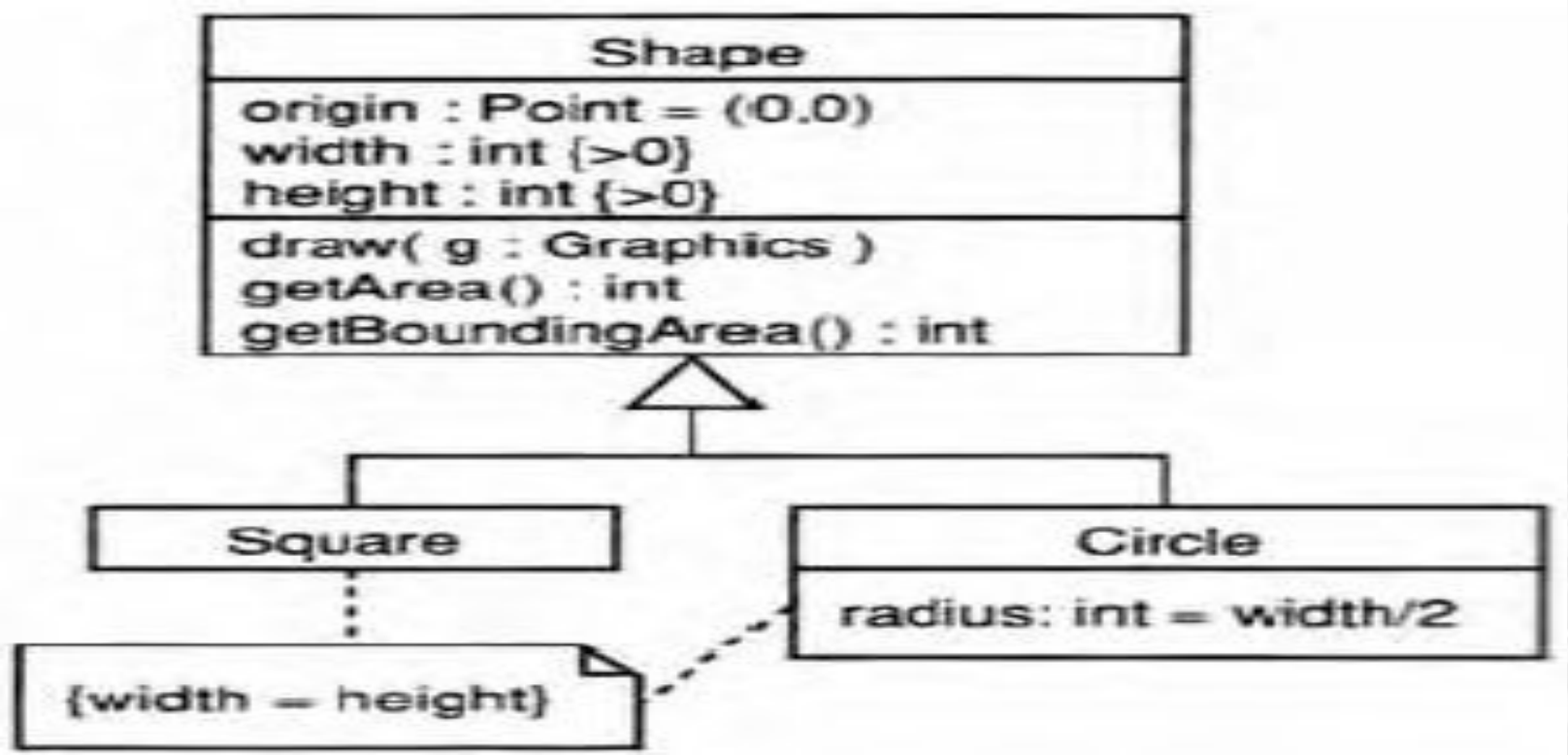
Generalization/Specialization

- Generalization hierarchies may be created by generalizing from specific things or by
- specializing from general things.

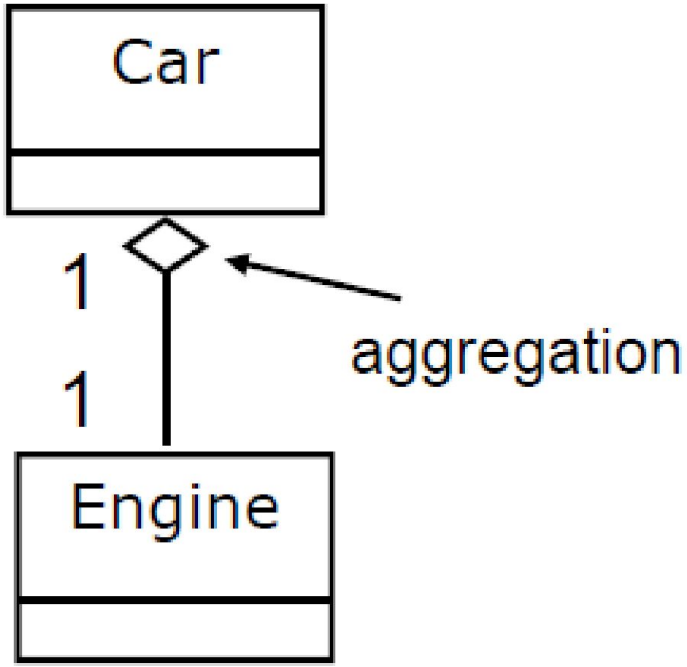


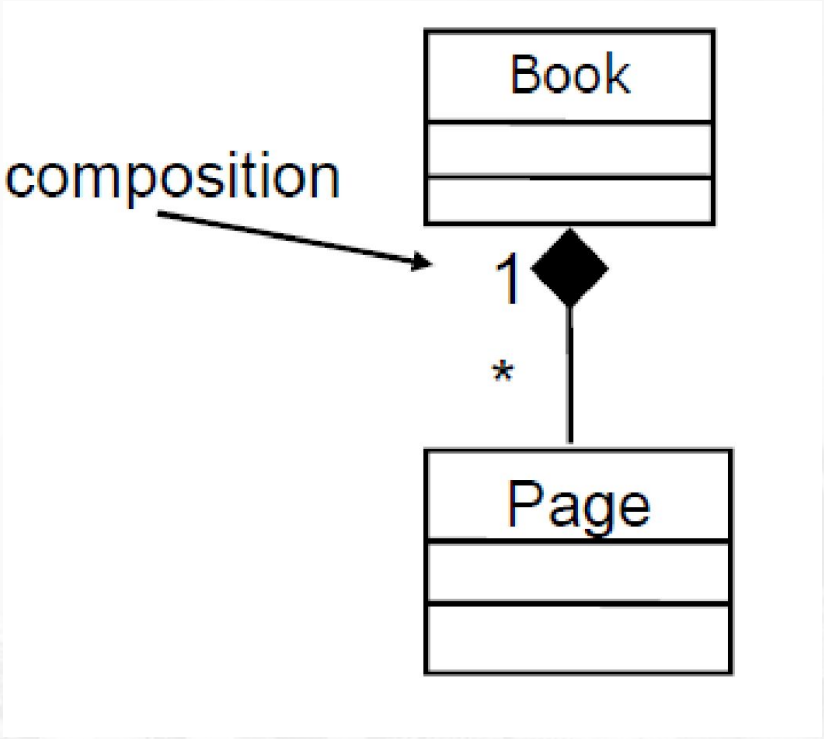
Inheritance

- Class inheritance is implicit in a generalization relationship between classes.
- Subclasses inherit attributes, associations, & operations from the superclass



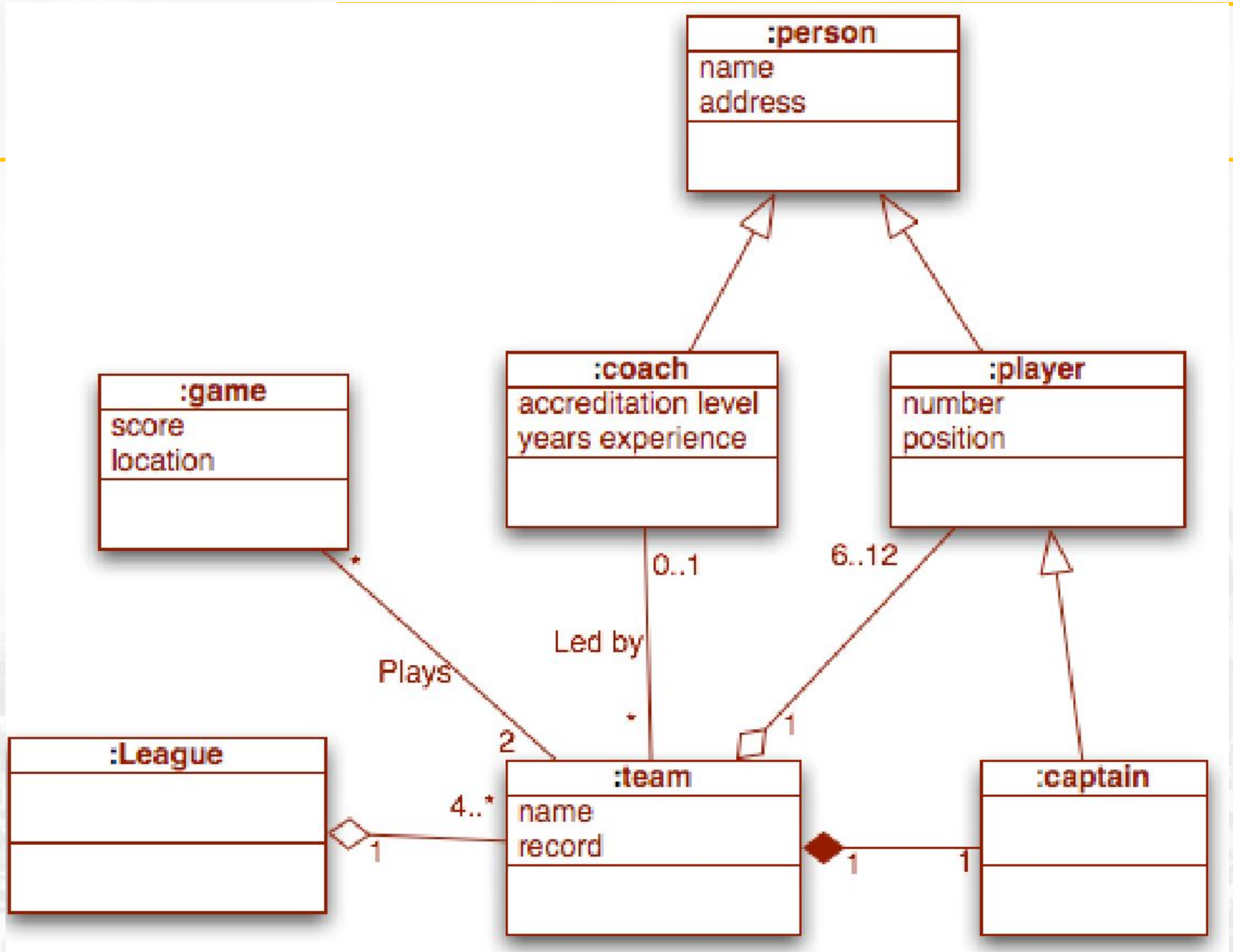
Another Example



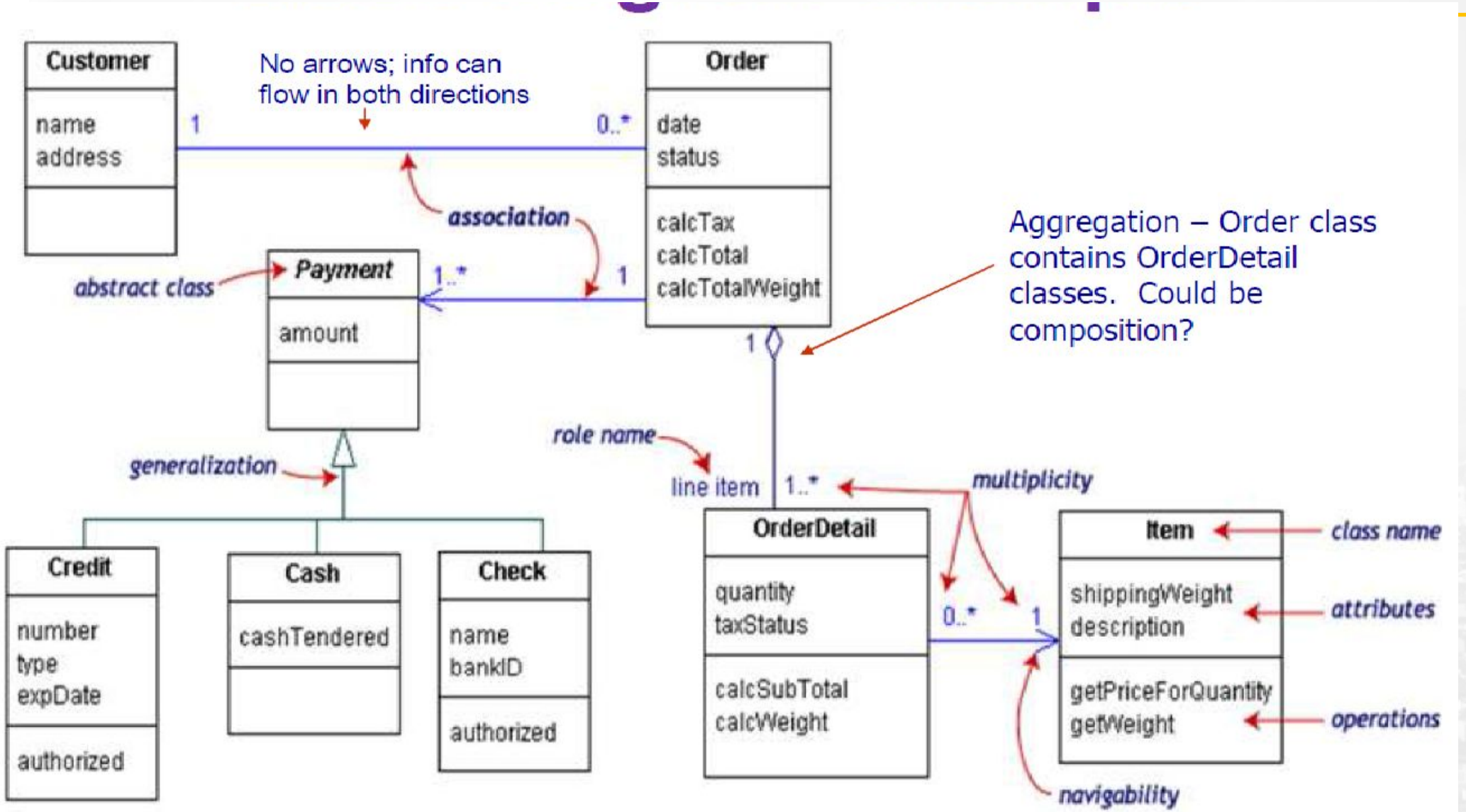


Example

- Draw a UML Class Diagram representing the following elements from the problem domain for a hockey league.
- A hockey league is made up of at least four hockey teams.
- Each hockey team is composed of six to twelve players, and one player captains the team.
- A team has a name and a record. Players have a number and a position. Hockey teams play games against each other. Each game has a score and a location. Teams are sometimes lead by a coach.
- A coach has a level of accreditation and a number of years of experience, and can coach multiple teams. Coaches and players are people, and people have names and addresses.


:person
name
address
:coach
accreditation level
years experience
:player
number
position
:game
score
location
0..1
6..12
Led by
Plays
*
*
2
1
:League
:team
name
record
:captain
4..*
1
1
1

Example : Order Details



Benefits of class diagrams

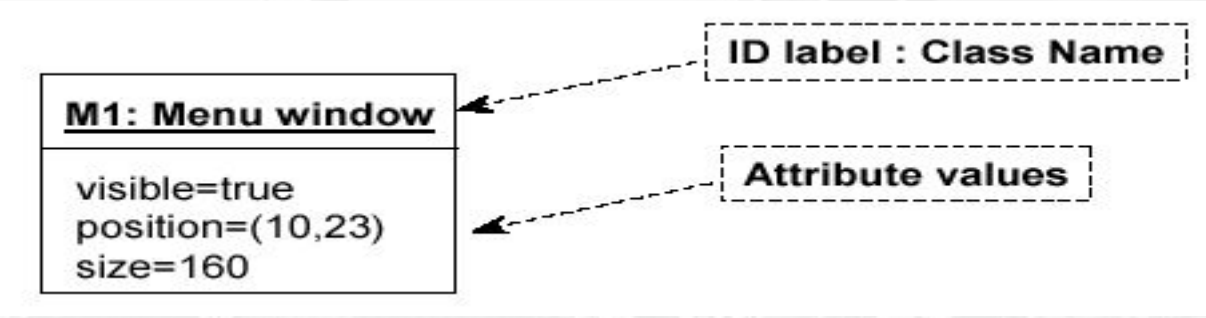
- To understand the general overview of plan of an application.
- Illustrate data models for information systems, no matter how simple or complex.
- Visually express any specific needs of a system
- Create detailed charts that highlight any specific code needed to be programmed and implemented to the described structure.
- Provide an implementation-independent description of types used in a system that are later passed between its components.

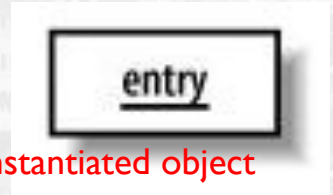
Object Diagram

- Derived from class diagram
- Shows a set of objects & their relationship
- Represents a static view of the system
- An object diagram is a diagram that shows a set of objects and their relationships at a **point in time.**
- Object diagrams help you capture the logical view of your model

Object Diagrams

- An object is shown with a rectangle and the title is underlined
- Format is
  - Instance name : Class name
  - Attributes and Values



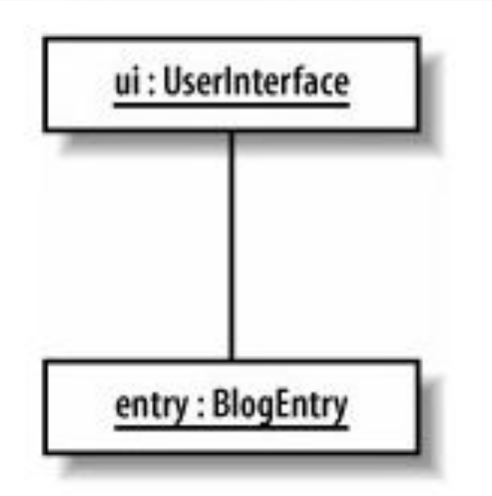


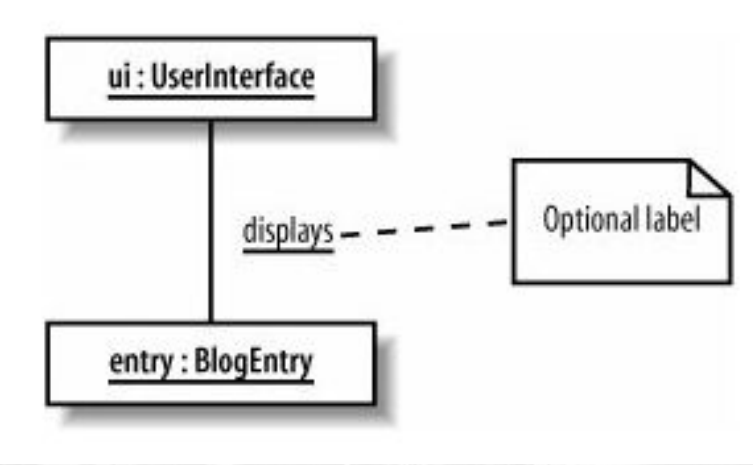
instantiated object

entry object is an instance of the BlogEntry class

Links

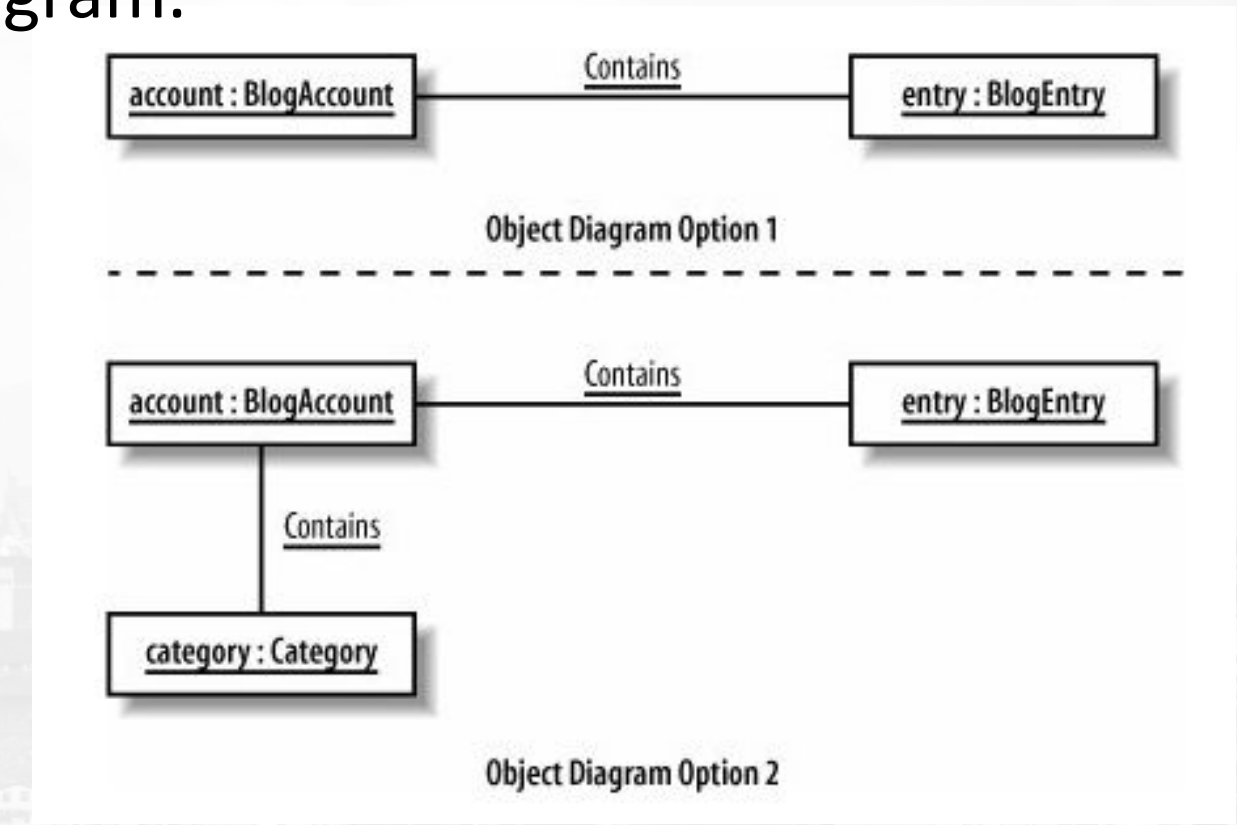
- To show how objects work together, links shows that two objects can communicate with each other
- There must be corresponding association between the classes in the class diagram
- Can add label that indicates the purpose of the link,



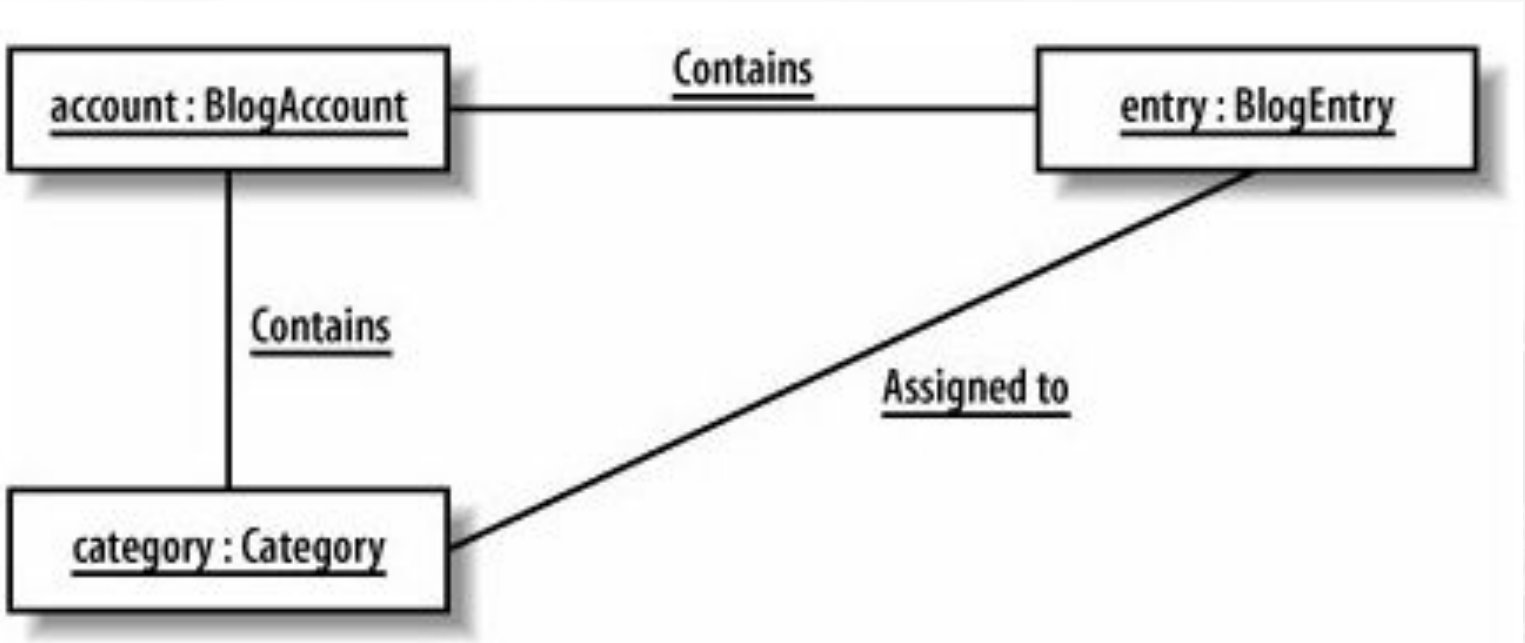


Links and Constraints

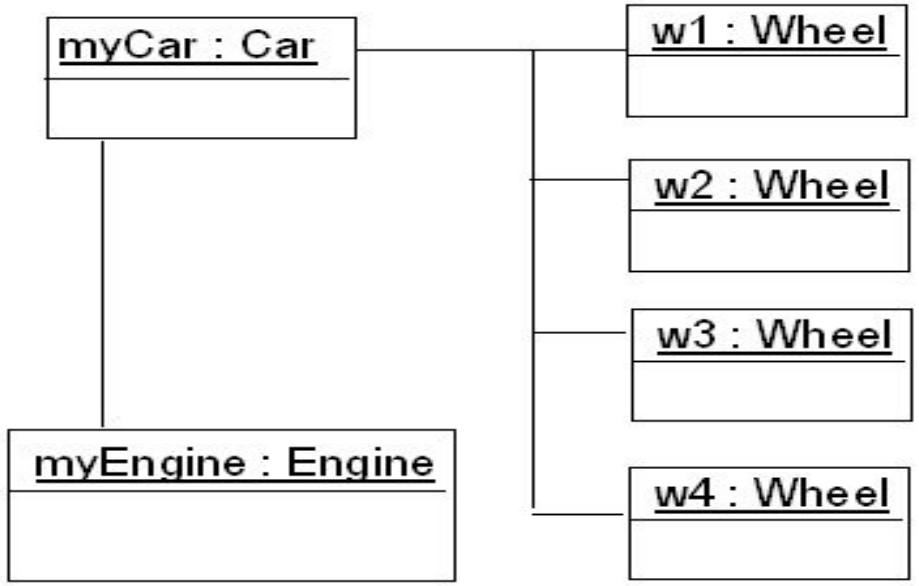
- Links must keep to the rules (constraints) given in class diagram.

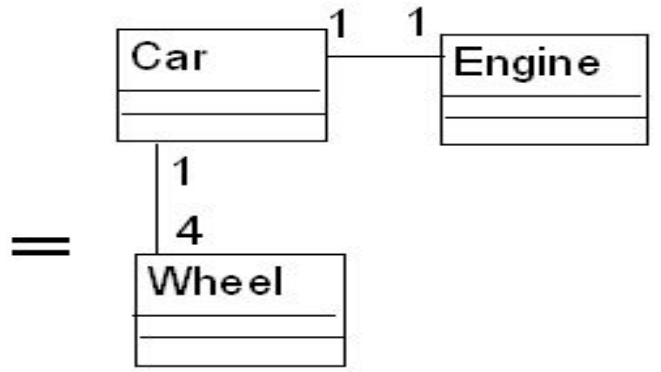


Both diagrams are valid.


account : BlogAccount
Contains
entry : BlogEntry
Contains
Assigned to
category : Category

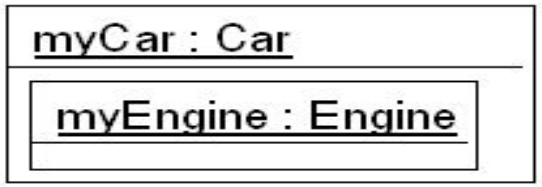
An Object Diagram example





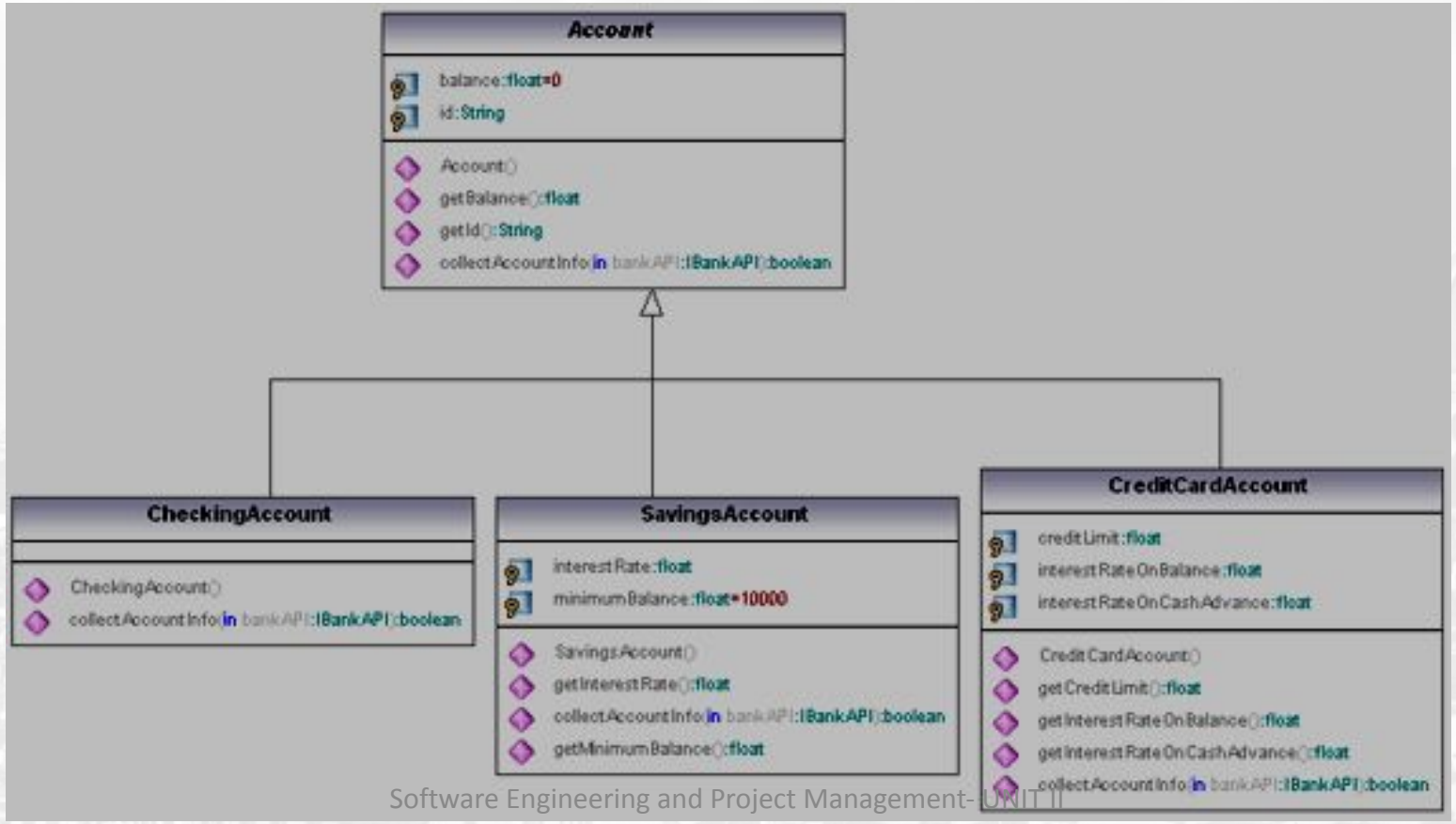
Composition with :

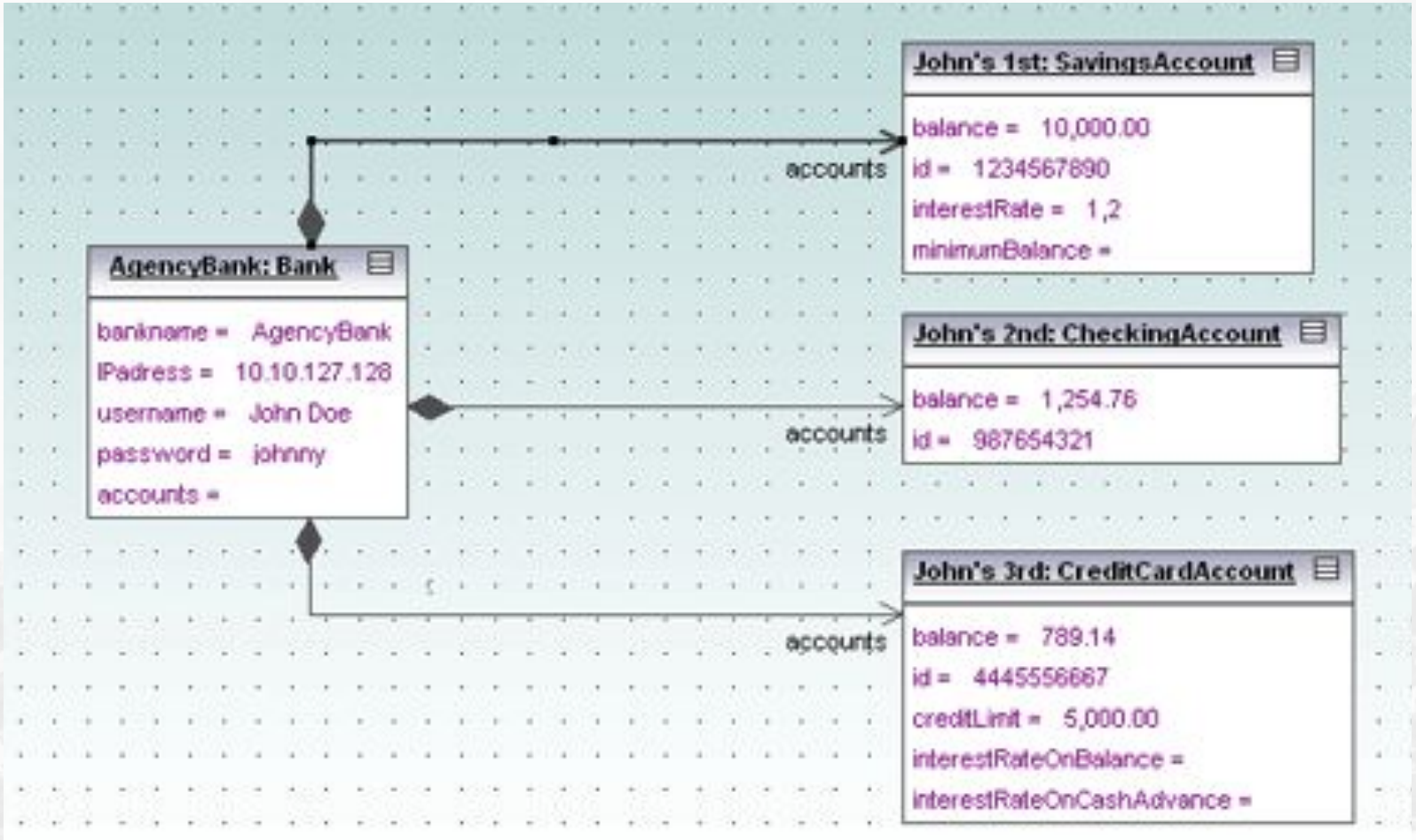
Or containment :



Eg.1

- From the given class diagram draw an object diagram for John's accounts

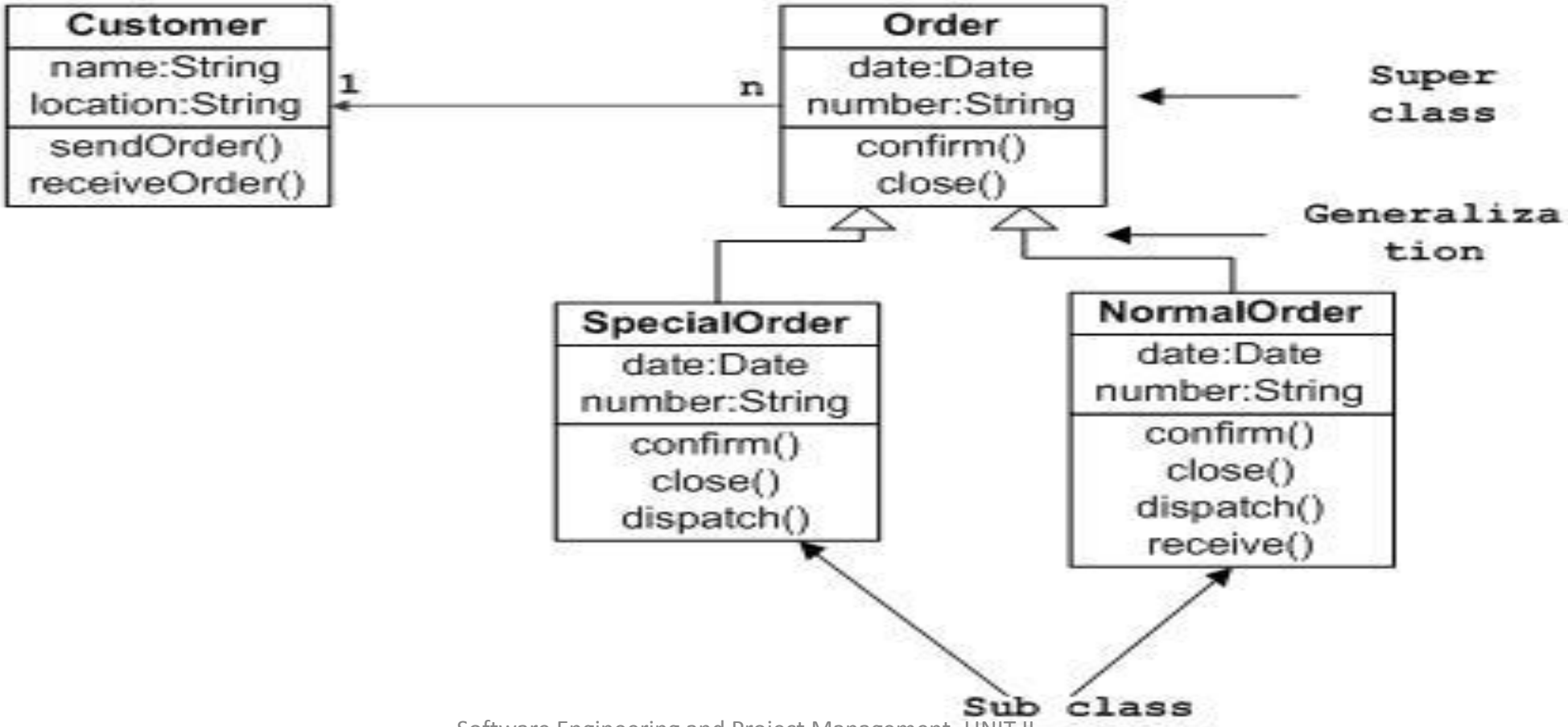



John's 1st: SavingsAccount
balance = 10,000.00
id = 1234567890
interestRate = 1,2
minimumBalance =
accounts
AgencyBank: Bank
bankname = AgencyBank
IPadress = 10.10.127.128
username = John Doe
password = johnny
accounts =
John's 2nd: CheckingAccount
balance = 1,254.76
id = 987654321
accounts
John's 3rd: CreditCardAccount
balance = 789.14
id = 4445556667
creditLimit = 5,000.00
interestRateOnBalance =
interestRateOnCashAdvance =
accounts

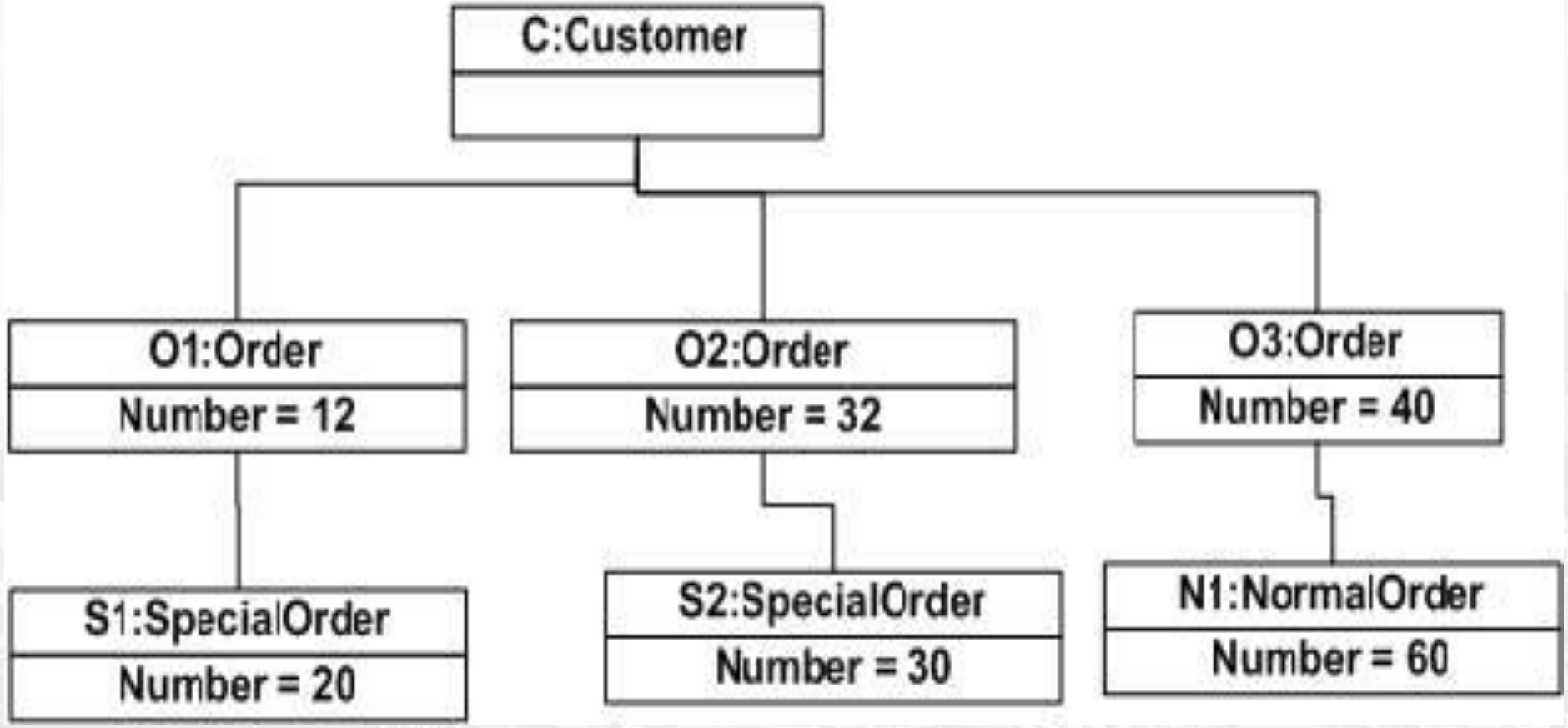
Eg. 2

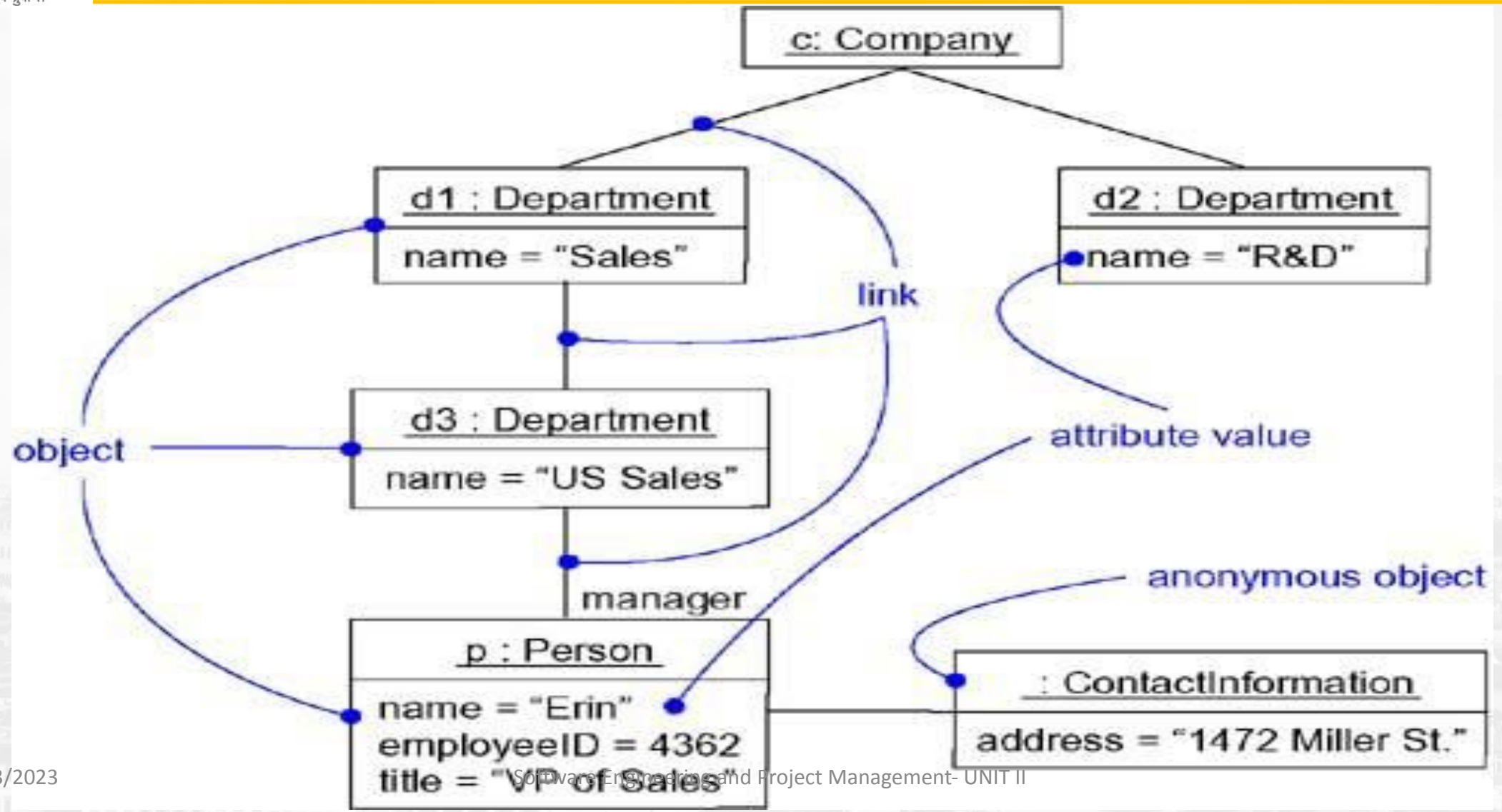
- Draw Object diagram for the *Order management system* . It has the following objects
  - Customer
  - Order
  - SpecialOrder
  - NormalOrder
- Customer object (C) is associated with three order objects (O1, O2 and O3). These order objects are associated with special order and normal order objects (S1, S2 and N1). The customer is having the following three orders with different numbers (12, 32 and 40) for the particular time considered.

Sample Class Diagram

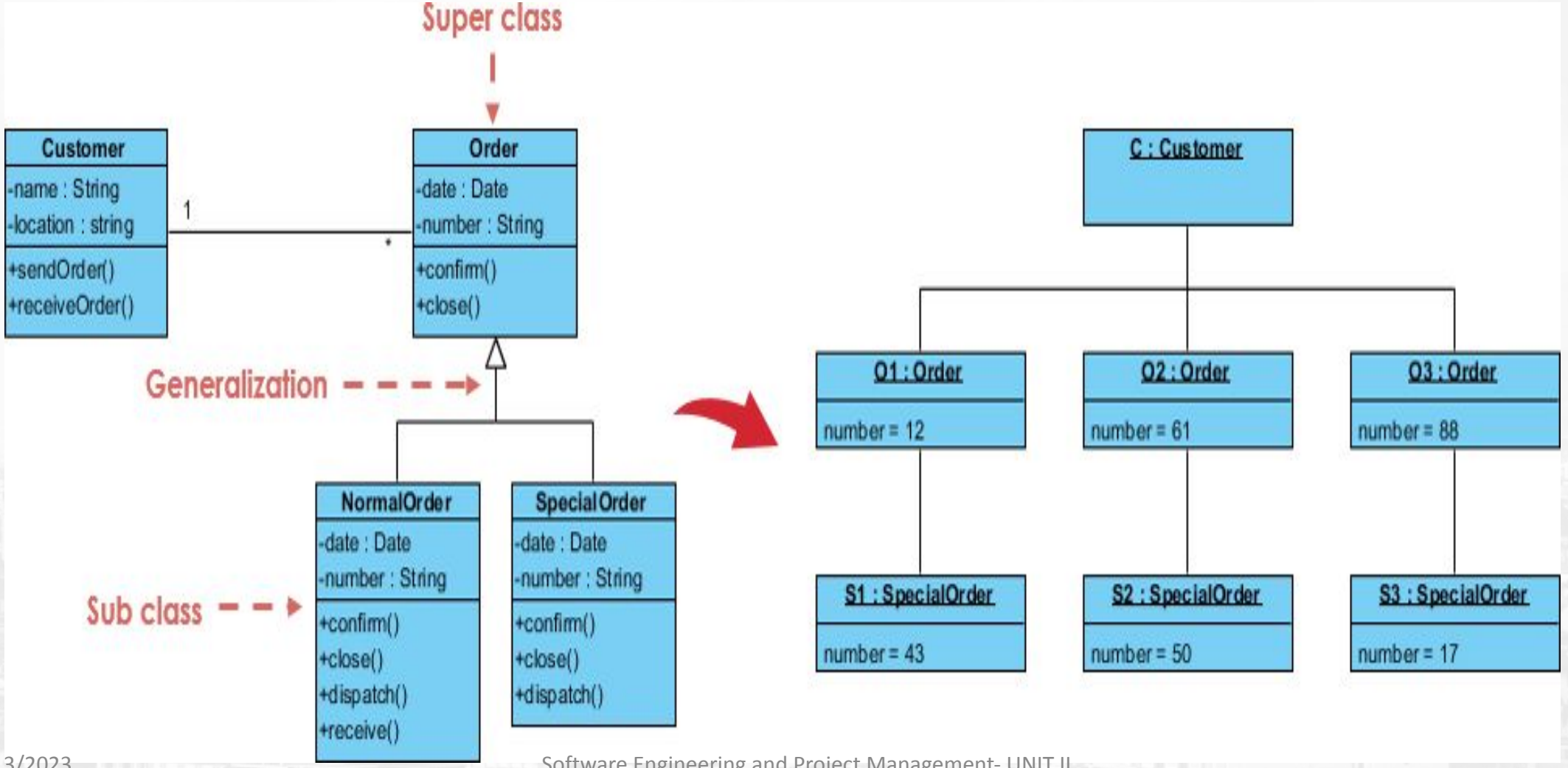


Object diagram of an order management system

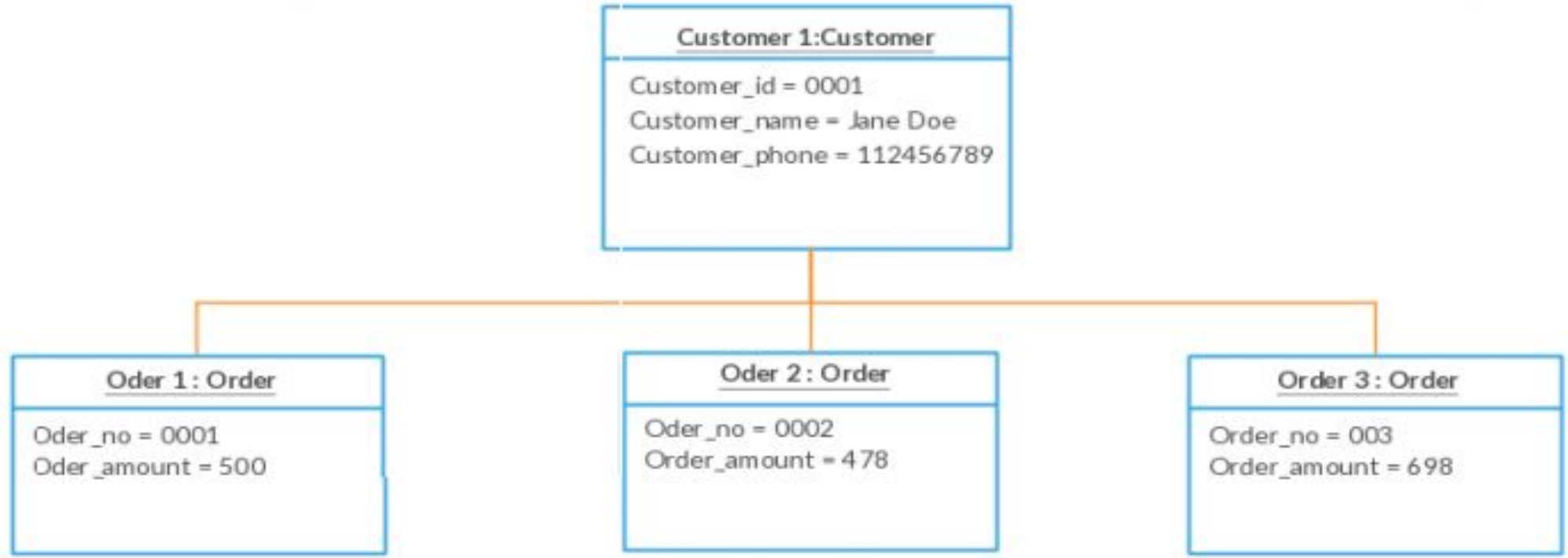



c: Company
d1 : Department
name = "Sales"
d2 : Department
name = "R&D"
link
d3 : Department
name = "US Sales"
object
attribute value
manager
anonymous object
p : Person
name = "Erin"
employeeID = 4362
title = "VP of Sales"
: ContactInformation
address = "1472 Miller St."

Class to Object Diagram Example - Order System

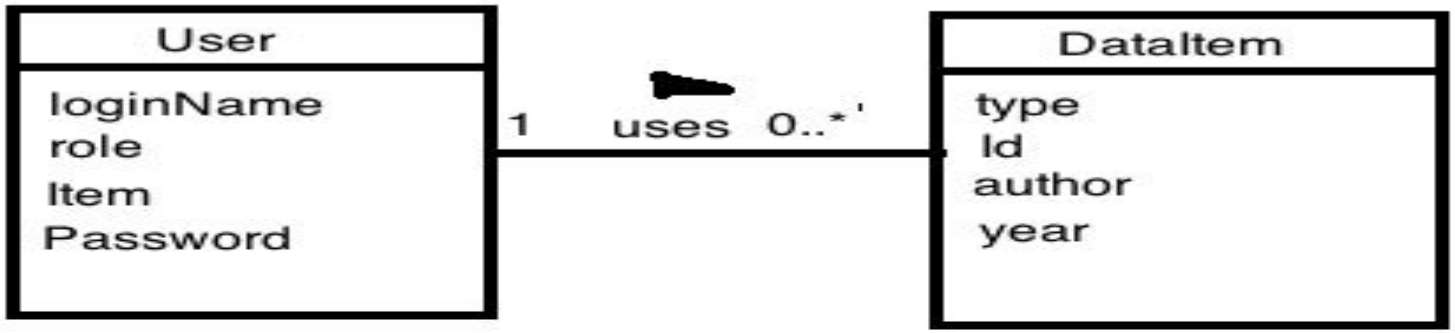


Example contd..

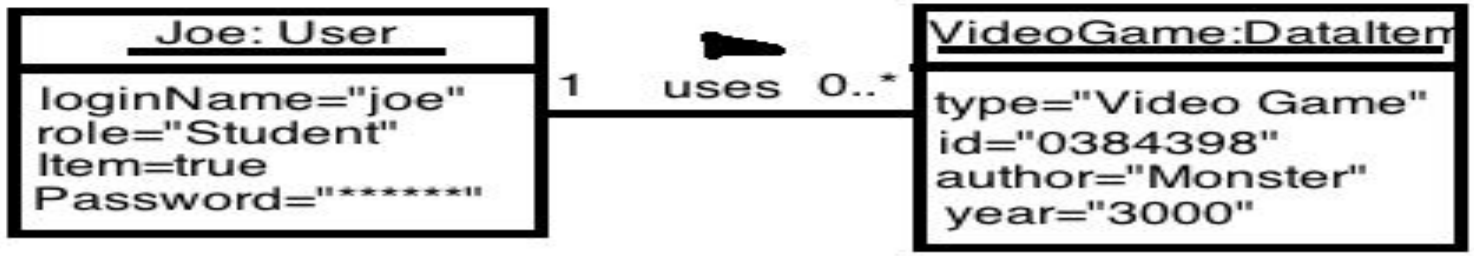


Example contd..

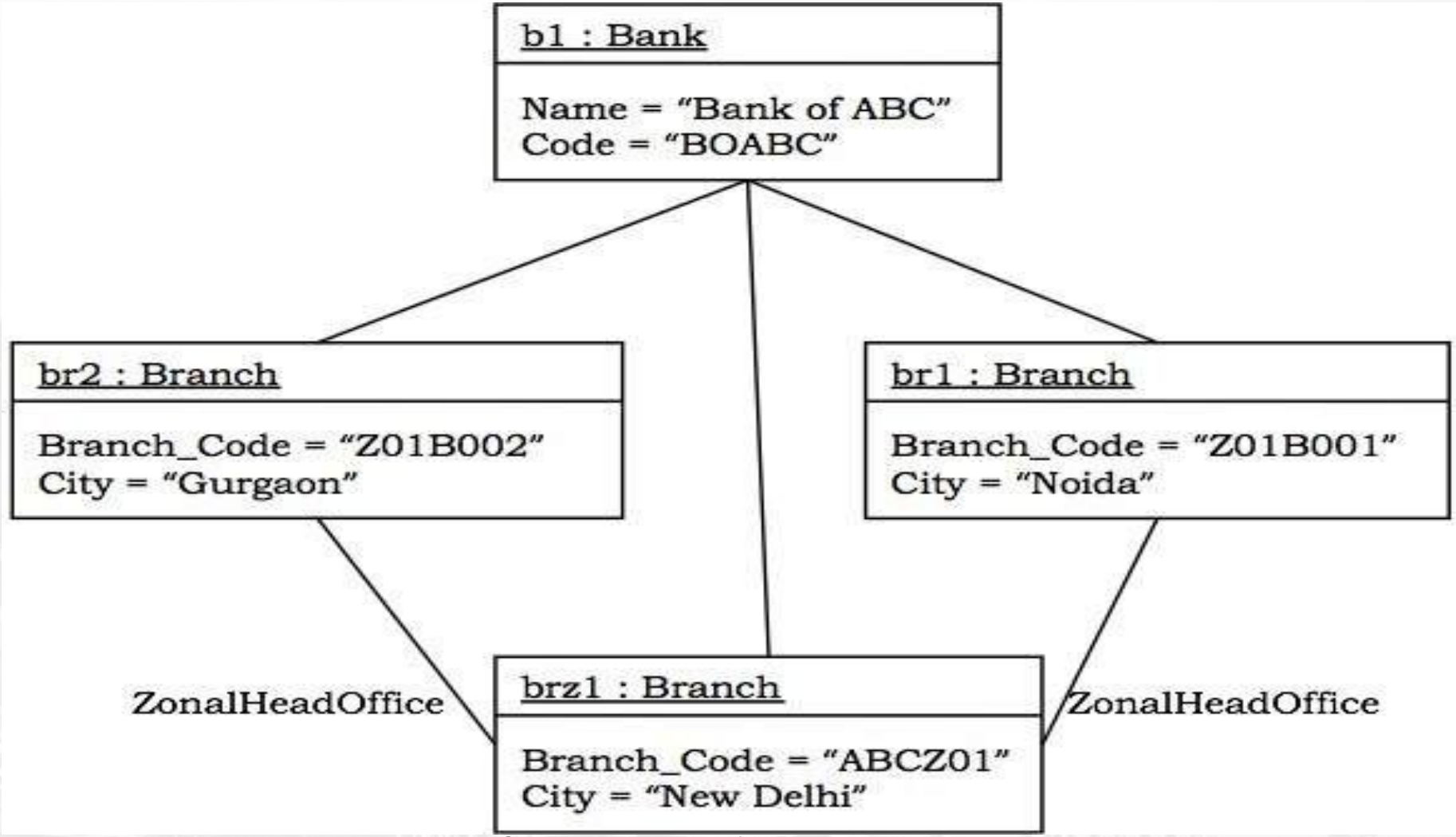
Class Diagrams



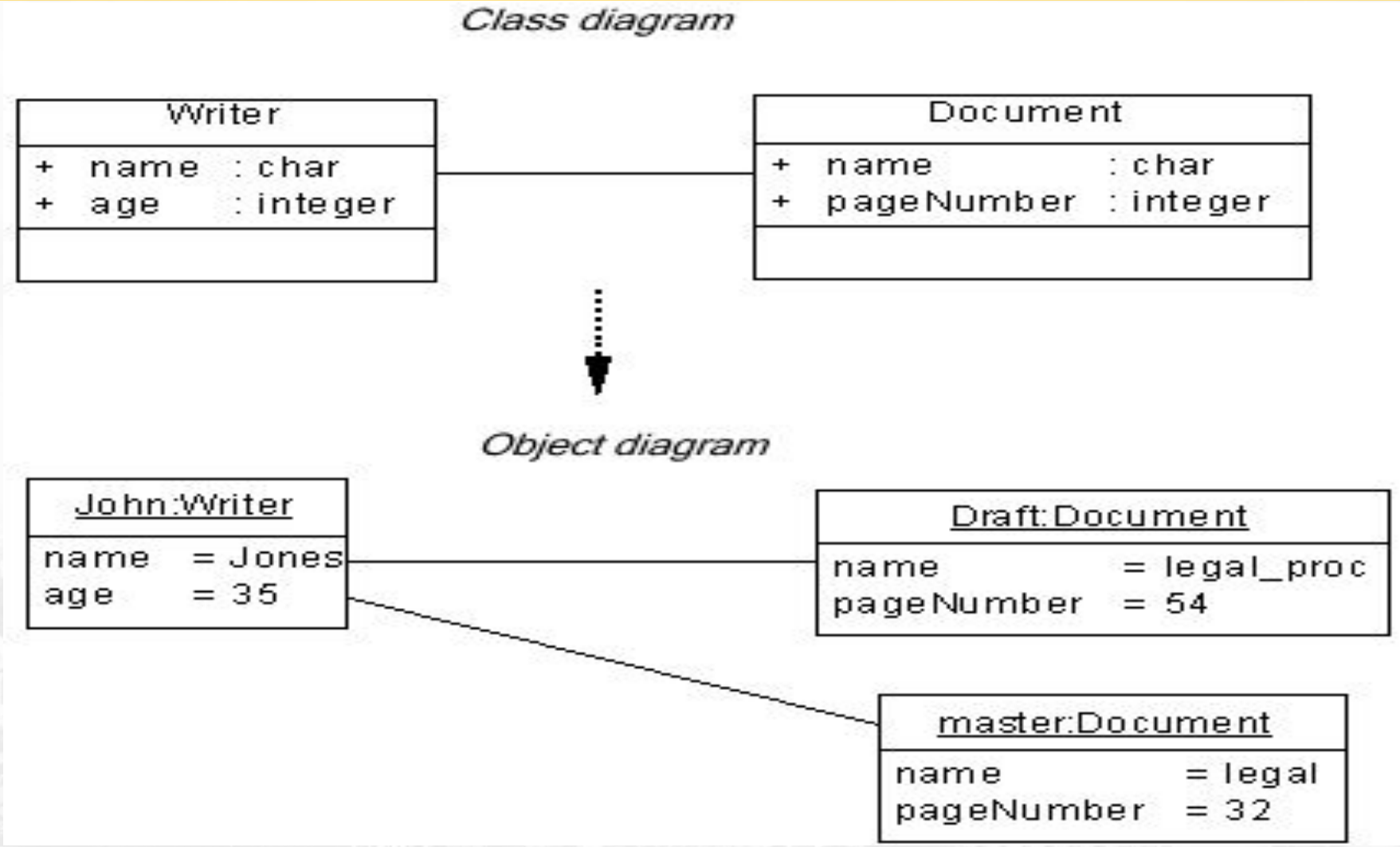
ObjectDiagrams



Example (contd..)



Example contd..

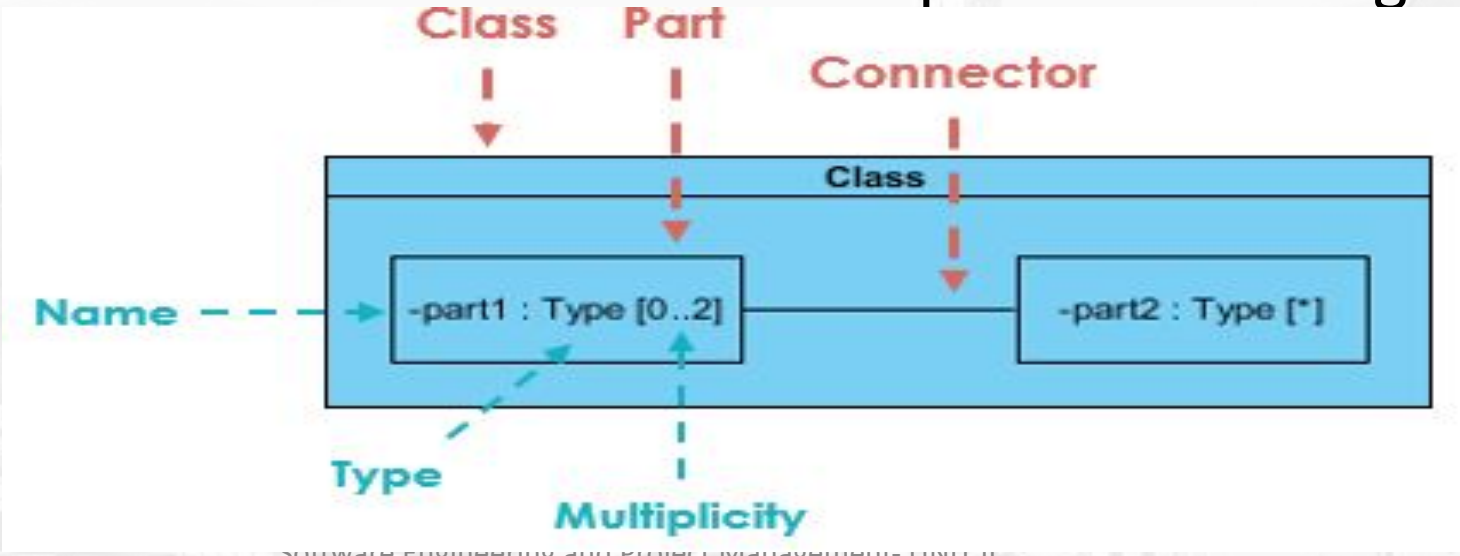


Composite Structure Diagrams

- A composite structure diagram is a UML structural diagram that contains classes, interfaces, packages, and their relationships, and that provides a logical view of all, or part of a software system.
- It shows the internal structure (including parts and connectors) of a structured classifier or collaboration.
- A composite structure diagram performs a similar role to a class diagram, but allows you to go into further detail in describing the internal structure of multiple classes and showing the interactions between them.

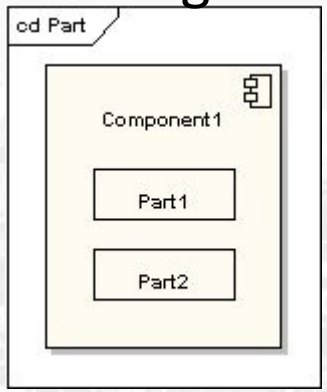
Representation

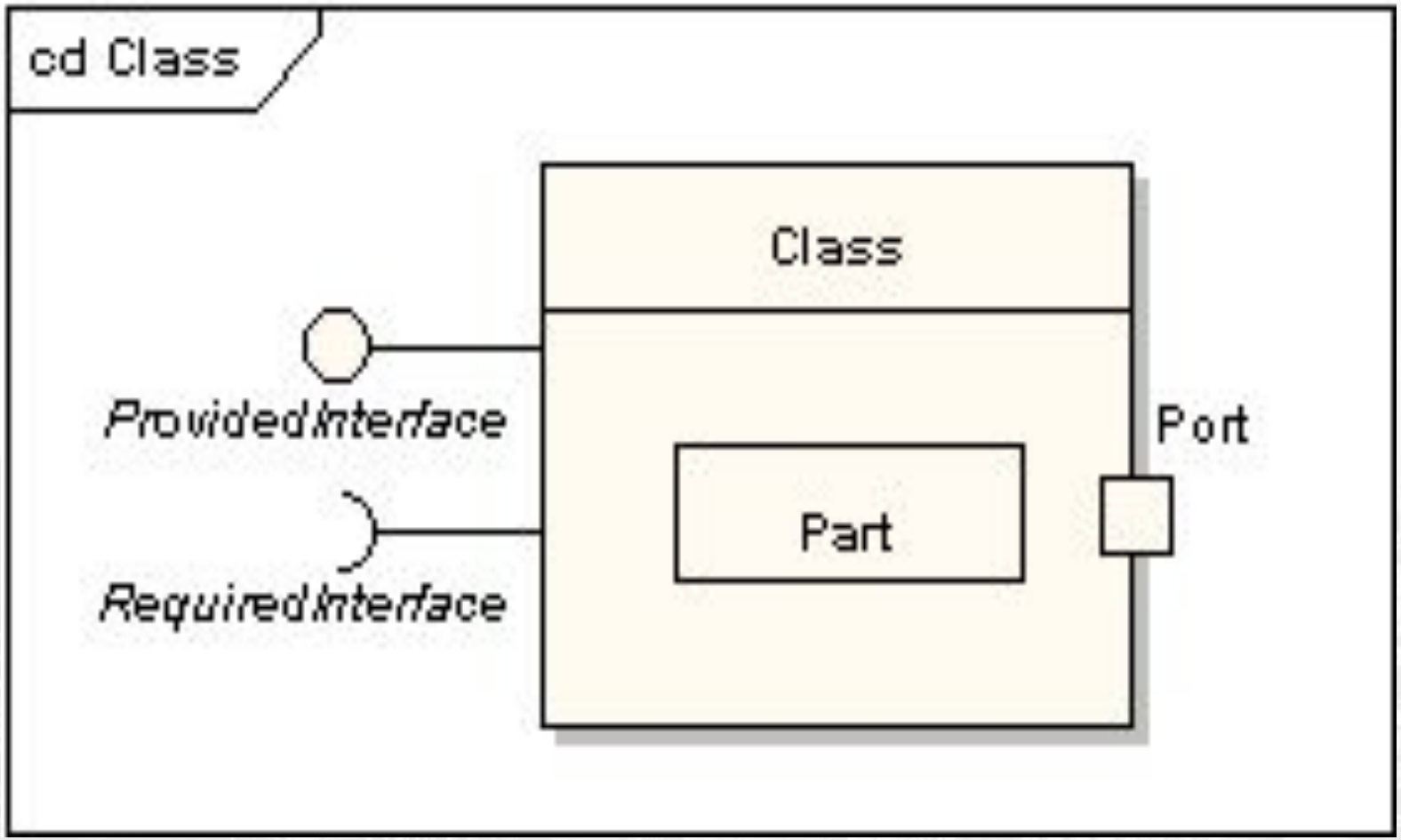
- Composite Structure Diagrams show the internal parts of a class.
- Parts are named: partName:partType[multiplicity]
- Aggregated classes are parts of a class but parts are not necessarily classes
- A part is any element that is used to make up the containing class.



Part

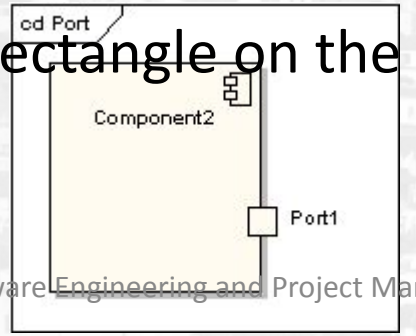
- A part is an element that represents a set of one or more instances which are owned by a containing classifier instance.
- If a diagram instance owned a set of graphical elements, then the graphical elements could be represented as parts.
- A part is shown as an unadorned rectangle contained within the body of a class or component element.




cd Class
Class
ProvidedInterface
RequiredInterface
Part
Port

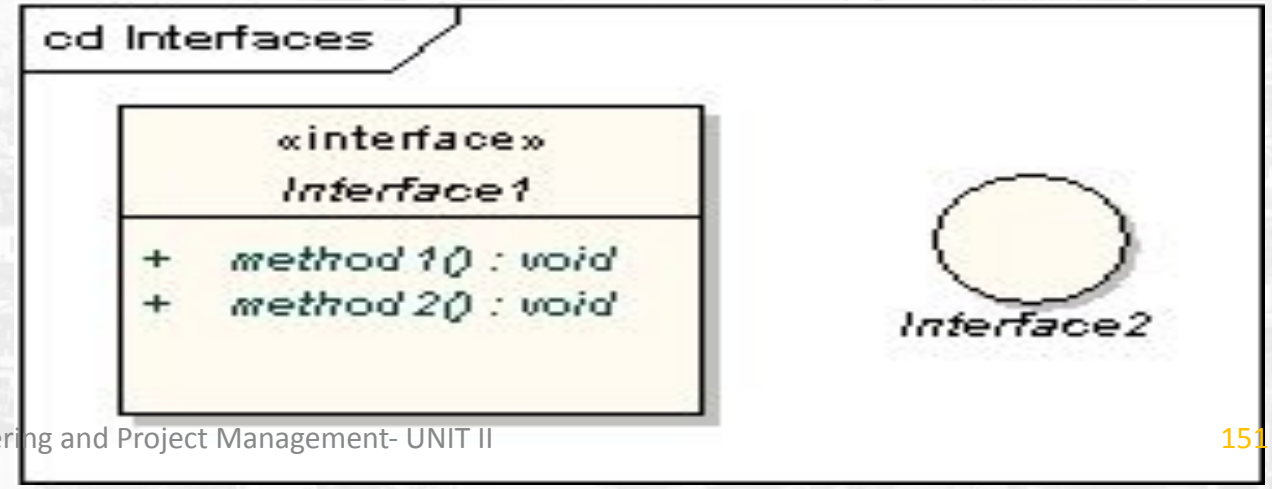
Port

- A port is a typed element that represents an externally visible part of a containing classifier instance.
- Ports define the interaction between a classifier and its environment.
- A port can appear on the boundary of a contained part, a class or a composite structure.
- A port may specify the services a classifier provides as well as the services that it requires of its environment.
- A port is shown as a named rectangle on the boundary edge of its owning classifier.



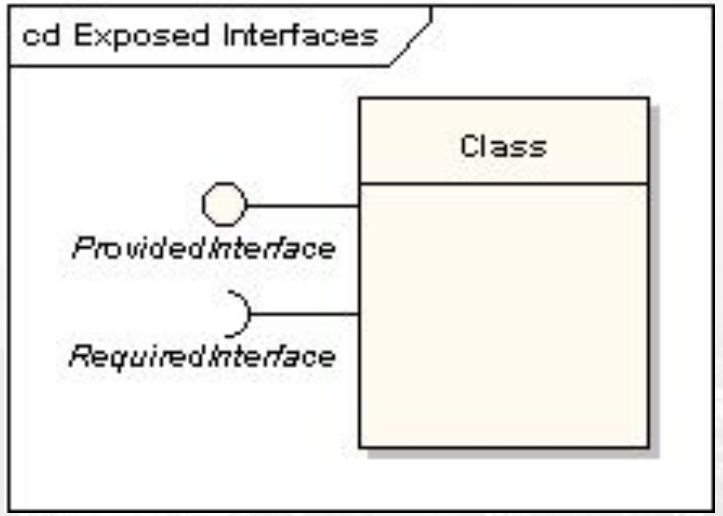
Interfaces

- All interface operations are public and abstract, and do not provide any default implementation.
- All interface attributes must be constants.
- An interface, when standing alone in a diagram, is either shown as a class element rectangle with the «interface» keyword and with its name italicized to denote it is abstract, or it is shown as a circle.



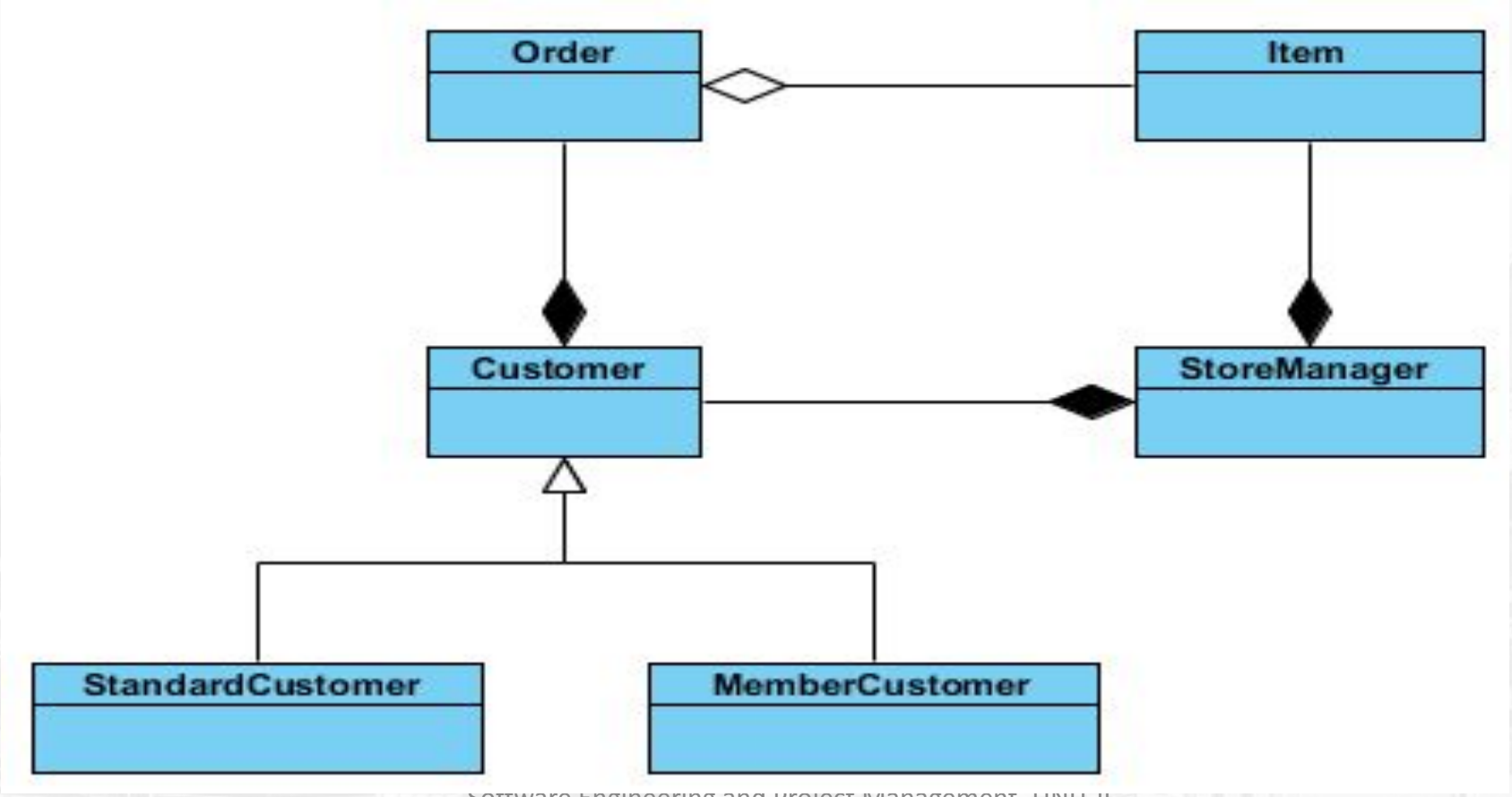
- An interface can be defined as either provided or required.

- A provided interface is shown as a "ball on a stick" attached to the edge of a classifier element.
- A required interface is shown as a "cup on a stick" attached to the edge of a classifier element.



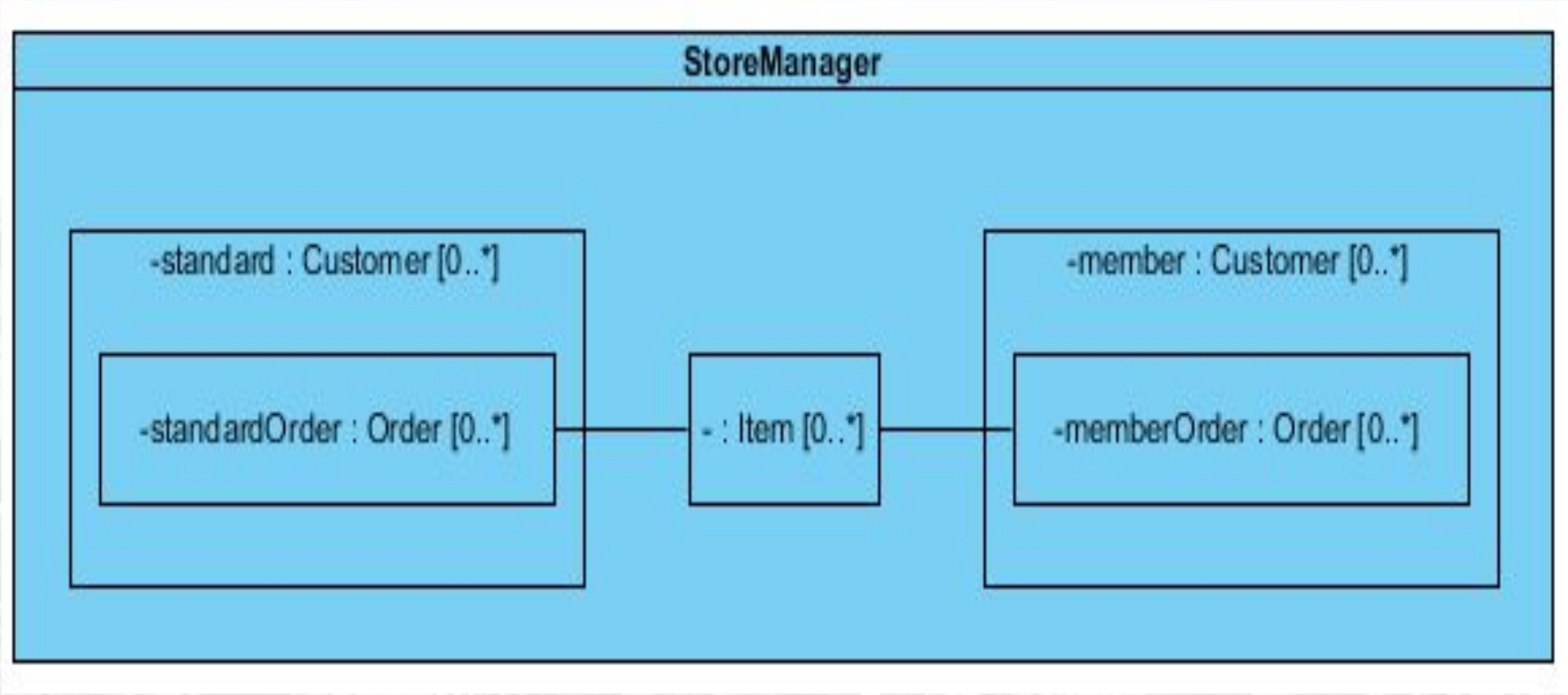
Example

- Online Store – Class Diagram

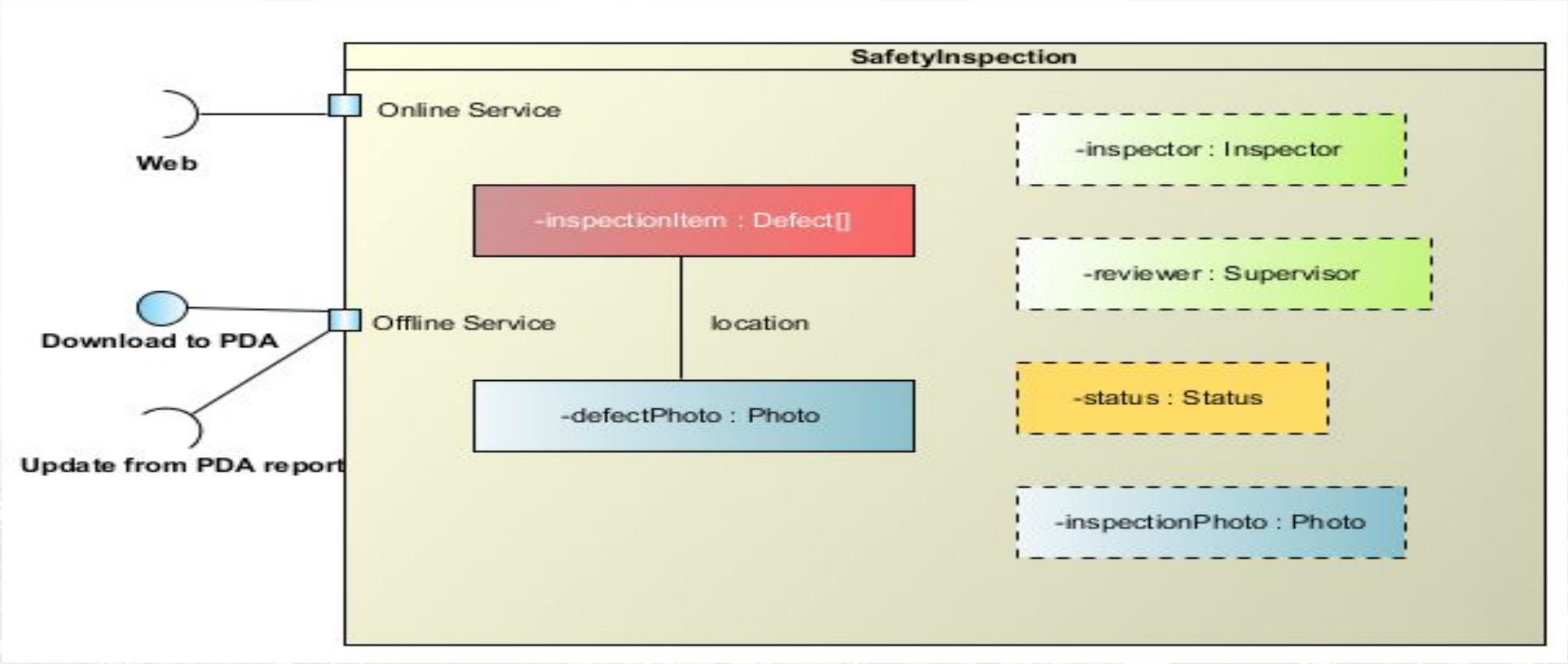


Example (contd..)

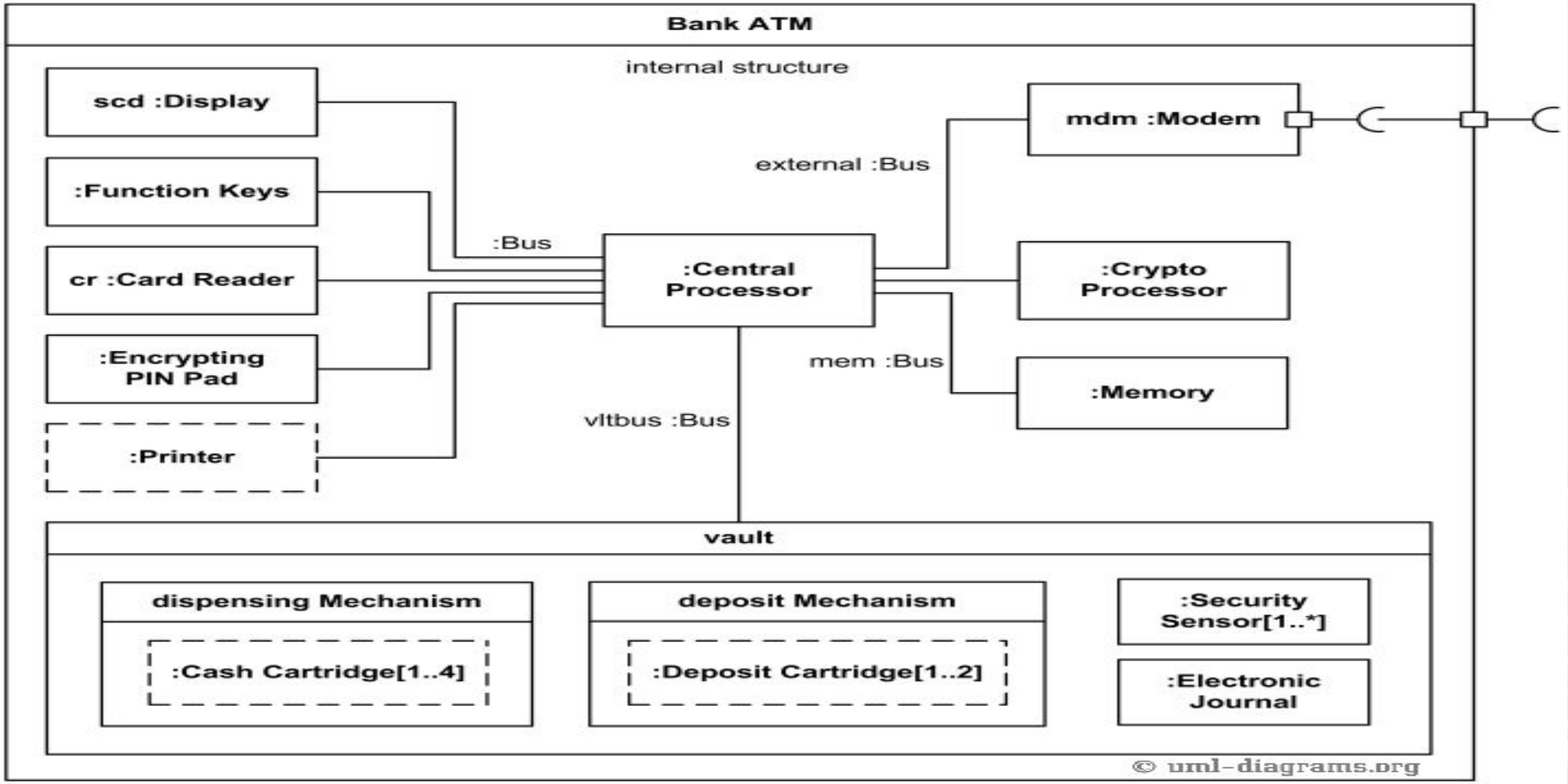
- Composite structure diagram for store manager



Example contd..



Example


Bank ATM
internal structure
scd :Display
:Function Keys
cr :Card Reader
:Encrypting PIN Pad
:Printer
:Bus
external :Bus
vltbus :Bus
mem :Bus
mdm :Modem
:Central Processor
:Crypto Processor
:Memory
vault
dispensing Mechanism
:Cash Cartridge[1..4]
deposit Mechanism
:Deposit Cartridge[1..2]
:Security Sensor[1..*]
:Electronic Journal
© uml-diagrams.org

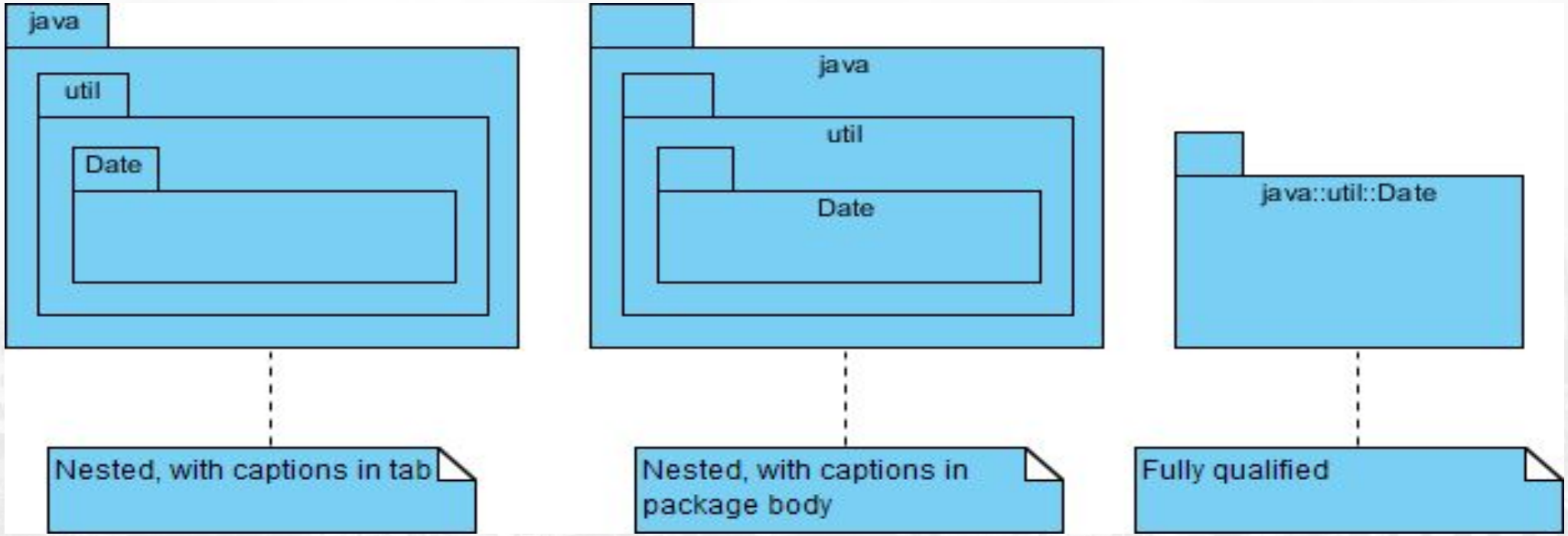
Package diagrams

- Package diagram is used to simplify complex class diagrams, you can group classes into packages. A package is a collection of logically related UML elements.
- Packages appear as rectangles with small tabs at the top.
- The package name is on the tab or inside the rectangle.
- The dotted arrows are dependencies.
- One package depends on another if changes in the other could possibly force changes in the first.

Package diagrams

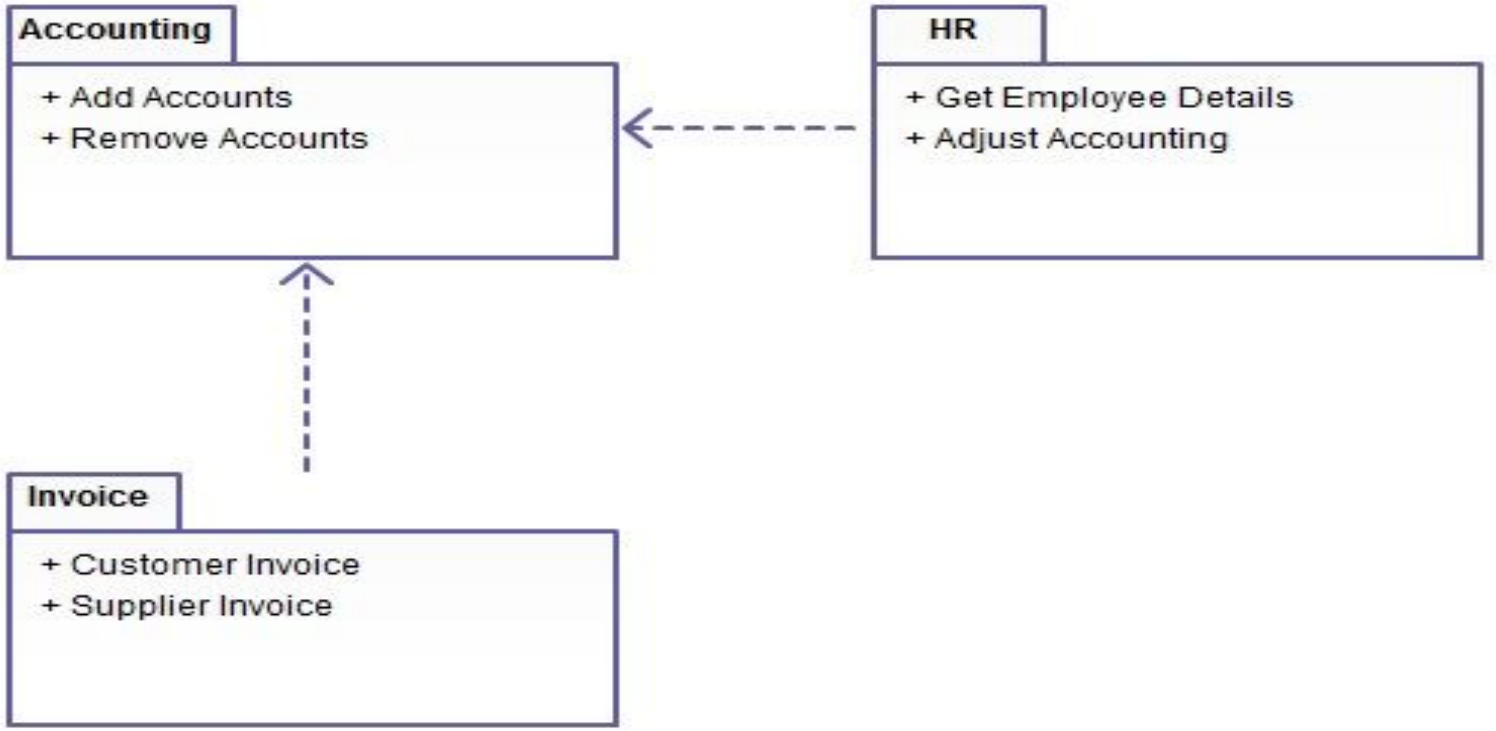
- Purpose of Package Diagrams
- Package diagrams are used to structure high level system elements. Packages are used for organizing large system which contains diagrams, documents and other key deliverables.
- Package Diagram can be used to simplify complex class diagrams, it can group classes into packages.
- Packages are depicted as file folders and can be used on any of the UML diagrams.

- Packages can be represented by the notations with some examples shown below:

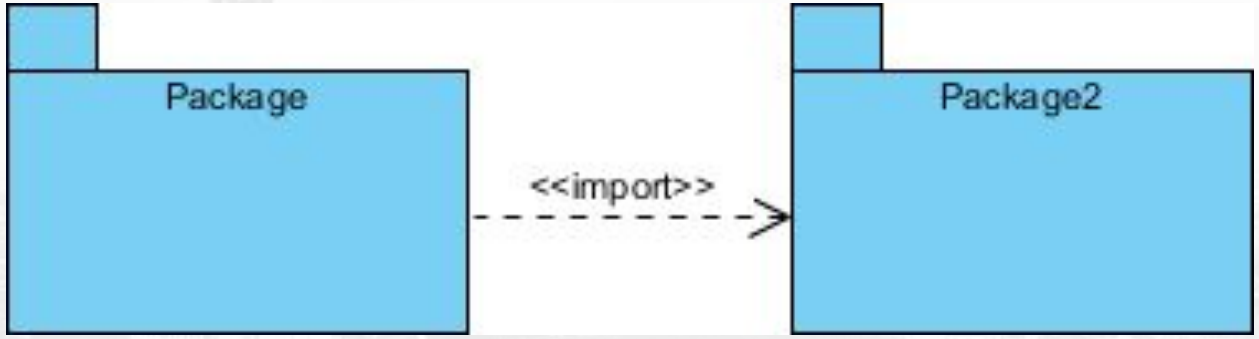


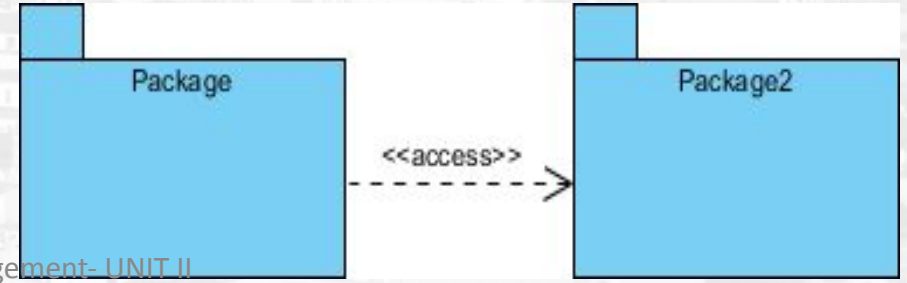
Package Diagram

- It shows the decomposition of the model itself into organization units & their dependencies

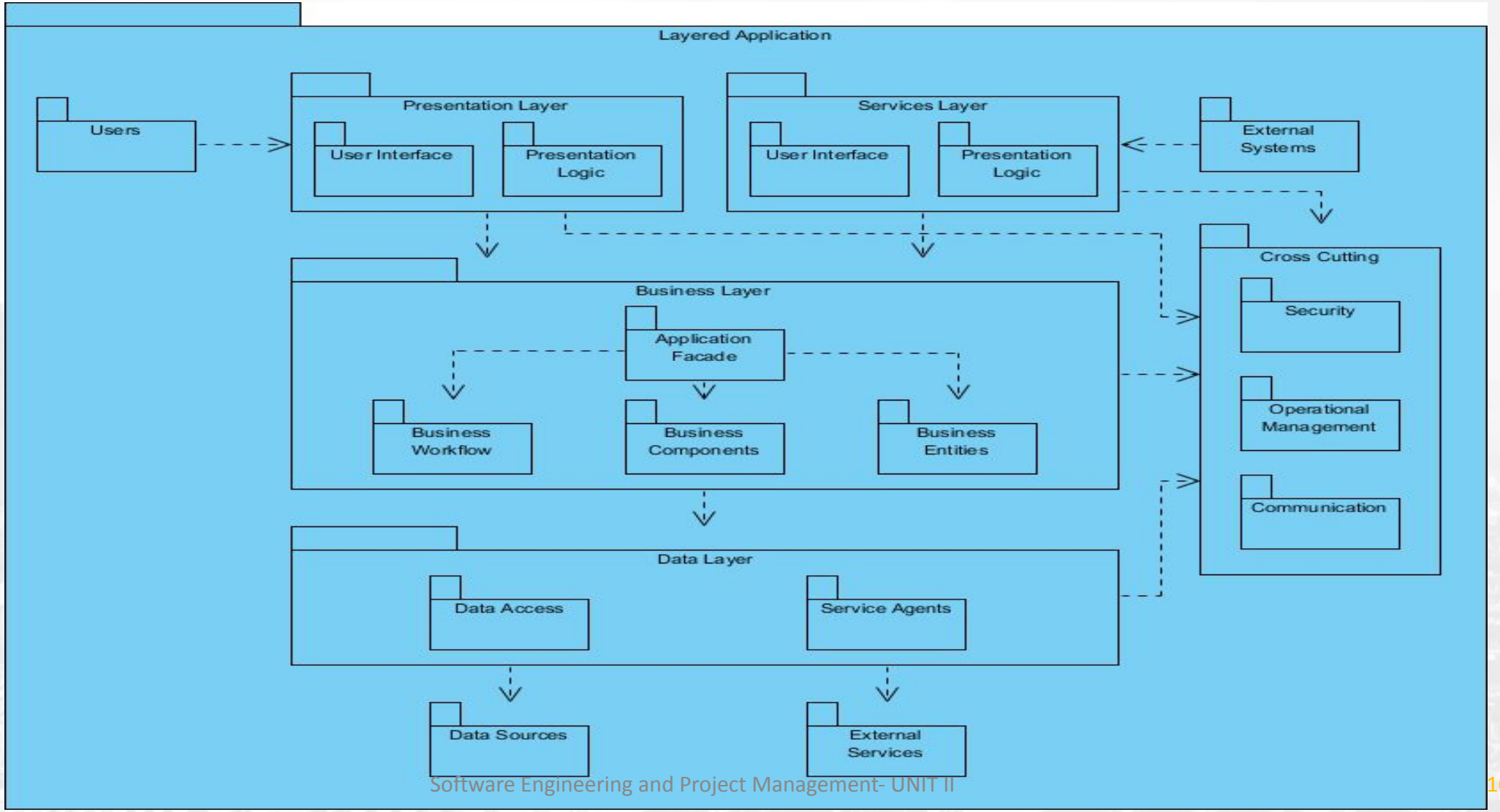


- Two stereotypes are used in dependency: <<import>> & <<access>>.
- **<<import>>** - one package imports the functionality of other package
- **<<access>>** - one package requires help from functions of other package.

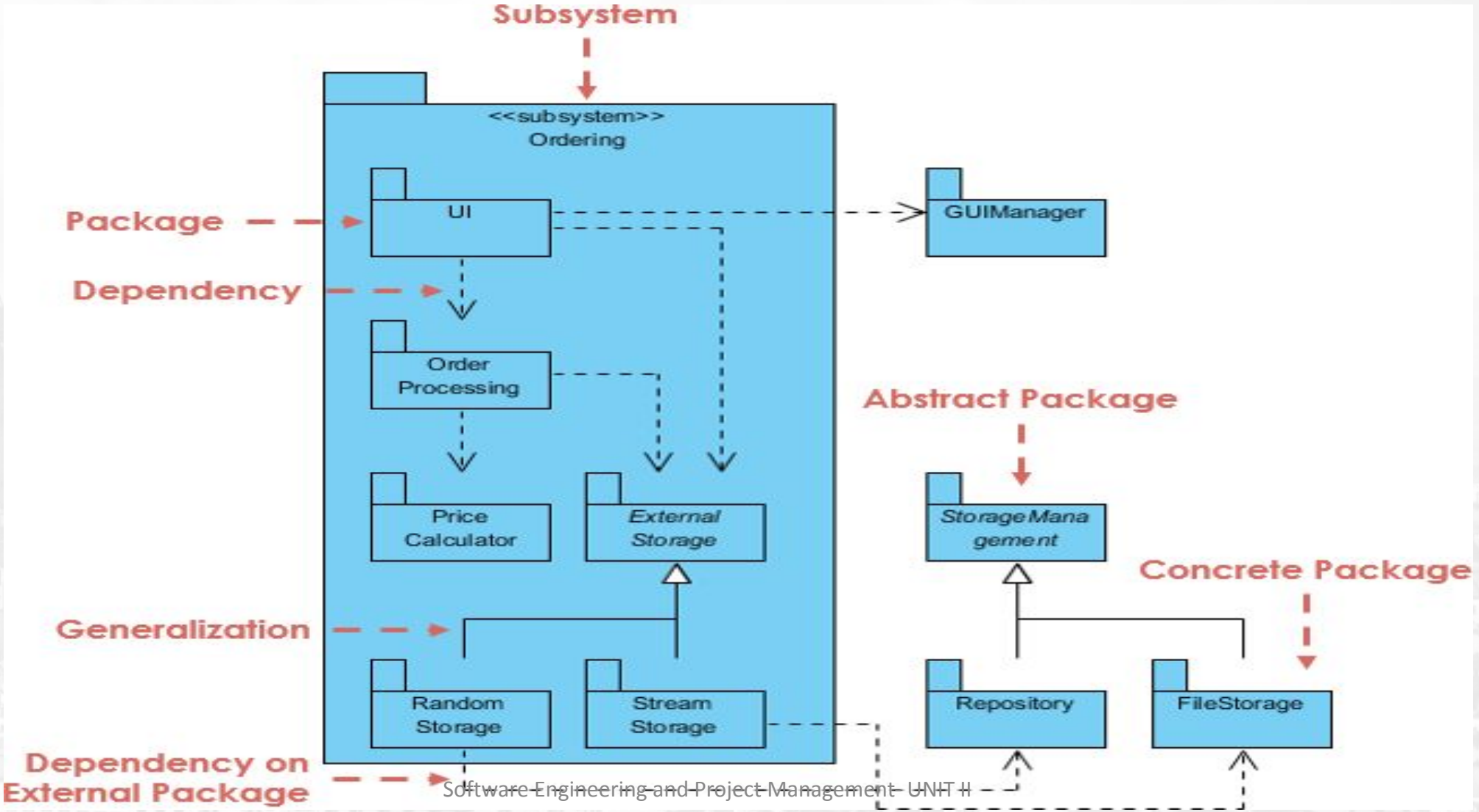




Package Diagram Example - Layered Application



Package Diagram Example - Order Subsystem



Package Diagram Example - Order Subsystem

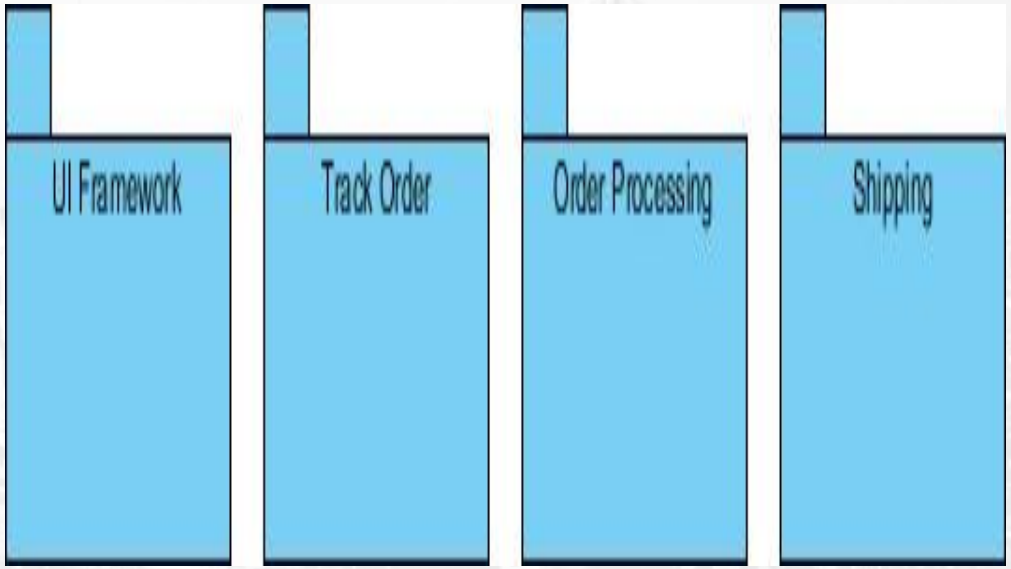
- **Order Processing System - The Problem Description**

We are going to design package diagram for "Track Order" scenario for an online shopping store. Track Order module is responsible for providing tracking information for the products ordered by customers. Customer types in the tracking serial number, Track Order modules refers the system and updates the current shipping status to the customer.

- Based on the project Description we should first identify the packages in the system and then related them together according to the relationship:

Identify the packages of the system

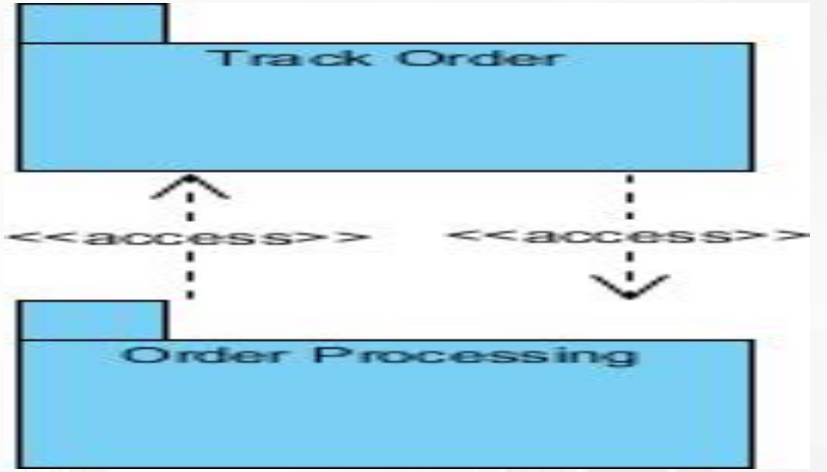
- There is a track order module, it has to talk with other module to know about the order details, let us call it "Order Details".
- Next after fetching Order Details it has to know about shipping details, let us call that as "Shipping".



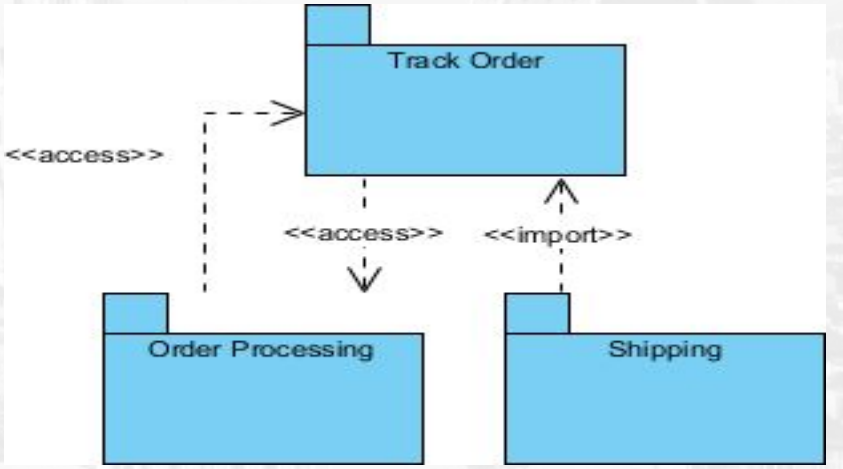
Package Diagram Example - Order Subsystem

Identify the dependencies in the System

- Track order should get order details from "Order Details" and "Order Details" has to know the tracking info given by the customer. Two modules are accessing each other which suffices <<access>> dual dependency

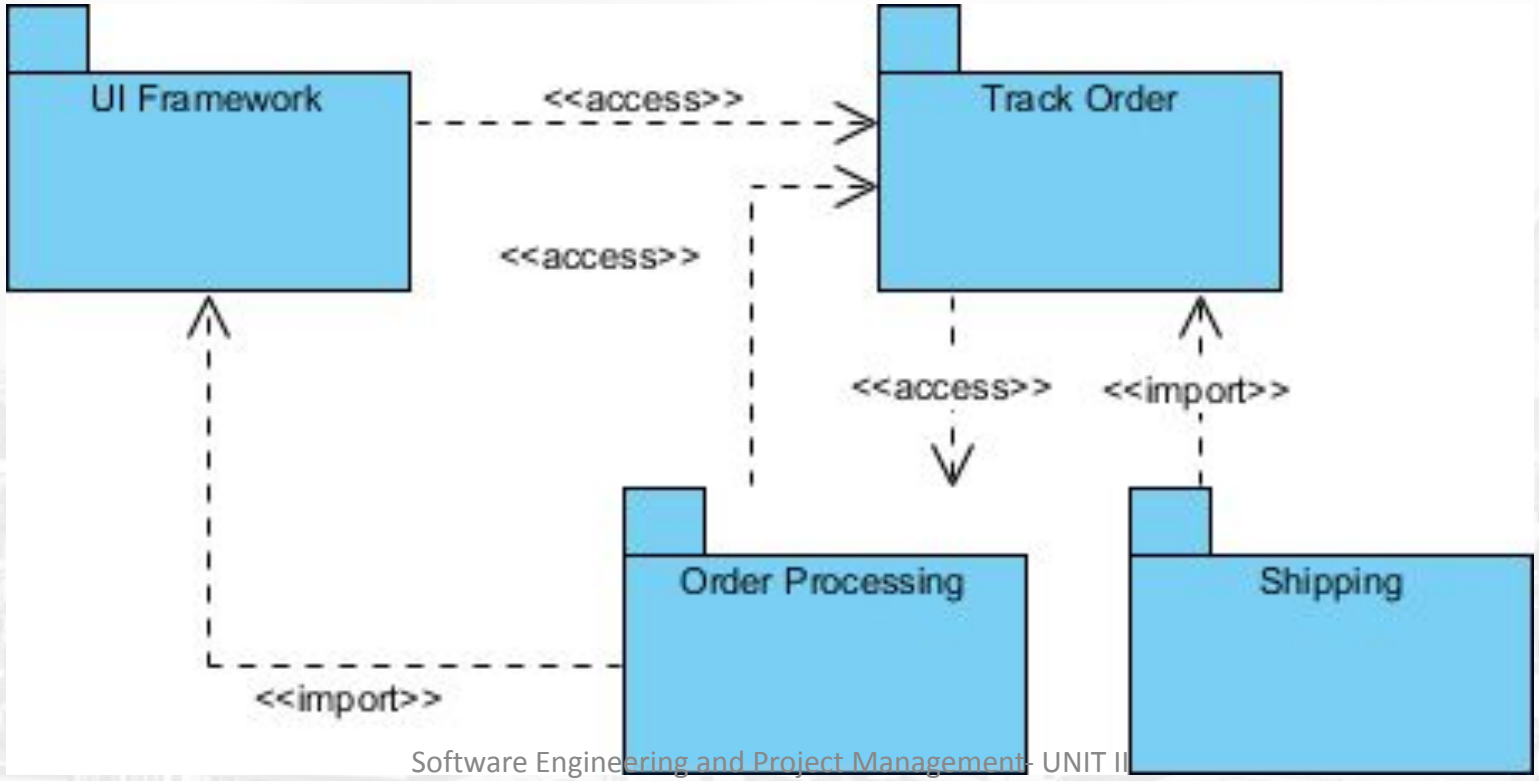


- To know shipping information, "Shipping" can import "Track Order" to make the navigation easier.



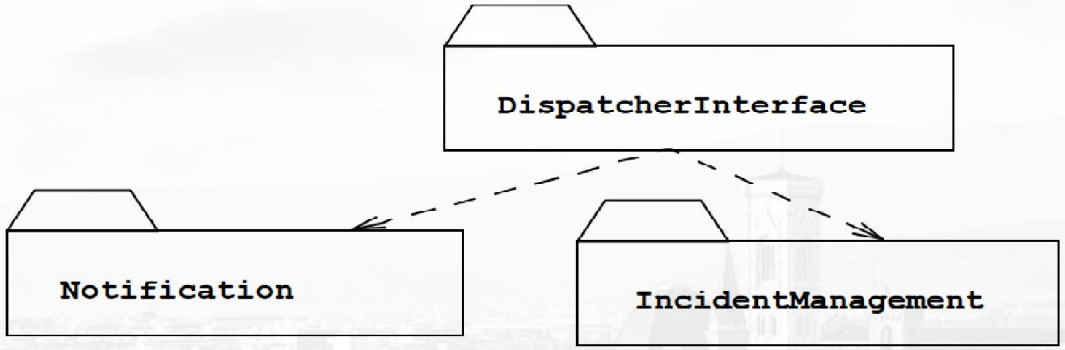
Package Diagram Example - Order Subsystem

- Finally, Track Order dependency to UI Framework is also mapped which completes our Package Diagram for Order Processing subsystem.

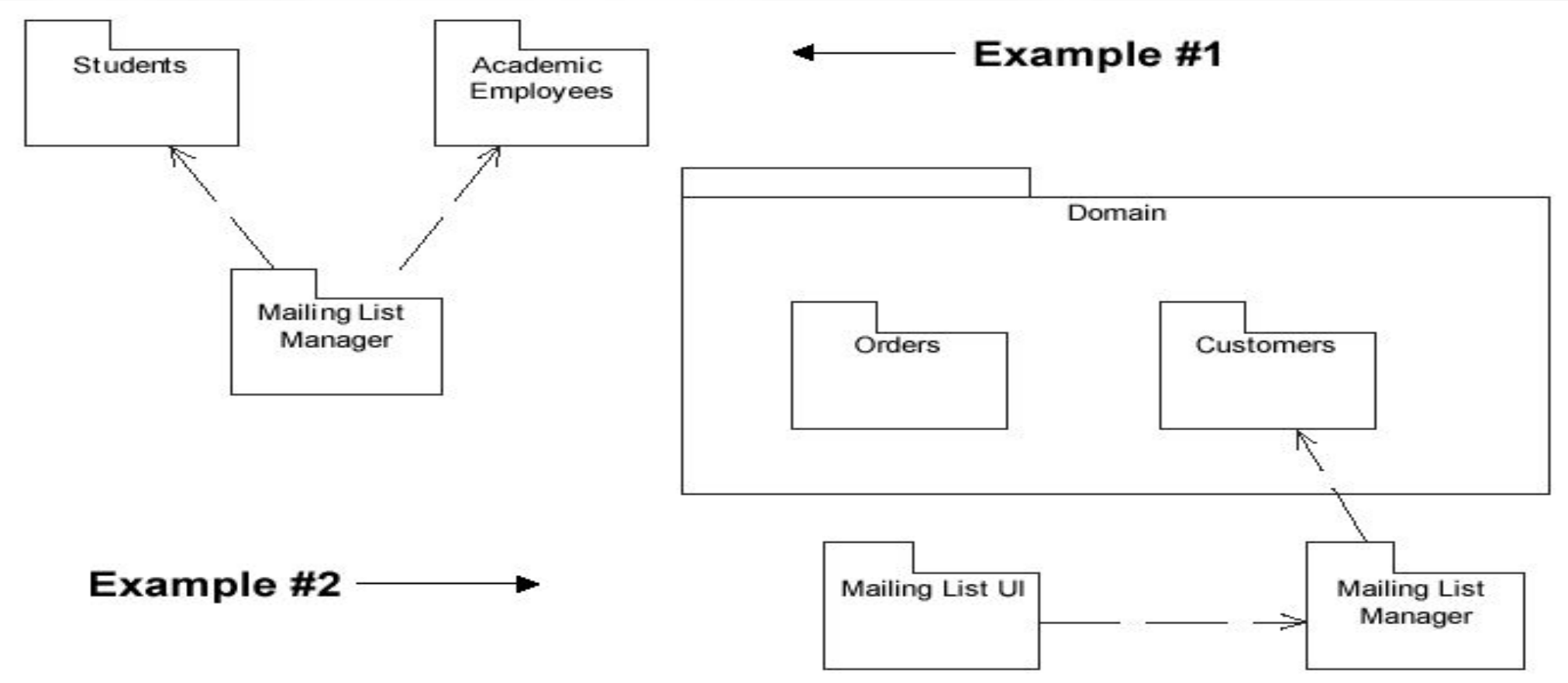


Package Example

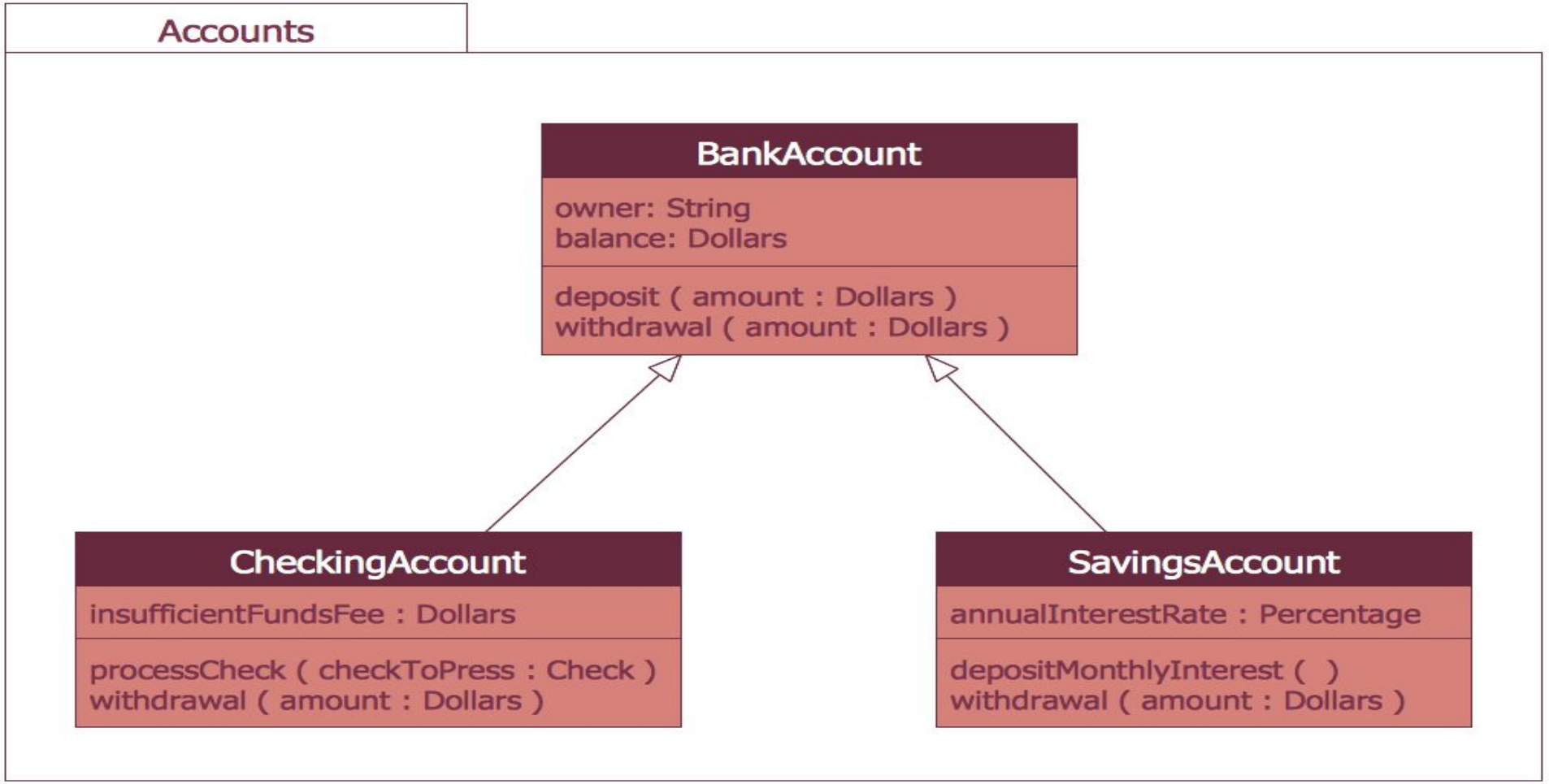
They provide a clear view of the hierarchical structure of the various UML elements within a given system.
These diagrams can simplify complex class diagrams into well-ordered visuals.
They offer valuable high-level visibility into large-scale projects and systems.
Package diagrams can be used to visually clarify a wide variety of projects and systems.
These visuals can be easily updated assystems and projects evolve.
Basic com



More Package Examples



Package Diagram Example - Accounts



Component Diagram

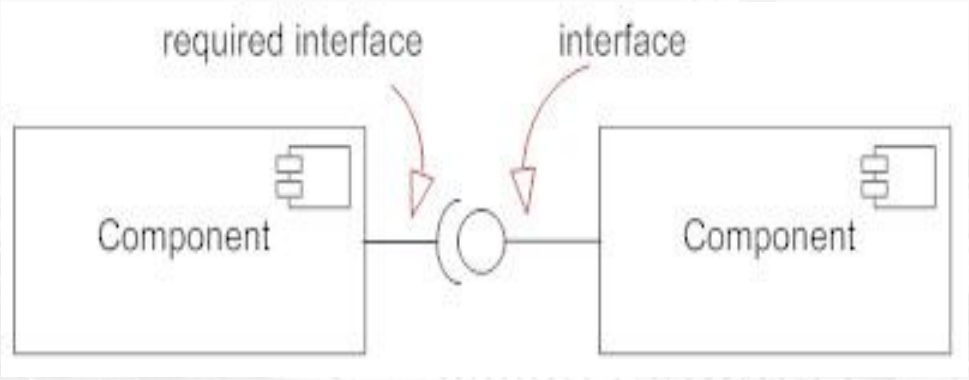
- Component diagrams are used to model the physical aspects of a system.
- Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.
- Component diagrams are used to visualize the organization and relationships among components in a system.
- Component diagrams can also be described as a static implementation view of a system.
- Static implementation represents the organization of the components at a particular moment.
- They show the organization and dependencies between a set of components.
- It represents the software layout of the system

Basic Component Diagram Symbols and Notations

- Component
- A component is a logical unit block of the system, a slightly higher abstraction than classes.

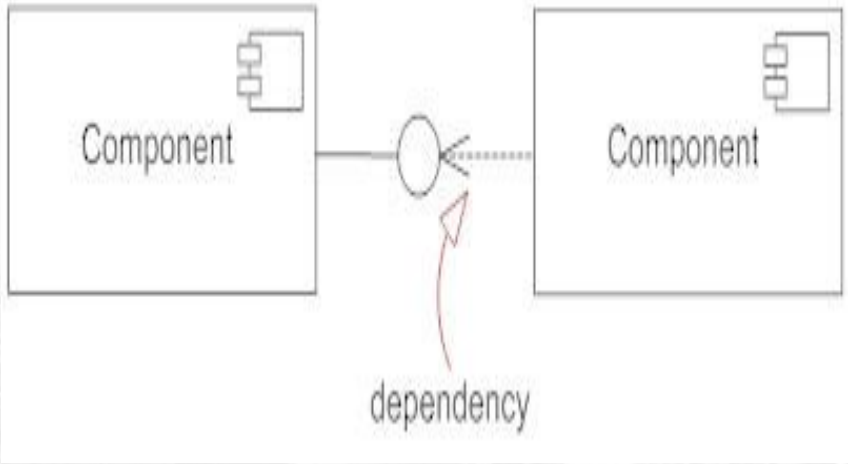
Basic Component Diagram Symbols and Notations

- Interface
- An interface (small circle or semi-circle on a stick) describes a group of operations used (required) or created (provided) by components. A full circle represents an interface created or provided by the component. A semi-circle represents a required interface, like a person's input.



Component diagram notation

- Dependencies



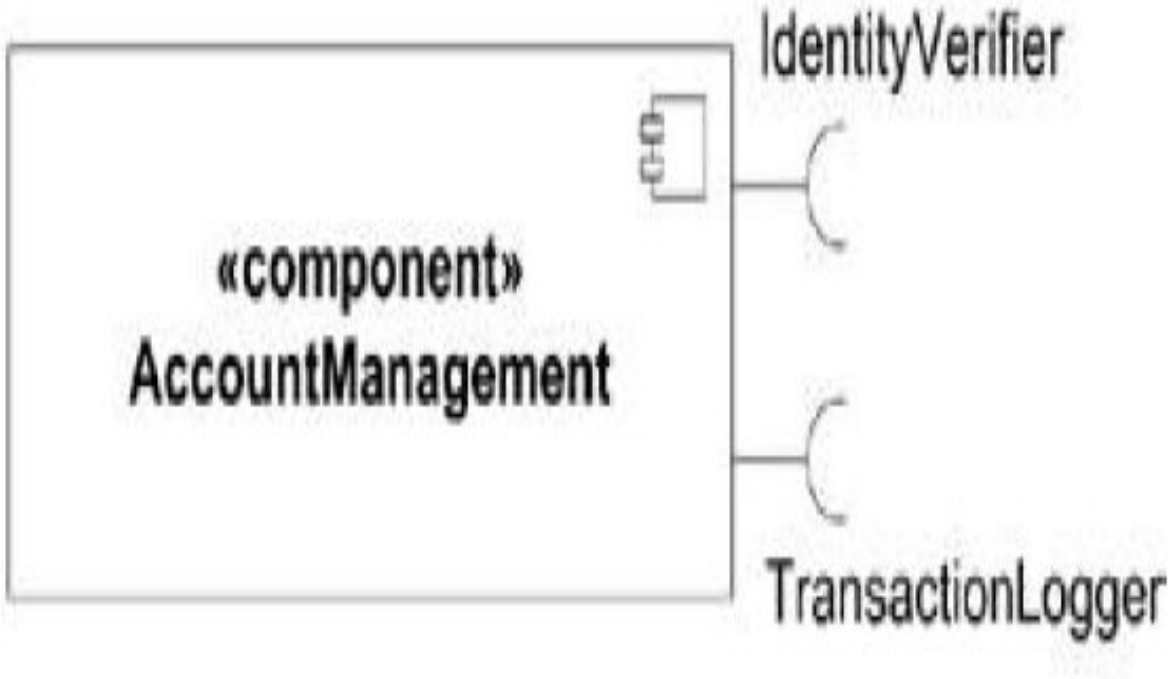
Component Views

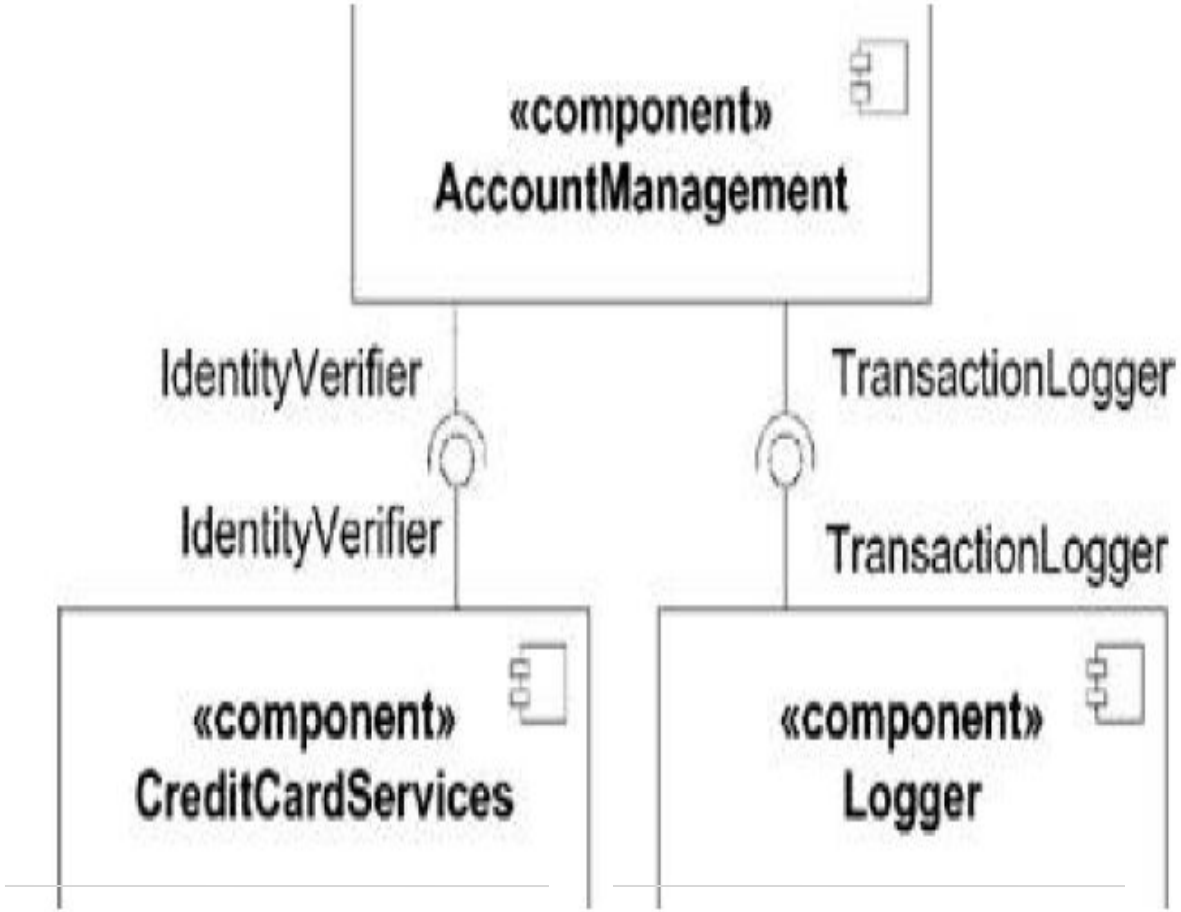
- UML uses two views of components, a **black-box view** and a **white-box view**.
- The black-box view shows a component from an **outside perspective**;
- The white-box view shows how a component **realizes the functionality** specified by its provided interfaces.

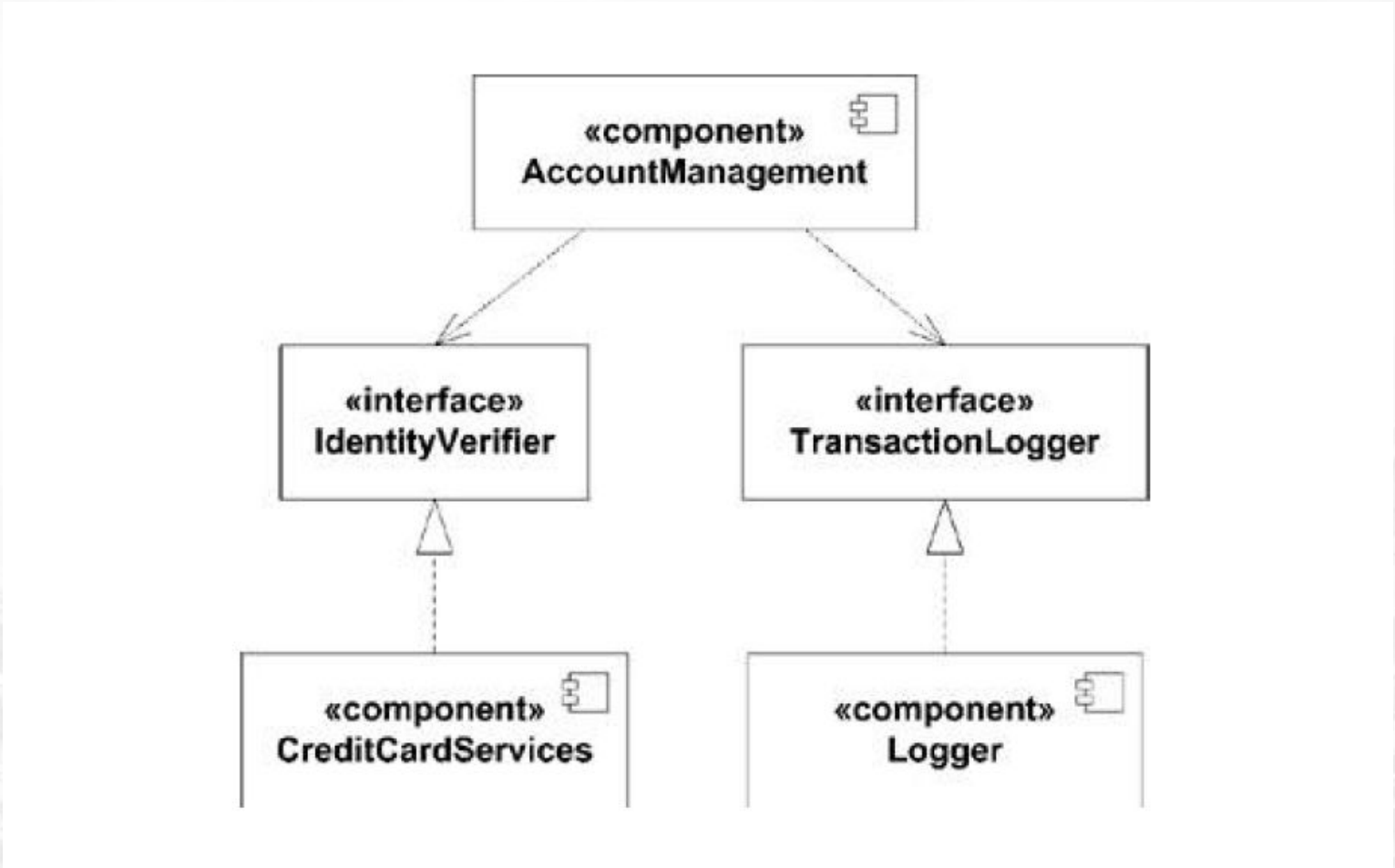
Black-Box View

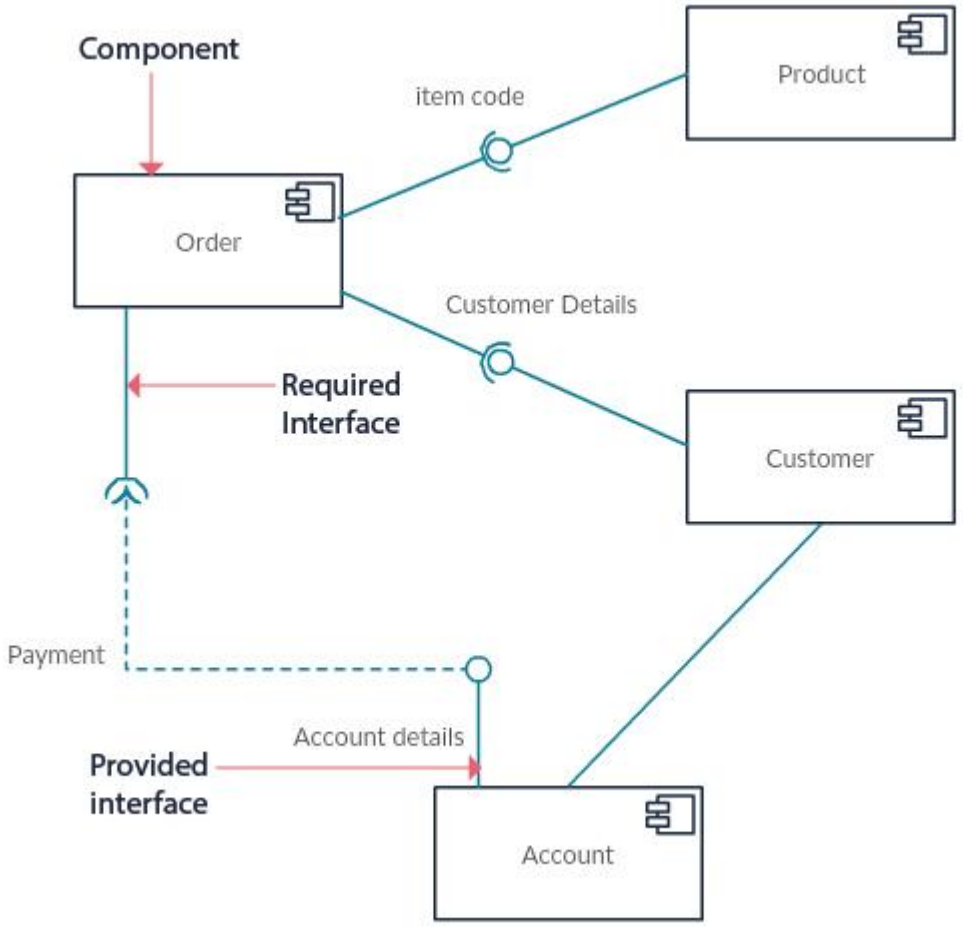
- The black-box view of a component shows the interfaces the component provides the interfaces it requires

- It does not specify anything about the internal implementation of the component.


IdentityVerifier
«component»
AccountManagement
TransactionLogger

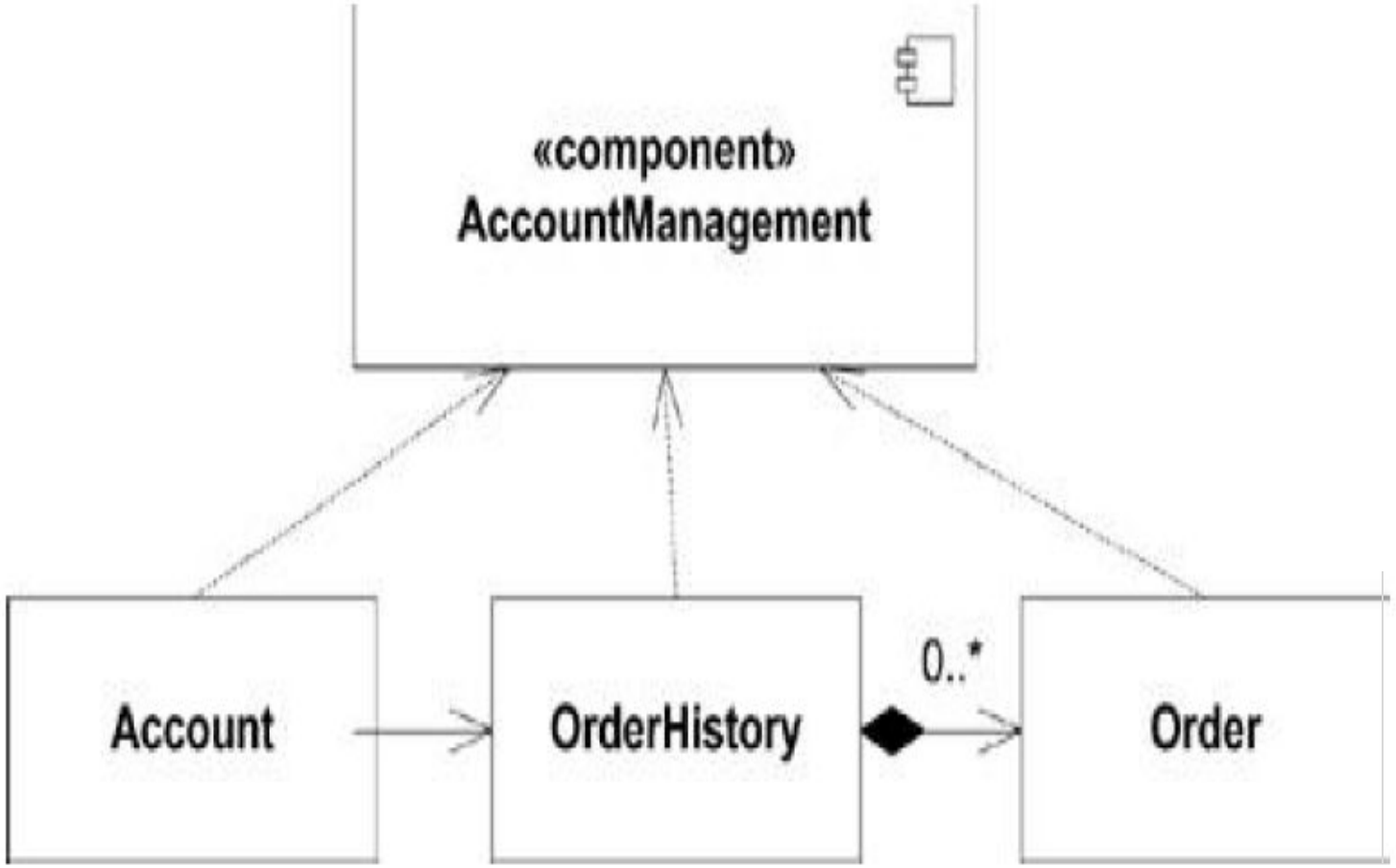

«component»
AccountManagement
IdentityVerifier
IdentityVerifier
TransactionLogger
TransactionLogger
«component»
CreditCardServices
«component»
Logger


«component»
AccountManagement
«interface»
IdentityVerifier
«interface»
TransactionLogger
«component»
CreditCardServices
«component»
Logger


Component
item code
Product
Order
Customer Details
Required
Interface
Customer
Payment
Account details
Provided
interface
Account

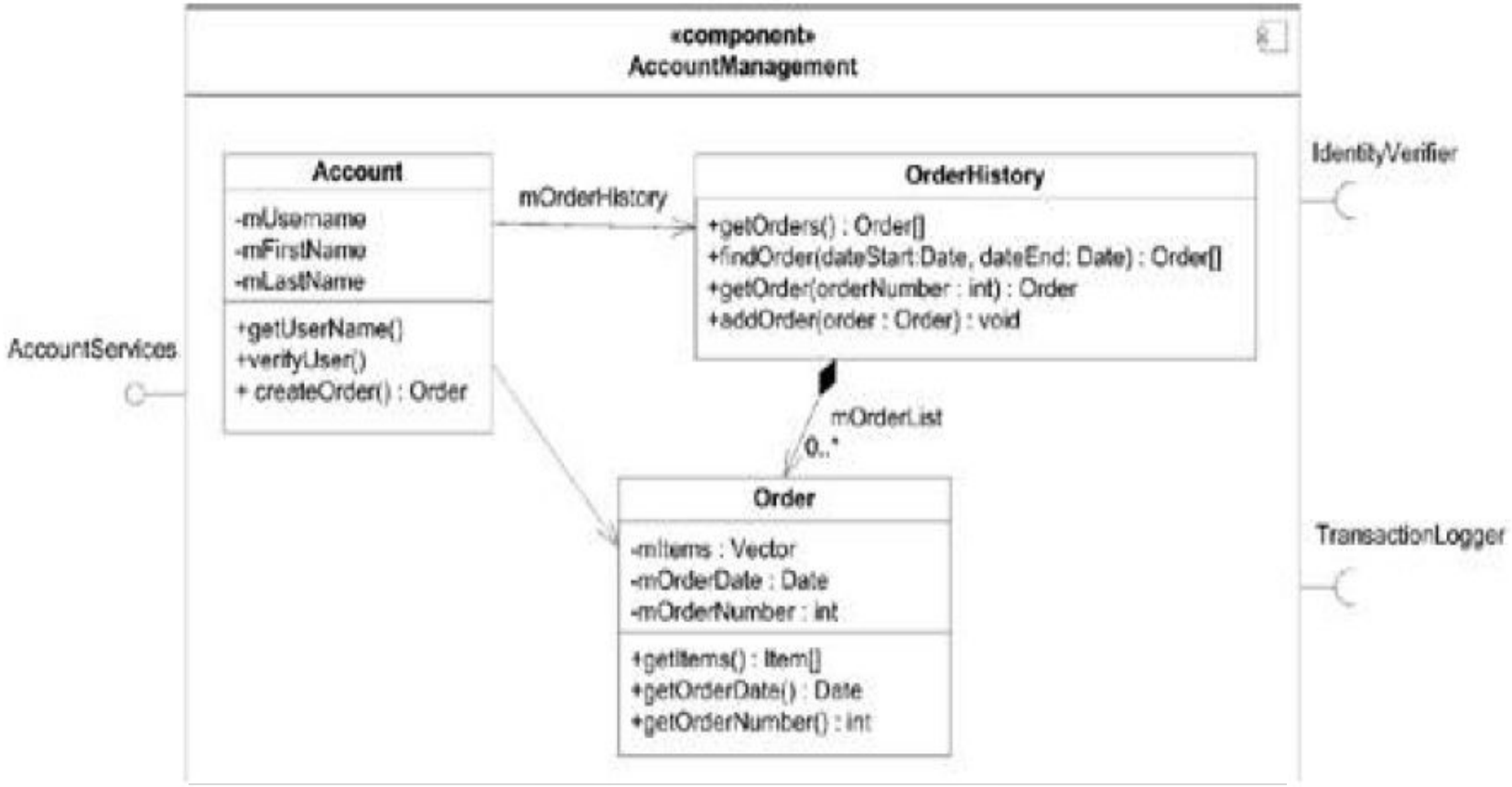
White Box view

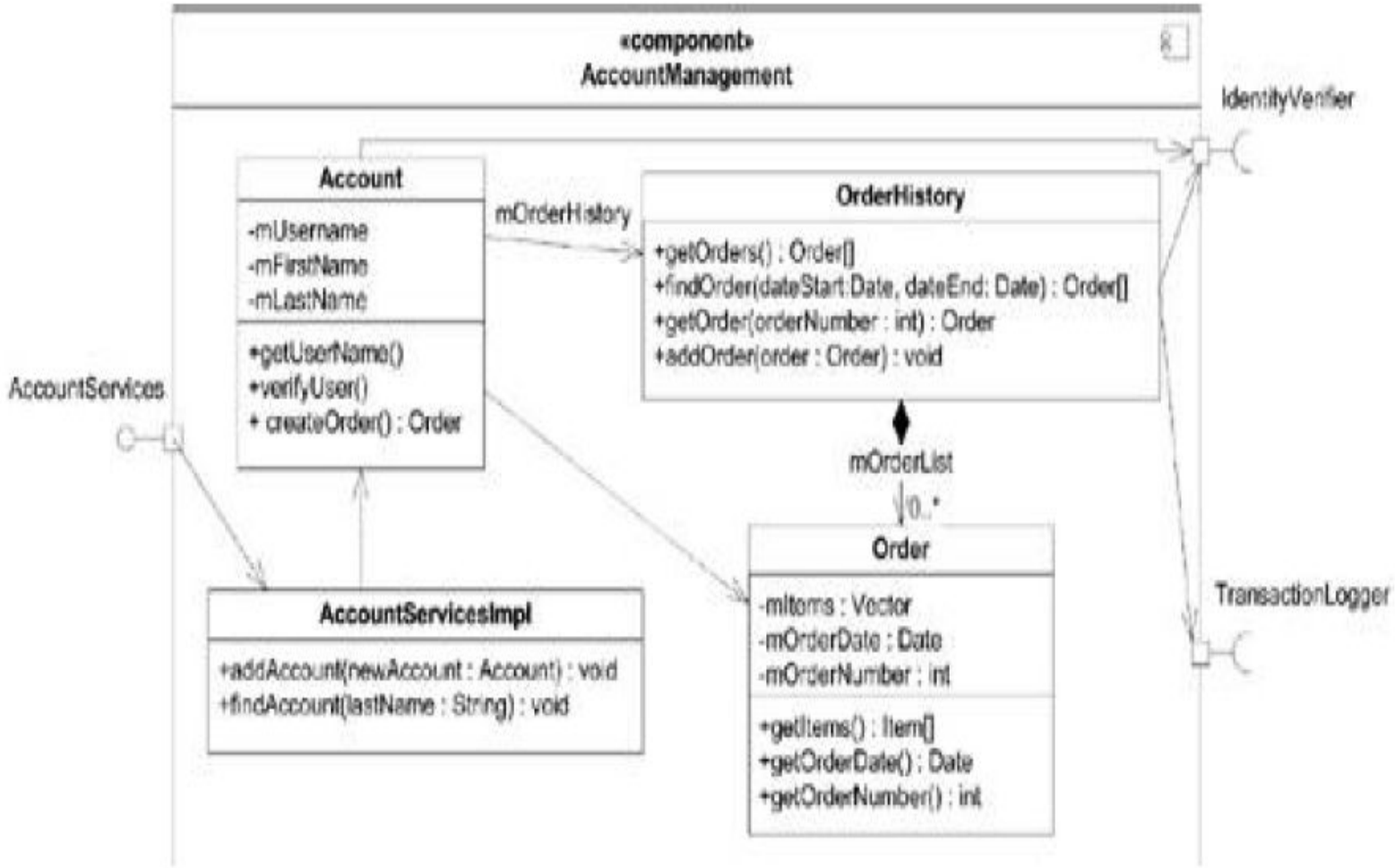
- In order to **provide details about the implementation** of a component, UML defines a white box view.
- The white-box view shows exactly **how a component realizes the interfaces** it provides.
- This is typically done using classes and is illustrated with a class diagram


«component»
AccountManagement
Account
OrderHistory
0..*
Order

Detailed Realization

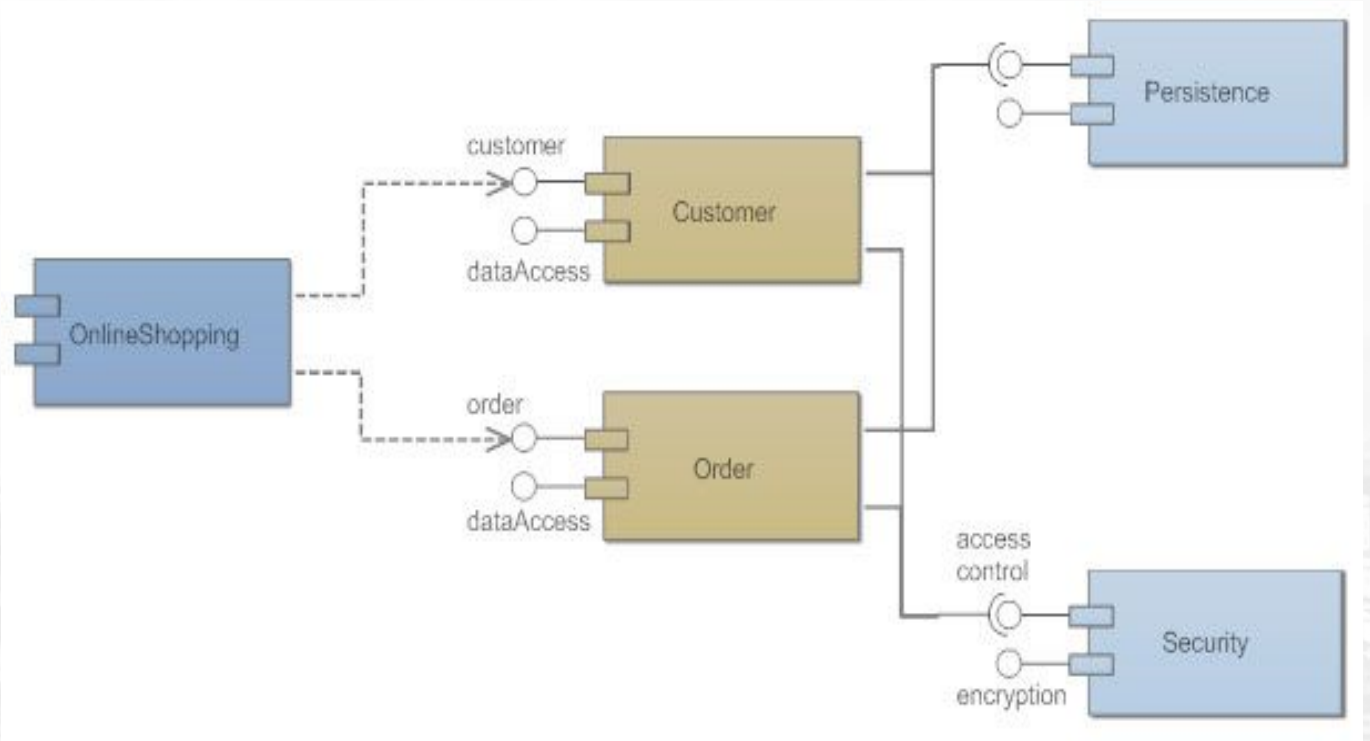
Level 2



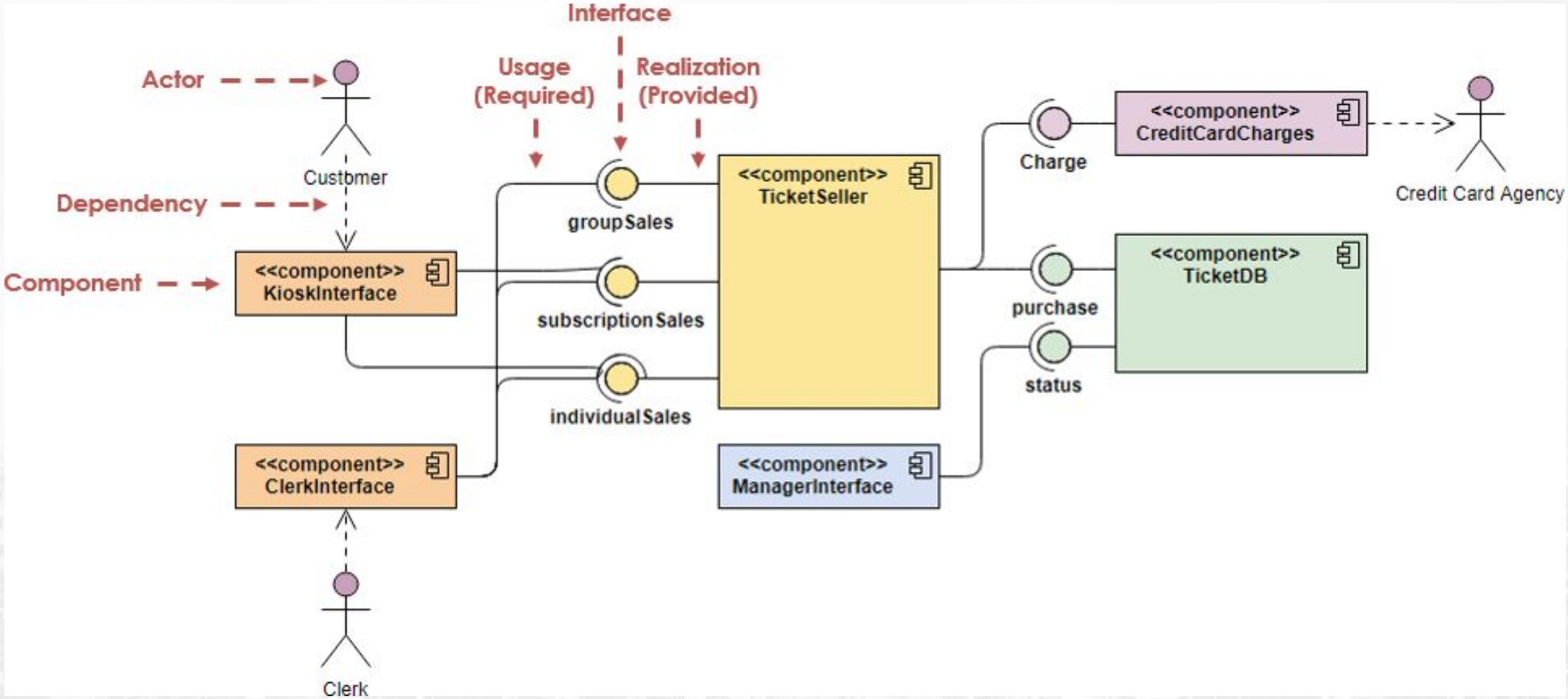

«component»
AccountManagement
IdentityVerifier
Account
-mUsername
-mFirstName
-mLastName
+getUserName()
+verifyUser()
+ createOrder() : Order
mOrderHistory
OrderHistory
+getOrders() : Order[]
+findOrder(dateStart:Date, dateEnd: Date) : Order[]
+getOrder(orderNumber : int) : Order
+addOrder(order : Order) : void
AccountServices
mOrderList
0..*
Order
-mItems : Vector
-mOrderDate : Date
-mOrderNumber : int
+getItems() : Item[]
+getOrderDate() : Date
+getOrderNumber() : int
TransactionLogger
AccountServicesImpl
+addAccount(newAccount : Account) : void
+findAccount(lastName : String) : void

Component diagram example

- Online shopping system – component diagram



Component diagram – Ticket selling system



Deployment Diagram

- Deployment diagrams are used for modeling configuration of ***run-time processing*** nodes and the components that live on them.
- Deployment diagrams are used to model the ***static deployment view*** of a system.
- This involves modeling the topology of the hardware on which the system executes.

Deployment Diagram

- Deployment diagrams Show the physical relationship between hardware and software in a system
- Hardware elements
- Computers (clients, servers)
- Embedded processors
- Devices (sensors, peripherals)
- Are used to show the nodes where software components reside in the run-time system

Deployment diagram

- **Purpose of Deployment Diagrams**

✔ They show the structure of the run-time system

✔ They capture the hardware that will be used to implement the system and the links between different items of hardware.

✔ They model physical hardware elements and the communication paths between them

✔ They can be used to plan the architecture of a system.

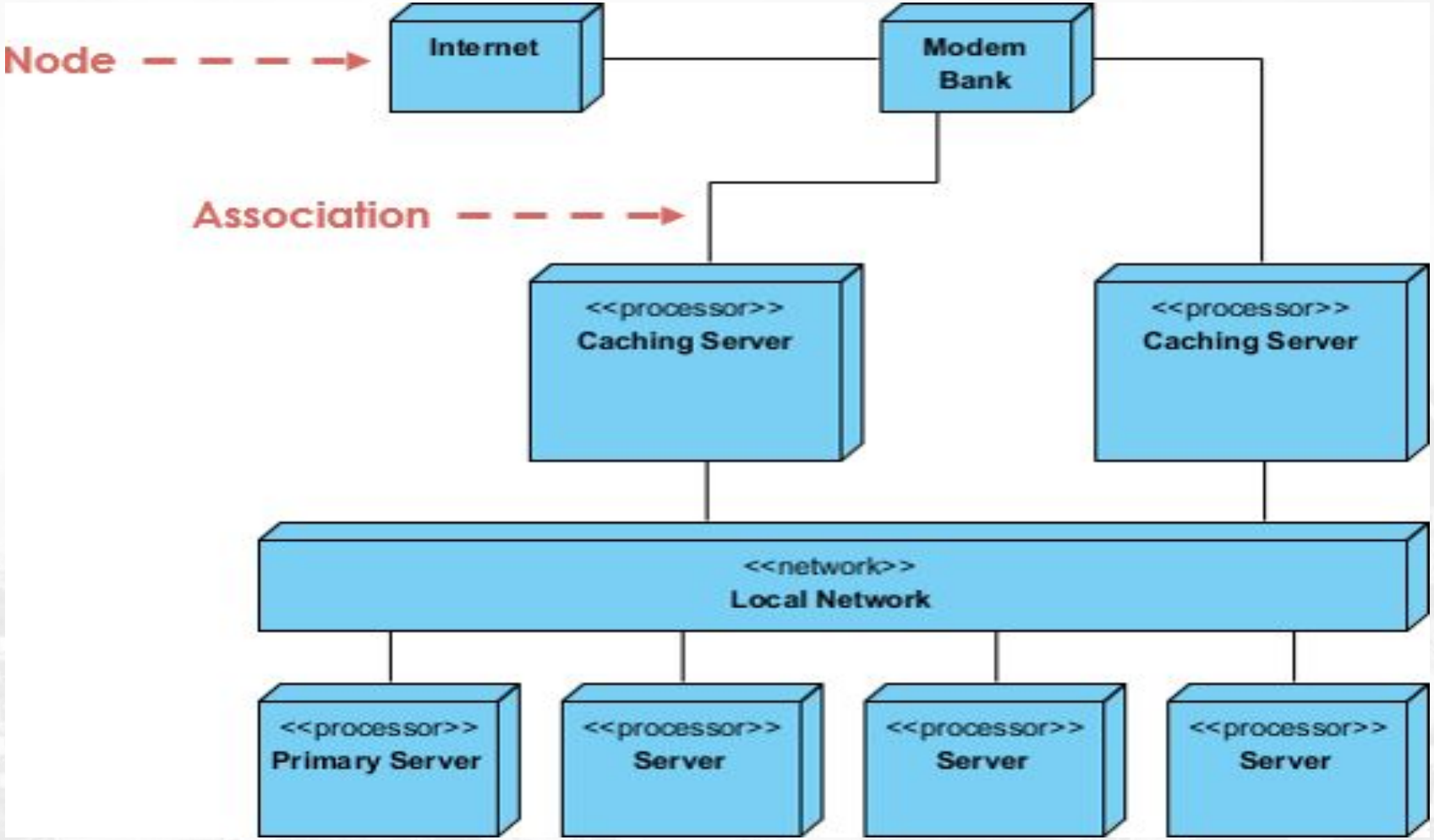
✔ They are also useful for Document the deployment of software components or nodes

Software Engineering and Project Management- UNIT II

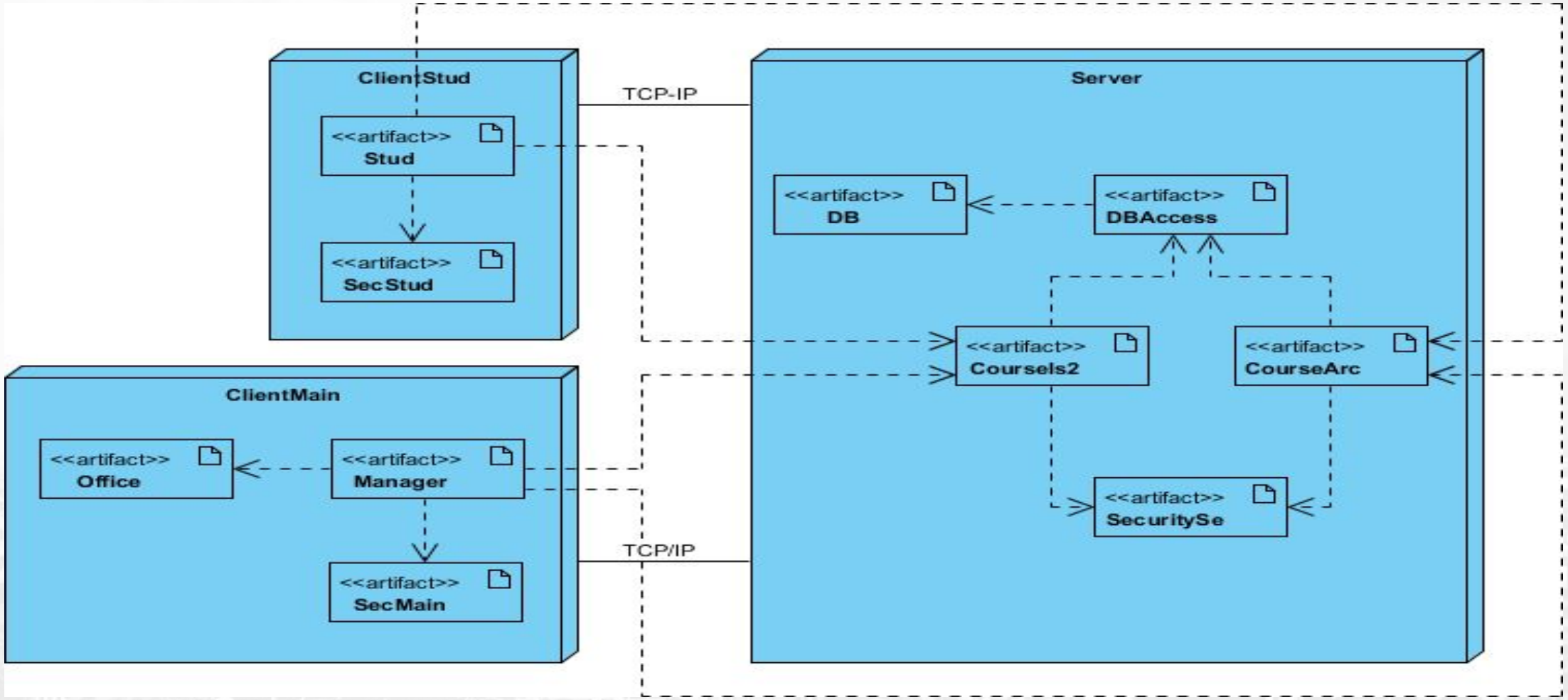
Deployment diagram

- Deployment diagram contains nodes and connections
- A node usually represent a piece of hardware in the system
- A connection depicts the communication path used by the hardware to communicate
- Usually indicates the method such as TCP/IP

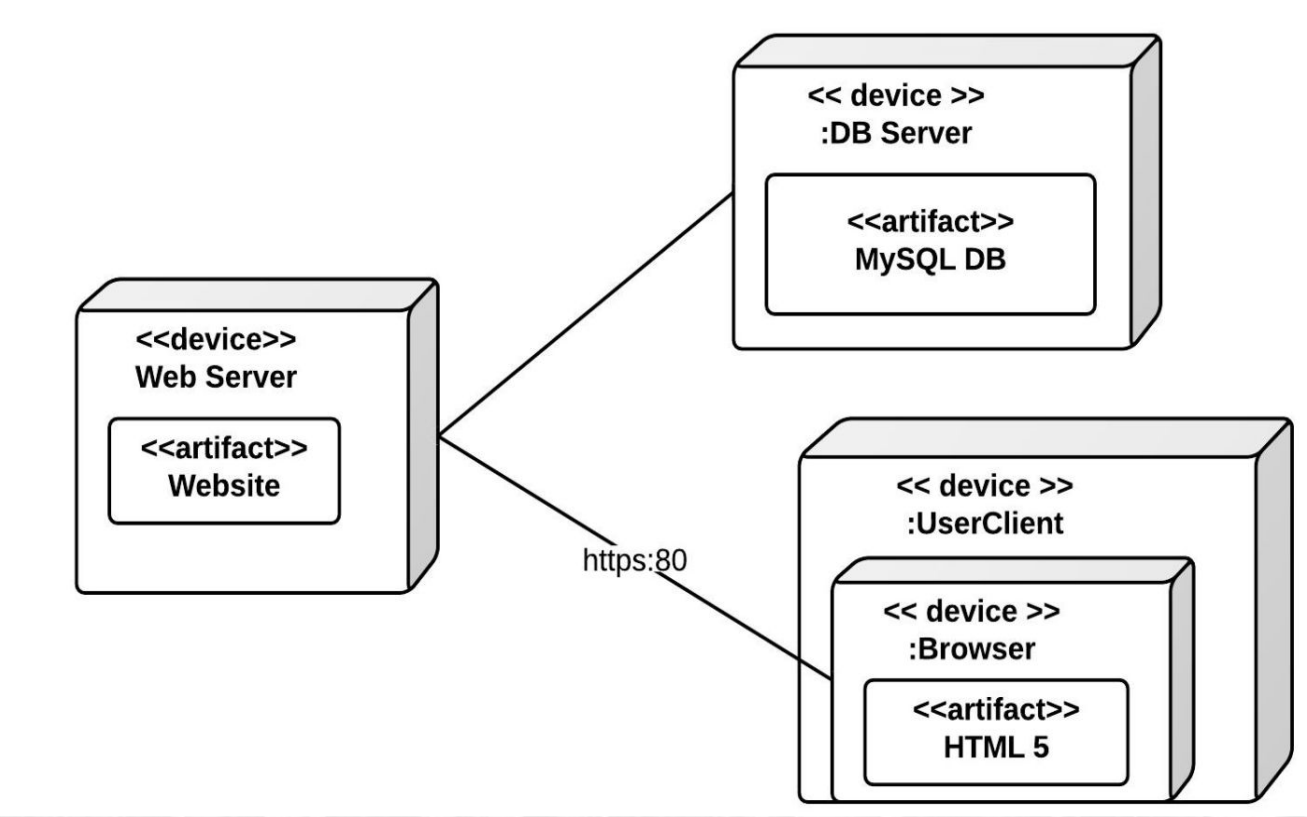
Deployment diagram of embeded system



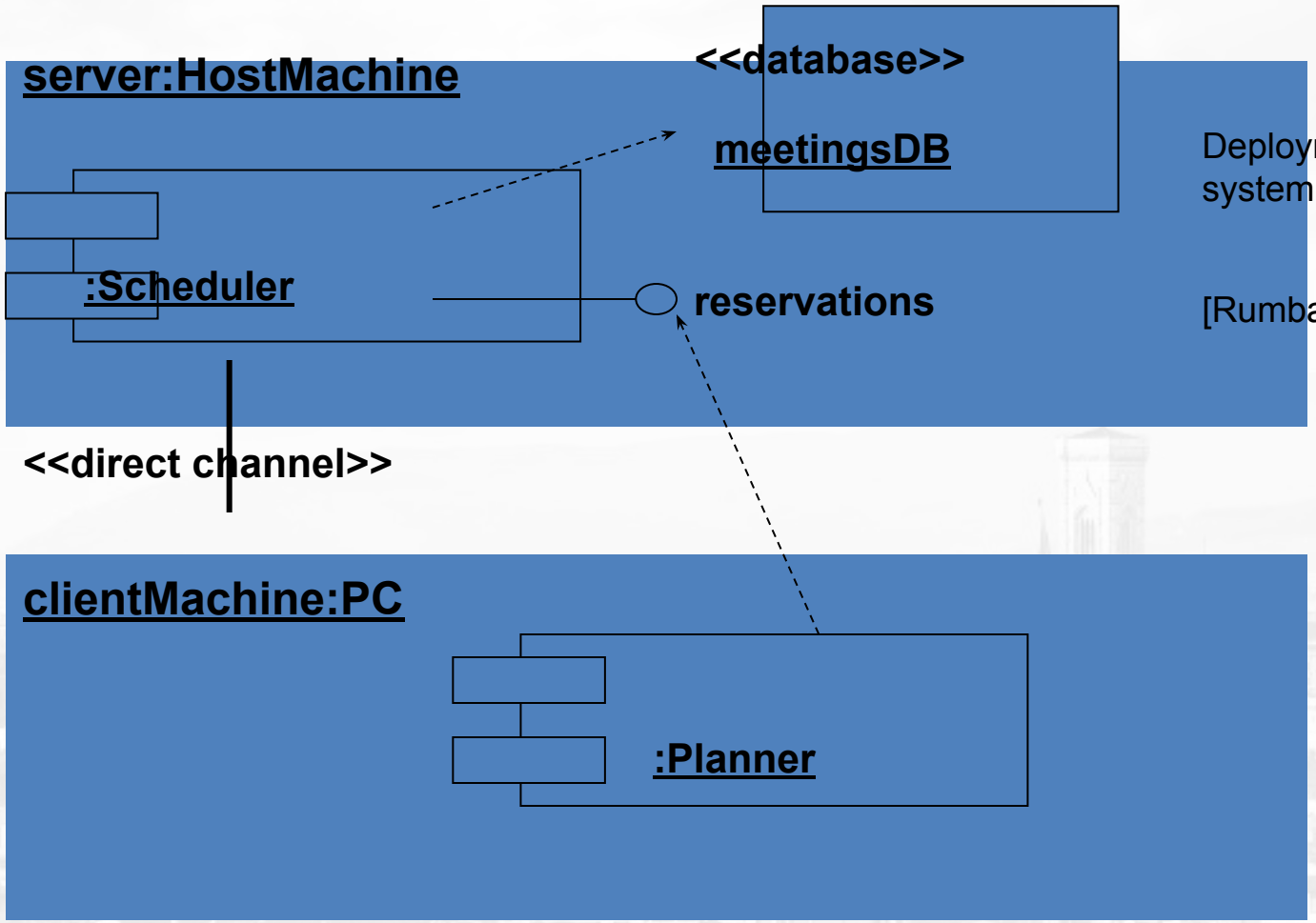
Deployment diagram of TCP/IP



Example:


<< device >>
:DB Server
<<artifact>>
MySQL DB
<<device>>
Web Server
<<artifact>>
Website
https:80
<< device >>
:UserClient
<< device >>
:Browser
<<artifact>>
HTML 5

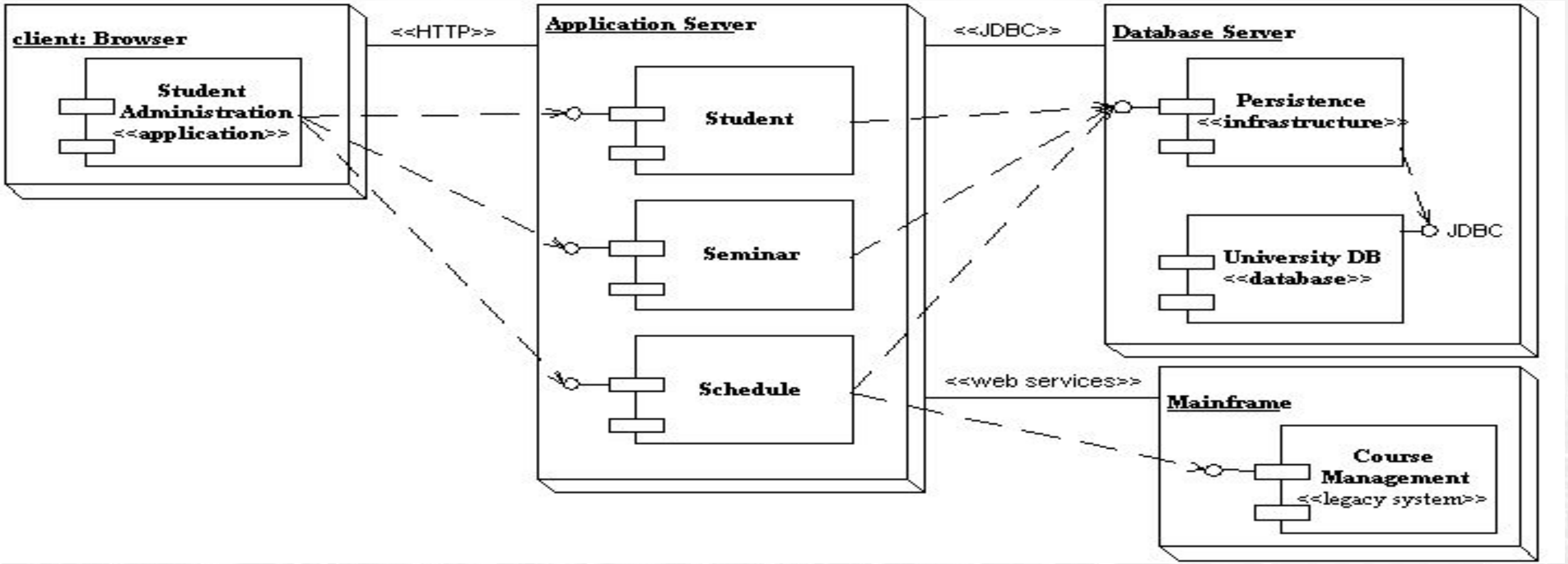
Deployment Diagram



Deployment diagram of a client-server system.

[Rumbaugh,99]

Sample Communication Links



References

- UML 2.0 in nutshell, by Dan Pilone & Neil Pitman,O'Reilly .
- Roger S Pressman, Software Engineering: A Practitioner's Approach, Mcgraw-Hill, ISBN: 0073375977, Seventh Edition, 2014
- Grady Booch, James Rambaugh, Ivar Jacobson, —The unified modeling language user guide, Pearson Education, Second edition, 2008, ISBN 0-321-24562-8

Disclaimer:

a. Information included in this slides came from multiple sources. We have tried our best to cite the sources. Please refer to the References to learn about the sources, when applicable.

b. The slides should be used only for academic purposes (e.g., in teaching a class), and should not be used for commercial purposes.