SOFTWARE ENGINEERING

SOFTWARE ARCHITECTURE

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

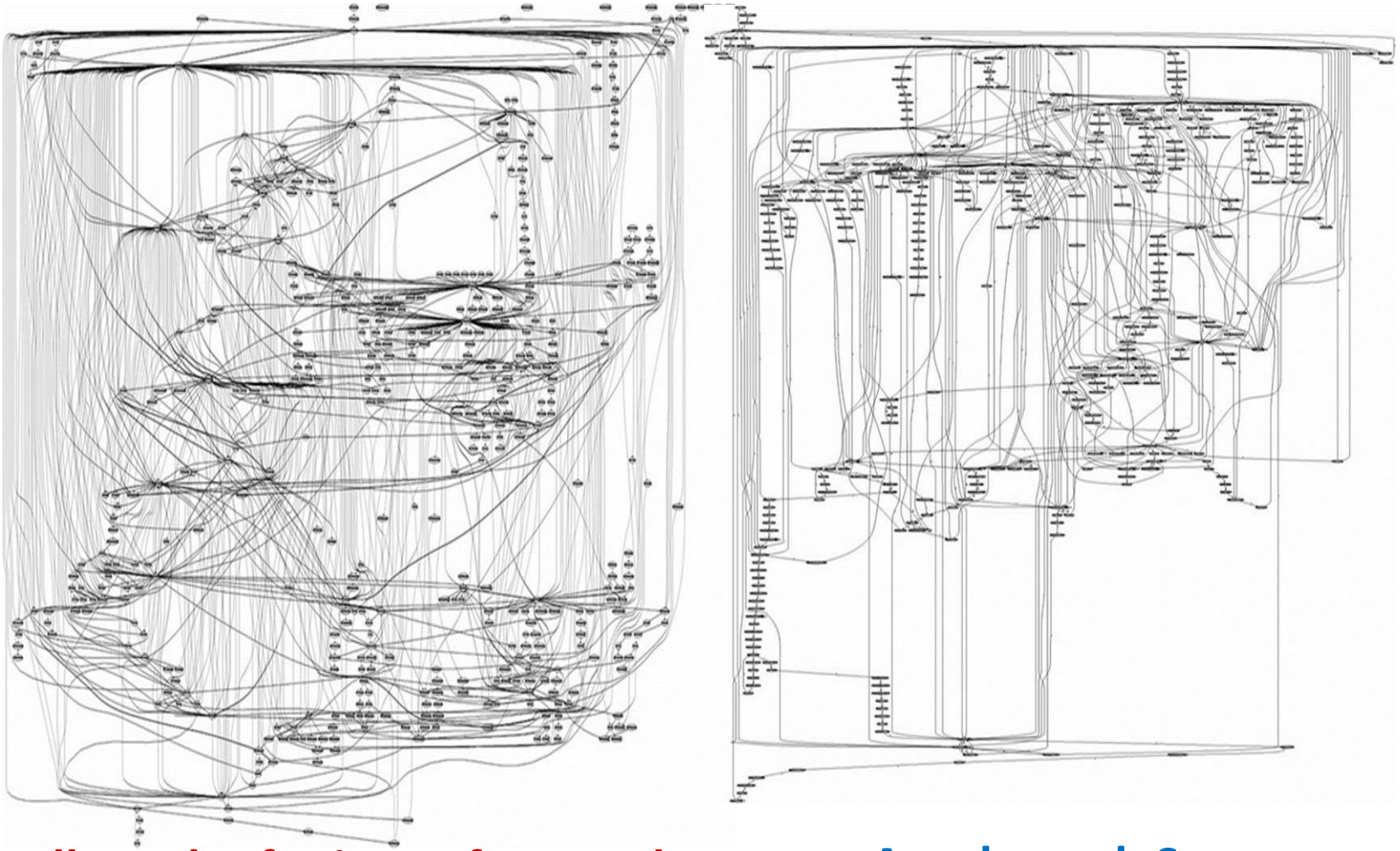
SOFTWARE ARCHITECTURE

Introduction to Software Architecture & Design

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Introduction to Software Architecture

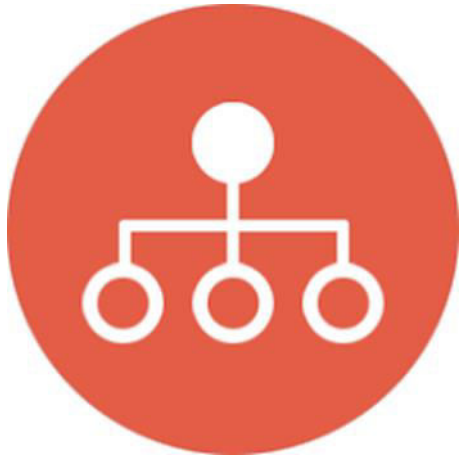
Guess what is this **.. and this**



Call graph of Microsoft IIS Webserver **Apache web Server**

SOFTWARE ARCHITECTURE

What is software architecture



- Software Architecture is the top level decomposition into major components together with a characterization of how these components interact
- Software architecture of a computing system can also be looked at as a big picture depiction of the system
  - Used for communication among stakeholders
  - Blueprint for the system and project under development
  - Negotiations and balancing of functional and quality goals happen during architecture

SOFTWARE ARCHITECTURE

Why is architecture important

- Architecture manifests the earliest set of design decisions
  - Constraints on implementation
  - Dictates organizational structure
  - Inhibits or enable quality attributes and supports WBS
- It's a vehicle for stakeholder communication
- Supports reuse at architectural system level like a Product lines or at component level where Systems can be built using large, externally developed elements
- Changes to Architecture is Expensive in the later phases of SDLC

Characteristics of software architecture

1. Addresses variety of stakeholder perspectives
2. Realizes all of the Use cases and scenarios envisaged for the problem
3. Supports separation of concerns
   - Stakeholder concerns using separate architectural views
4. Quality driven
   - Closely related to quality attributes like availability, security ..
5. Recurring styles
   - Recurring solutions like architectural styles, patterns
6. Conceptual integrity
   - It should represent an overall vision of what the system should do separated from its implementation

Factors that influence the software architecture

1. Functionality
2. Data Profile
3. Audience
4. Usage characteristics
5. Business Priority
6. Regulatory and Legal obligations
7. Architectural standards
8. Dependencies and Integration
9. Cost Constraints
10. Initial State
11. Architect and the staff background and the skill level
12. Technical and organizational environment
13. Technology constraints
14. Type of user experience planned

Introduction : Design Principles

Detailed design can involve

- Further Decomposition (post architecture) of the components being developed if necessary.
- Identification and description of the behavior of components/sub-systems
- Description of how the interfaces will actually be realized using the appropriate algorithms and data structures
- Description on how the system will facilitate interaction with the user through the user interface
- Use of appropriate structural and behavioral design patterns
- Having Maintenance and reuse as couple of its goals

SOFTWARE DESIGN

Techniques which enable design and Issues that need to be handled

Enabling techniques

1. Abstraction
   - Focus on essential properties
2. Modularity, coupling and cohesion
   - Modularity is the degree to which the larger module can be decomposed
   - Cohesion is the extern to which modules are dependent on each other. Focused responsibility **(Strong)**
   - Coupling indicates how strongly the modules are connected. **(Loose)**
3. Information hiding
   - Need to know
   - Encapsulation and Separation of interface and Implementation
4. Limiting complexity
   - Higher complexity worser design
   - Inter-modular or intra-modular
5. Hierarchical structure

Key Issues to be handled in design

1. Concurrency
2. Event Handling
3. Distribution of Components
4. Non-Functional Requirements
5. Error, Exception Handling, Fault-Tolerance
6. Interaction and Presentation
7. Data Persistence

CONTRASTING SOFTWARE ARCHITECTURE & DESIGN

Differences are in terms of

- When each of these happen in the lifecycle, areas of responsibility and the levels of decision making
- Architecture is the bigger picture in terms of frameworks, tools, languages, scope, goals and high level methodologies while design is the smaller picture of implementation methodology, with local constraints of how different parts of the system will look, design patterns, programming idioms, refactoring's & how code is organized.
- Architecture faces towards strategy, structure and purpose which is at higher level. Design is tactical and faces towards implementation and practice, more towards the concrete.

CONTRASTING SOFTWARE ARCHITECTURE & DESIGN

4. Software architecture is the structure (or structures) of the system, which comprise of software components, the externally visible properties of those components, and the relationships between them while

   Software design is problem-solving & planning for a software solution internal to the system

4. Architectural decisions are harder to change compared to design decisions which are simpler with lesser impact

5. Software architecture has more influence on the Non Functional Requirements of the system where the Design has more influence on the functional requirements of the system

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

SOFTWARE ENGINEERING

SOFTWARE ARCHITECTURE

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

SOFTWARE ARCHITECTURE

Easing in & Architectural Model

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Easing into Architecture

Consider a very commonly used technology for talking to a remote person.

- Plain old telephone

- Central office hardware (PSTN switch)
- Benefits
  - Works through power outage
  - Reliability

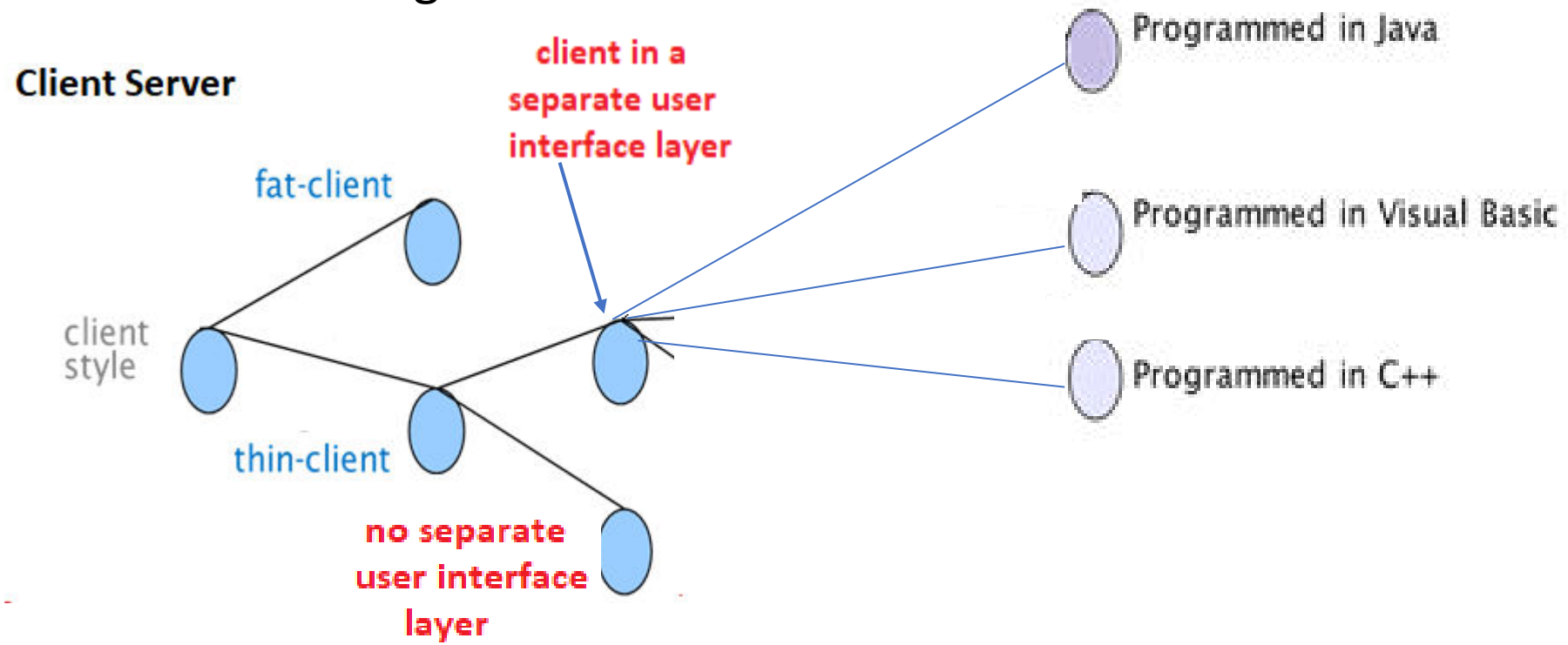
- Skype
- Peer to peer
- Benefits
  - Scales without changes
  - Features can be added easily

Two architectural choices

- Centralized versus peer to peer

Design issues, options and decisions

- A designer can face a series of design issues or problems
- He / She then makes a design decision to resolve each issue
  - This involves choosing the best option from among the alternatives. E.g.



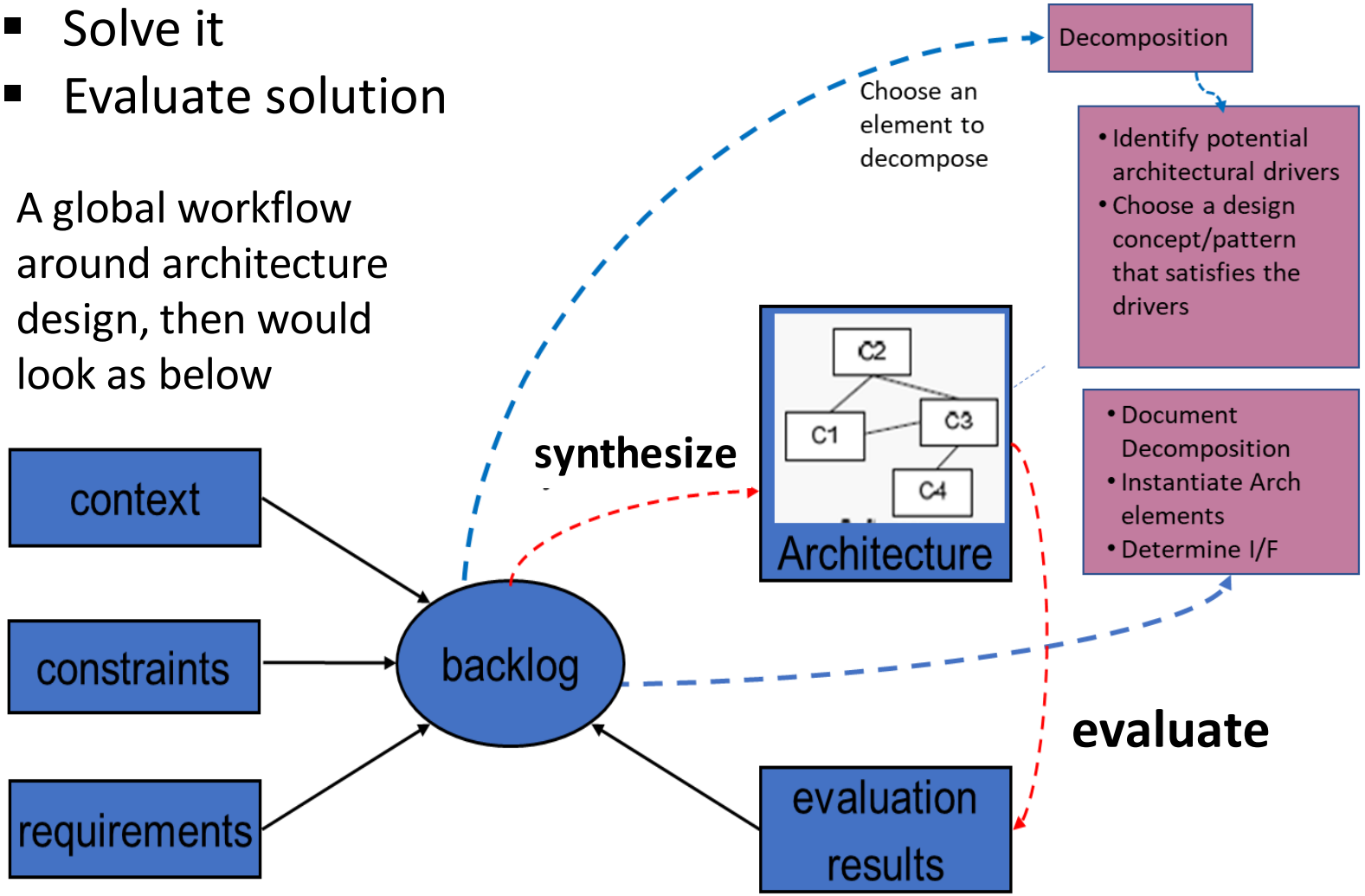
Architectural conflicts

- Using large-grain components improves performance but reduces maintainability.
- Introducing redundant data improves availability but makes security / data integrity more difficult.
  - Normalization/De-Normalization in DB
- Localizing safety-related features usually means more communication so degraded performance.
- ……

Generalized model for architecting

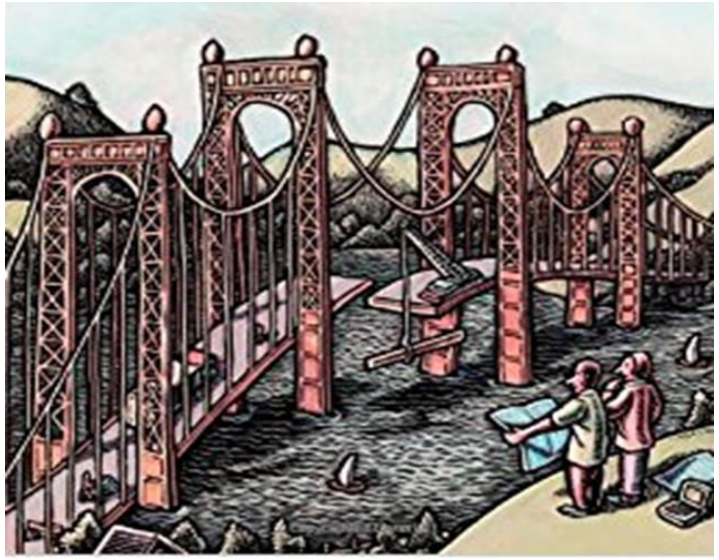
- Understand problem
- Solve it
- Evaluate solution

A global workflow around architecture design, then would look as below



PES UNIVERSITY ONLINE

Decomposition



Decomposition approaches

1. **Decomposition based on layering**
   - Ordering the system to different layers
2. **Decomposition based on Distribution among "COMPUTATIONAL RESOURCES" based on or due to system having**
   - Dedicated task owning a thread of control and hence some processes not needing to wait for this process
   - Many clients needing access
   - Needing Greater fault isolation
   - Distribution for separation of concern with redundancy which allows for high availability

Decomposition approaches

3. **Decomposition based on Exposure**

   On how the component is **Exposed** and consumes other components. This could be based on the components - Service offered, logic and Integration

4. **Decomposition based on Functionality**

   Grouping within the problem domain and separating concerns based on the **Functionality**. E.g. login module, customer module, inventory module etc. Done with the mindset of an Operational process

5. **Decomposition based on Generality**

   Considering Generality or trying to understand components which can be used in other places as well. (Not to overdo)

Decomposition Other approaches

1. **Divide and conquer** - Divide a complex problem into smaller simpler problems
2. **Stepwise refinement -** Start with a simple solution and enhance it in steps
3. **Top-down approach** - Start with an overview of the system, then detail the subsystems
4. **Bottom-up approach** - Specify individual elements in detail and then compose them together
5. **Information hiding** – E.g. Encapsulation, separation of interface from implementation

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

SOFTWARE ENGINEERING

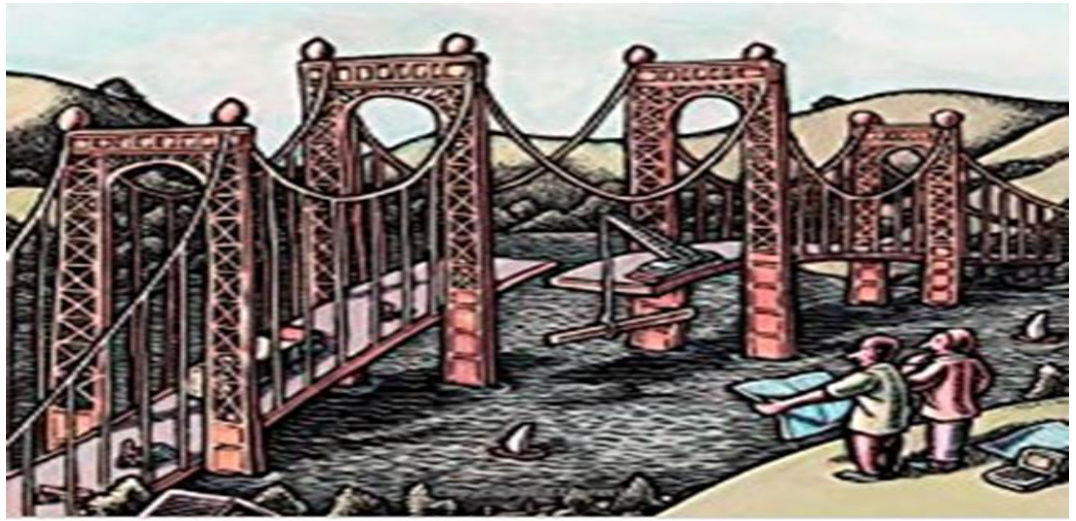
SOFTWARE ARCHITECTURE & DESIGN

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

SOFTWARE ARCHITECTURE AND DESIGN

Generic Design Methods
Architectural Views & Styles



Prof. Phalachandra H. L
Department of Computer Science and Engineering

Recap – Discussions on Architectural Approach or Model
&
Decomposition

Design Methods

- Detailed Design Methods support us in decomposing the components representing the system requirement into components/subcomponents well
- This could be structured which is development of modules and the synthesis of these modules in a so called "module hierarchy“ using some kind of charts, module specifications and a data dictionary eg. DFDs

or

could be using some object oriented approaches like Booch or Fusion etc

Design Method 1: Data Flow Design

- Yourdon and Constantine (early 70s)

- It's a two-step process:
  - 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**

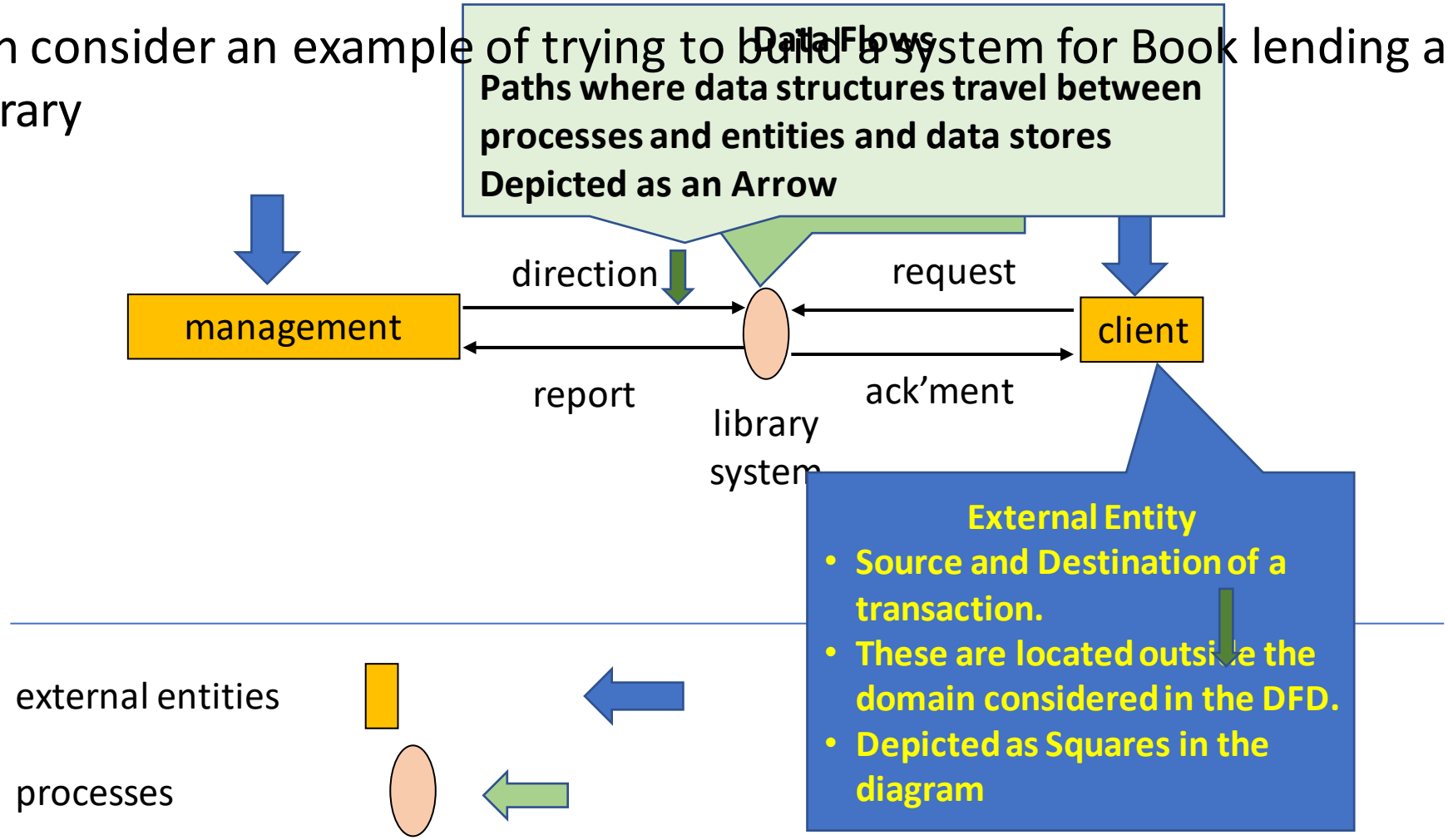
Design Method 1: Data FLOW DIAGRAMS

We typically have the following components in the DFD

- External Entity
  - Source and Destination of a transaction.
  - These are located outside the domain considered in the DFD.
  - Depicted as Squares in the diagram
- Processes
  - Transforms the data
  - Depicted as circles
- Data Stores
  - These lie between processes and are places where data structures reside
  - Depicted between two parallel lines
- Data Flows
  - Paths where data structures travel between processes and entities and data stores
  - Depicted as an Arrow

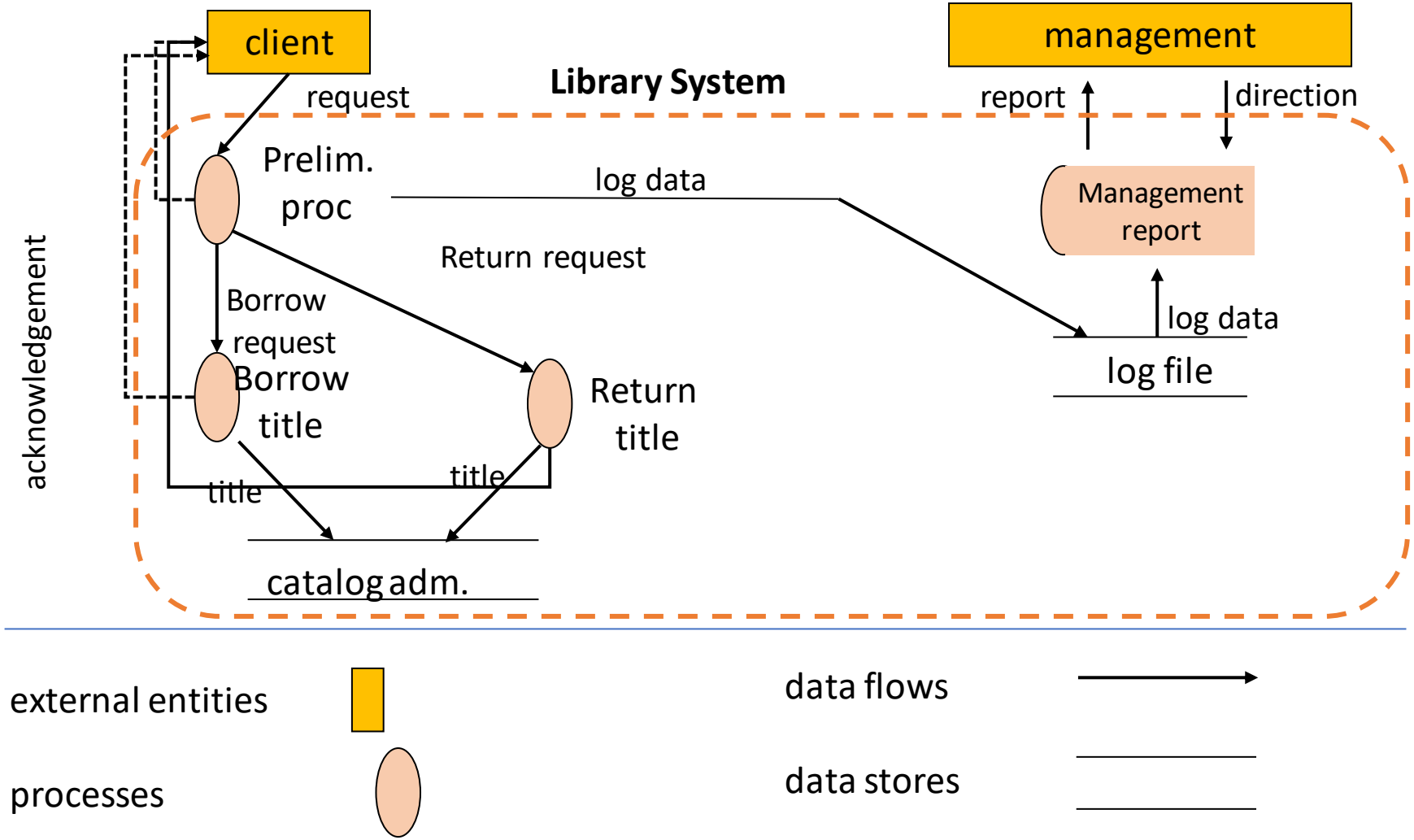
Design Method 1: Top-level DFD: context diagram

If we can consider an example of trying to build a system for Book lending and managing for a Library

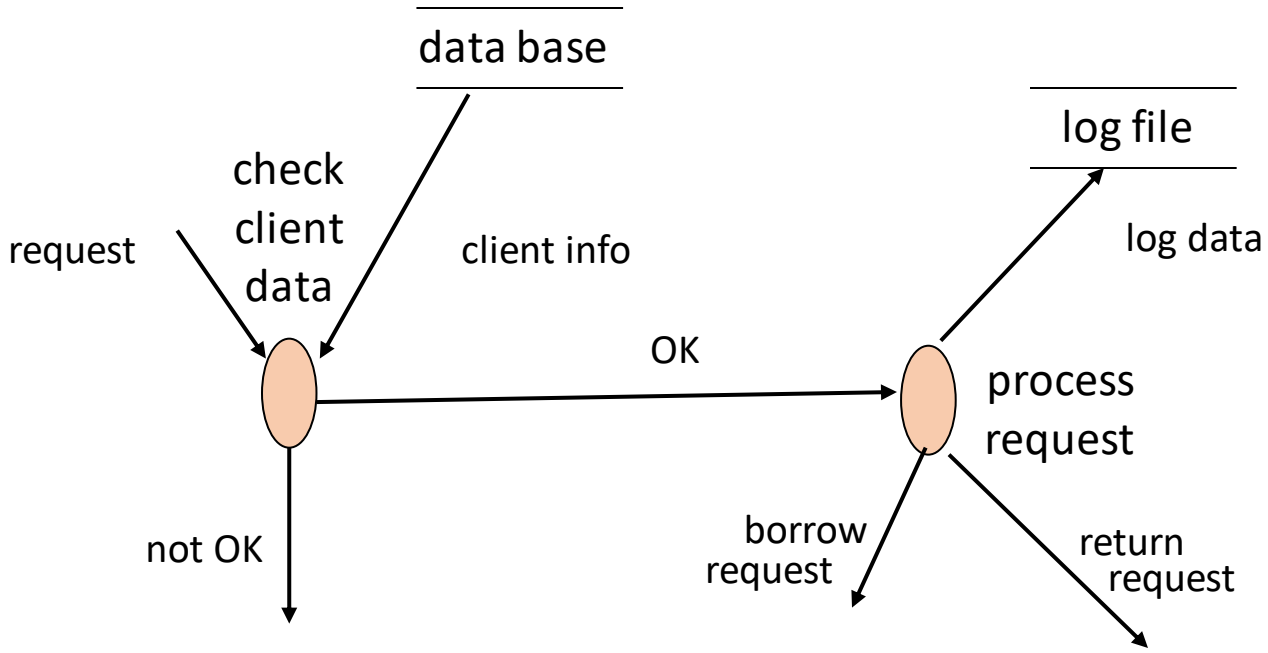


Design Method 1 :Data flow diagram – First Level Decomposition

(Book Lending Automation Example)



Design Method 1: Second-level decomposition for "preliminary processing"



Design Method 1: Example minispec

Once the process in DFDs become sufficiently straight forward and does not warrant further decomposition, then these processes are described as "minispecs" as below.
This communicates the algorithm for the relevant parties

Identification: Process request for a Book

Description:

1 Enter type of request

1.1 If invalid, issue warning and repeat step 1

1.2 If step 1 repeated 5 times, terminate transaction

2 Enter book identification

2.1 If invalid, issue warning and repeat step 2

2.2 If step 2 repeated 5 times, terminate transaction

3 Log client identification, request type and book identification

4 ...

Design Method 1: Data dictionary entries

The contents of the DFDs after we are at a logical decomposed state are recorded in a data dictionary. It's a precise description of the structure of data

borrow-request = client-id + book-id

return-request = client-id + book-id

log-data = client-id + [borrow | return] + book-id

book-id = author-name + title + (isbn) + [proc | series | other]

Conventions:

[ ]: include one of the enclosed options

|: separates options

+: AND

(): enclosed items are optional

Design Method 1: From data flow diagrams to structure charts

Structured Analysis (SA) resulted in :

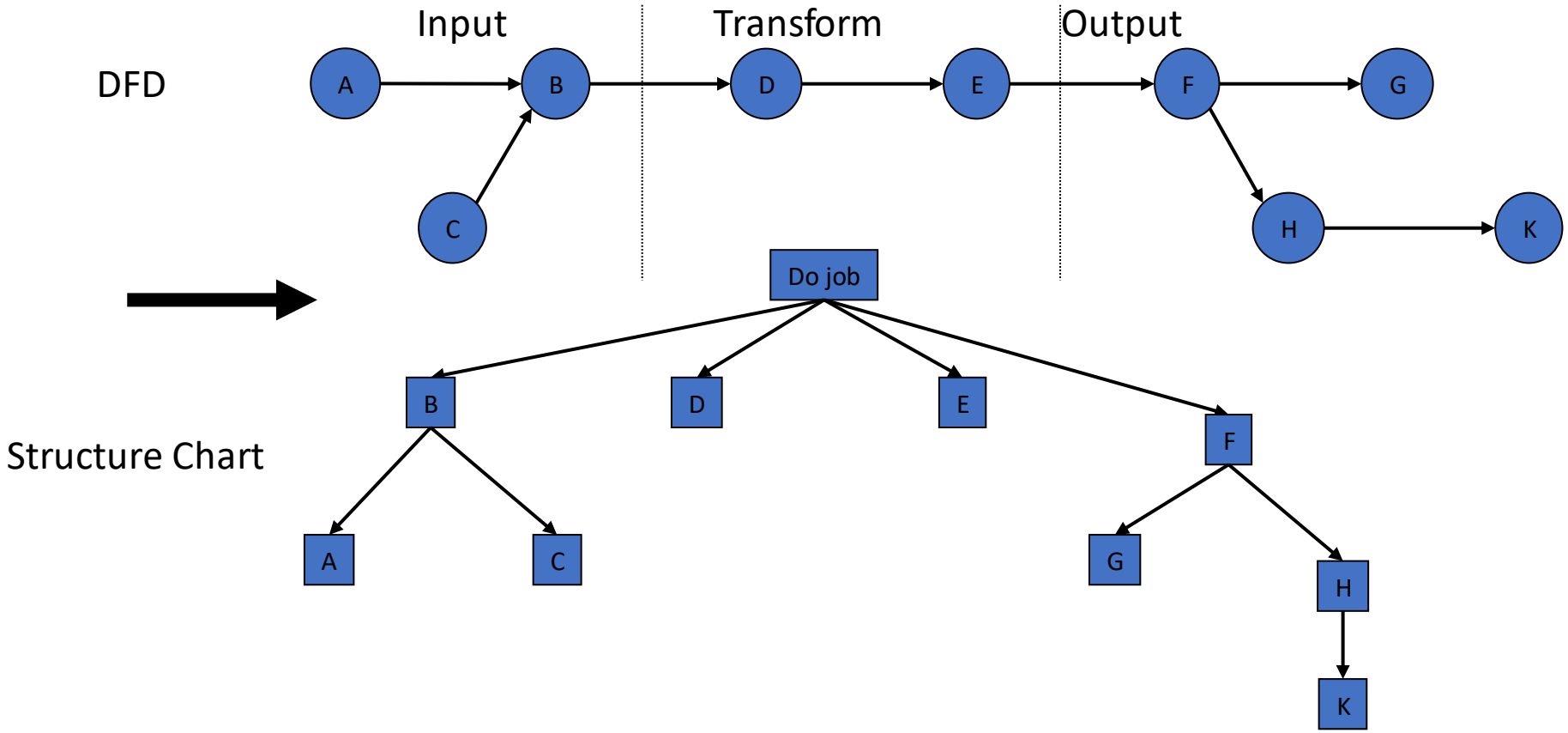
- Logical model, consisting of a set of DFD's Augmented by minispecs & data dictionary, etc.

Structured Design :

DFD's are transitioned to hierarchical structure charts

- heuristics for this transition are based on notions of coupling and cohesion
- major heuristic concerns choice for top-level structure chart, most often: *transform-centered*

Design Method 1: Transform-centered design



- Trace the input through the data flow diagram until it can no longer be considered input. Same from output ..the bubble in between is transform.
- Process in DFD becomes components of structure chart .. Data flow becomes -Component invocations

Design Method 2 : Object Oriented Booch's method

- The Booch method is a method for object-oriented software development
- Authored by Grady Booch when he was working for Rational Software (acquired by IBM) published in 1992/94
- It is composed of
  - Object modelling language (is a standardized set of symbols used to model a software system) currently UML is used for this
  - An iterative object-oriented development process
  - A set of recommended practices
- Its made up of

Macro Process

- Conceptualization : establish core requirements
- Analysis : develop a model of the desired behaviour
- Design : create an architecture
- Evolution: for the implementation
- Maintenance : for evolution after the delivery

Micro Process

- Identification of classes and objects
- Identification of their semantics
- Identification of their relationships
- Specification of their interfaces and implementation

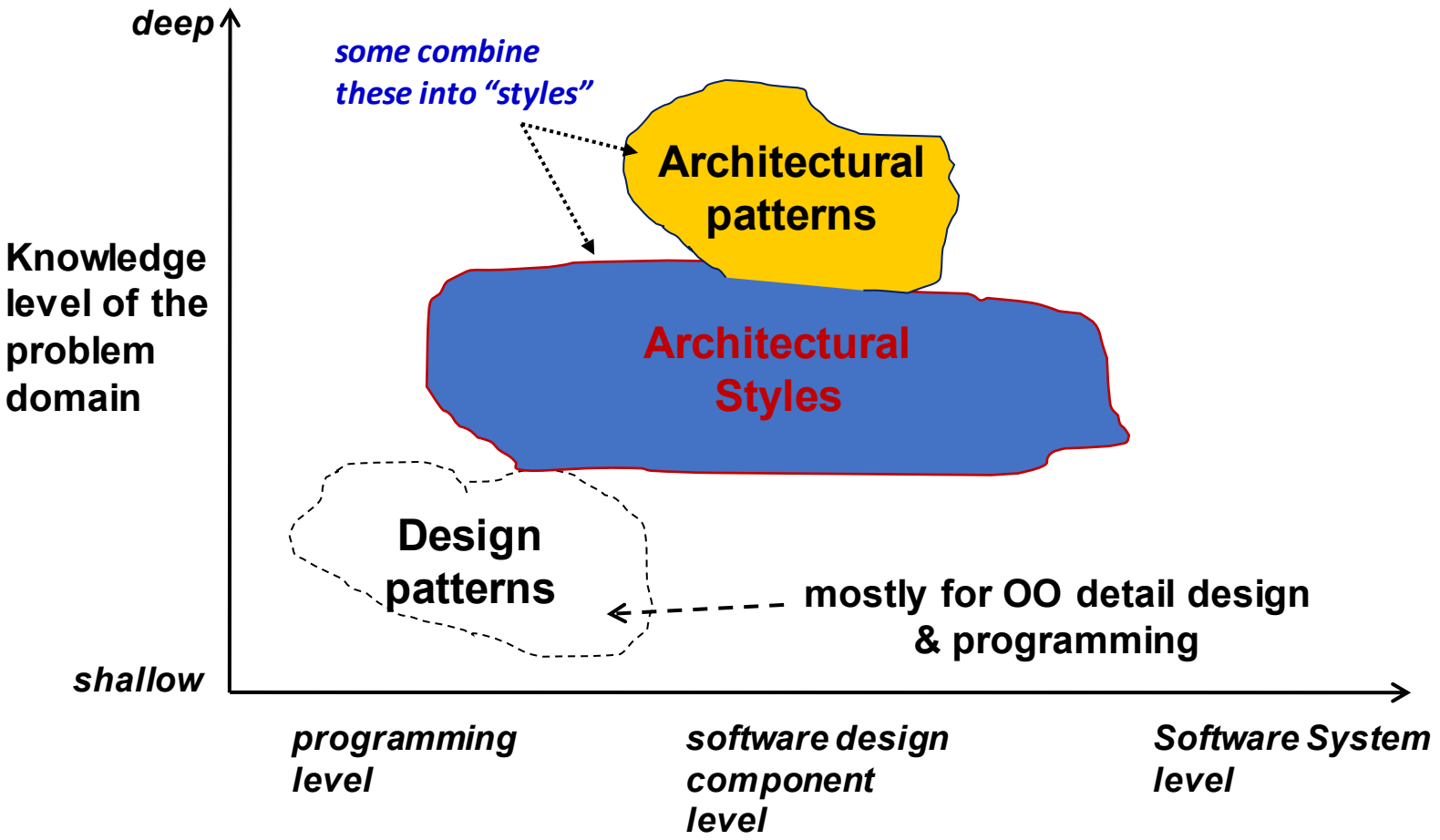
Introduction to Architectural Views, Styles and Patterns

These Architecture and Design Approaches discussed typically would need to be used by other stakeholders with different perspectives and structured towards solutions.

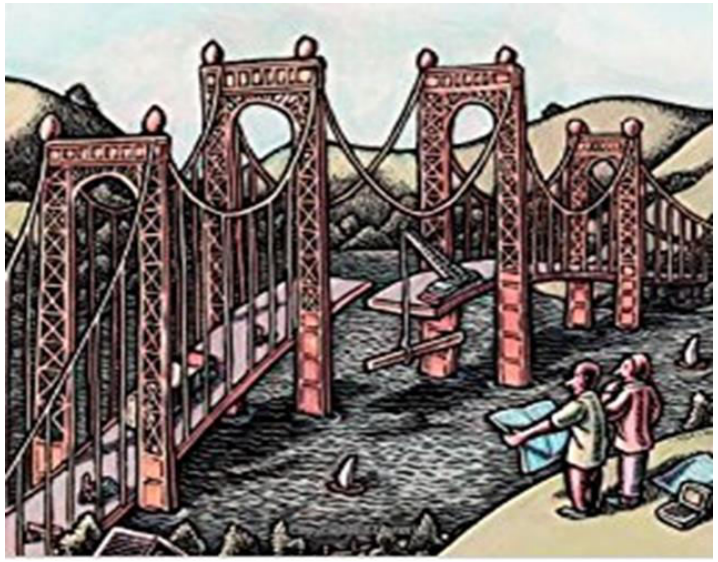
Architectural View	Architectural Styles	Architectural/Design Pattern
Views represent ways of describing the Software Architecture, enabling the system to be viewed by different stake holders in perspectives of their interest.	Demonstrates how the subsystems or elements are organized or structured It's a way of organizing code	Is a known or a proven solution to the architectural/Design problem of structuring and functioning of the subsystems which has been used earlier and is know to work for the problem scenario
E.g. UI View, Process View	E.g. Pipe & filters, Client-Server, Peer-to-Peer etc.	E.g. MVC separating UI from the rest

SOFTWARE ARCHITECTURE AND DESIGN

Combined View of Architectural Styles, Patterns and Design Patterns



Architectural Views

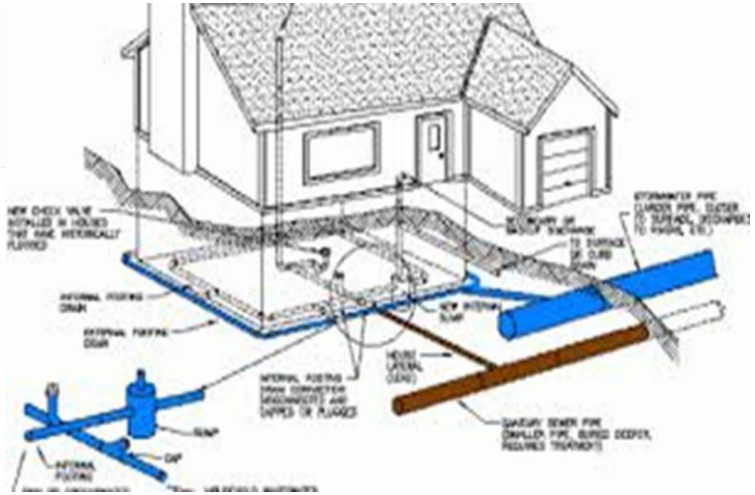


Architectural Views

Example Architectural views of a House

Elevation

Plumbing View



Plan

Architectural Views

There are 4 ways architectural elements can be viewed

1. **As a Structure of Modules (or Module View point)**
   - Modules are units of (code) Implementation with some functional responsibility
   - Structure the system as a set of code units (modules)
   - An Architect enumerates what the units of software will have to do and assigns each item to a module
   - The larger modules may be decomposed to sub-modules of the acceptable fine level by ensuring that likely changes fall within the purview of at most a few small modules.
   - It is often used as the basis for the development project's organization and deliverables like documentation.
   - There is less emphasis on how the resulting software manifests itself at runtime

2. **As a component-and connector structure (or View point)**
   - Dynamic View of considering the system in execution or runtime

     E.g. Processes view which would include a set of processes connected by synchronization links

Component or Processing Element is a software which converts inputs to outputs. It could do a computation or act as a server with state and operations or something which governs a sequence of events. E.g. filter or a controller

Data Element information needed for processing or information to be processed e.g. memory with data like a DB

Connecting Element is the glue between the components and the Data element like procedure calls, RPCs.

Architectural Views

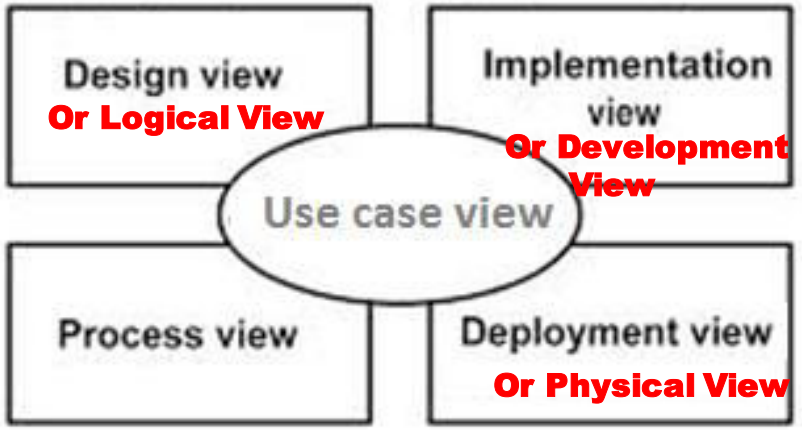
3. **As an allocation structure (or View point) includes the**

Deployment structure which shows how software is assigned to the hardware elements and which communication paths are used. Relations are "allocated-to“, showing on which physical units the software elements reside, and "migrates-to" if the allocation is dynamic. This view allows an engineer to reason about performance, data integrity, availability, and security. It is of particular interest in distributed or parallel systems

Implementation structure which indicates how software is mapped onto file structures in the system's development, integration, or configuration control environments

Work Assignment structure which shows who is doing what and helps to determine which knowledge is needed where. E.g. Functional commonality to a single team.

Architectural Views



4. **Krutchens (4+1 View) or model**
   - **Use case view** (exposing the requirements of the system or the scenarios)
   - **Design view** (exposes vocabulary of the problem space and the solution space) Class Diagrams, Sequence diagrams etc.
   - **Process view** encompasses the dynamic aspects or the runtime behavior of the system. Threads and processes that form the systems. Addresses performance, Concurrency etc.
   - **Implementation view** (addresses the realization of the system. UML diagrams like package diagrams are used)
   - **Deployment view** (focuses on system engineering issues)

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

SOFTWARE ENGINEERING

SOFTWARE ARCHITECTURE & DESIGN

Prof. Phalachandra H. L
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

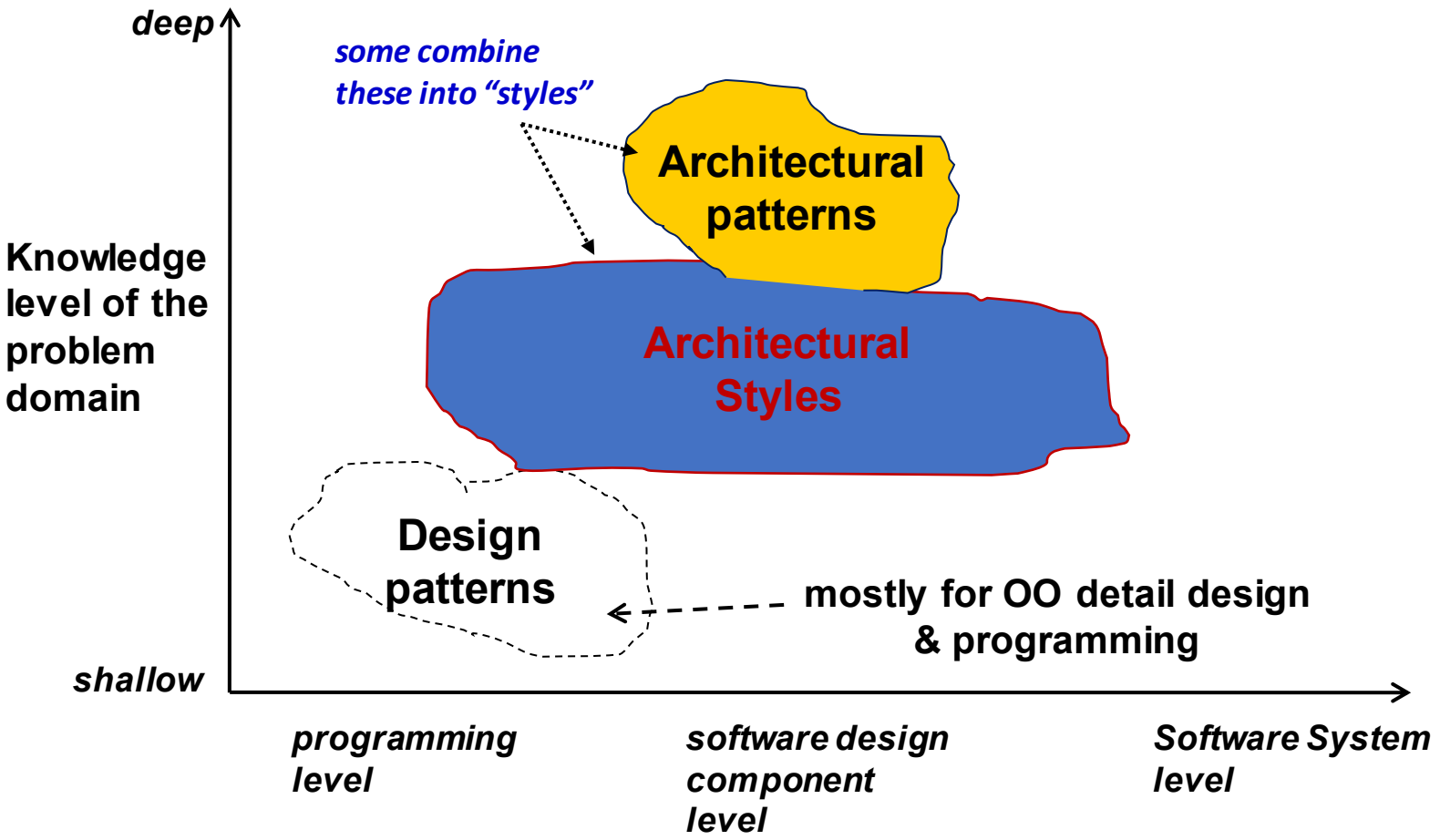
SOFTWARE ARCHITECTURE AND DESIGN

Software Architectural Styles and Architectural & Design Patterns

Prof. Phalachandra H. L
Department of Computer Science and Engineering

SOFTWARE ARCHITECTURE AND DESIGN

Combined View of Architectural Styles, Patterns and Design Patterns



SOFTWARE ARCHITECTURE AND DESIGN

Architectural Styles

Prof. Phalacha
Department of C

Architectural Styles

Can you identify this picture

Architectural Styles

Can you identify this picture

Architectural Styles

Can you identify this picture

Architectural Styles

Can you identify this picture

Architectural Styles

Can you identify this picture

Architectural Styles

Architecture from Hampi and Vijayanagar

Neo Classical Architecture

Roman Architecture

Persian Islamic Architecture

Gothic Architecture

Architectural Styles

- Architectural style is a pattern of organization of the components in an architecture, characterized by the features that make it notable.
- It can be looked at as a tool box containing tools of architecture (common patterns of software organization or recurrent structure) which could be used in constructing software modules
- It addresses the structure and behavior of the system
- It's a way of organizing modules (contrary to a pattern, it doesn't exist to "solve" a problem)

  Eg. Pipe & filter, Client/server, Main program & subroutine, Abstract Data Types / OO

Architectural Styles

It provides

- Vocabulary: A set of design elements such as pipes, filters, client, severs, parsers, databases, etc.
- Design rules: These are the set of constraints that dictates how the processing elements would be connected.
- Semantic interpretation: It provides a well-defined meaning of the connected design elements.
- Analysis: Many styles provide analyses that can be performed on systems build in that style, i.e., deadlock detection, scheduling, etc.
- This could be generically looked at as Components and Connectors

Architectural Styles

There are many recognized architectural styles like:

- main program with subroutines
- implicit invocation
- pipes and filters
- repository (blackboard)
- layers of abstraction
- Client Server (Can be looked at as a Style as well as a pattern)
- Component Based System
- Service Oriented Architecture
- Object Oriented Architecture

Architectural Styles : Main-program-with-subroutines

Generic: Traditional Language-Influenced Style

Problem: System is described as a hierarchy of functions; This is the natural outcome of the functional decomposition of the system. The top level module acts as a main program and invokes the other modules in the right order. Usually a single thread of control

Context: Language with nested procedures

Solution:

System model: Procedures and modules are defined in a hierarchy. Higher level module calls lower level modules. Hierarchy may be strict (n can only call n-1) or week (where n can call n-i) may be weak or strong, coupling/cohesion arguments

Components: Procedures which can be viewed as residing in the main program, and have their own local and global data

Connectors: procedure call and shared access to global data

Control structure: single centralized thread of control: main program pulls the strings

Architectural & Design Patterns

Architectural & Design Patterns

- Are well known or proven approaches (or a solutions to an architectural or design problem) of structuring, functioning and solutioning of the subsystems which has been used earlier and is know to work for the problem scenario
- It’s a potential solution to a recurring problem and is a good starting point
- It’s a named collection of architectural decisions or design approaches which has resulted in a successful solution
- These "solve" a problem E.g. MVC solves the problem of separating the UI from the rest
- These factor in the properties expected of architectures or design and supports with description for the same.
- Architectural pattern could be looked at from the perspective of number of layers between the user and the data as in being Tiered or Client Server etc. and Design patterns can be looked at to be procedural or object-oriented

Architectural Pattern : Tiered Architecture : Single tiered

Monolithic or Single tiered architecture : It consists of a single application layer that supports the user interface, the business rules, and the manipulation of the data all in one

Example : Microsoft Word is an example of a monolithic application. The user interface is an integral part of the application. The business rules, such as how to paginate and hyphenate, are also part of the application. The file access routines, to manipulate the data of the document, are also part of the application

Usage : This is widely used in client server application

Architectural Pattern : Tiered Architecture : Two tiered architecture

The business rules and user interface remain as part of the client application. The data retrieval and manipulation is performed by another separate application, usually found on a physically separate system or User interface remains separate and the business rules are executed on the data storage system. A client application can explicitly call a stored procedure, which would then be run on the server. A trigger can also execute a stored procedure, which is the occurrence of a specific event in the data.

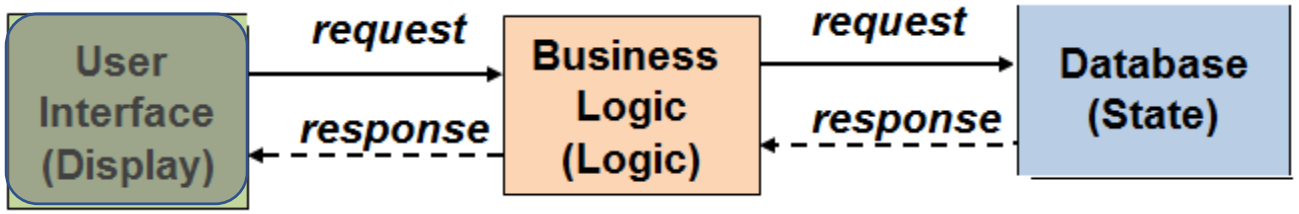
Example: An application could be something like SQL Server or Oracle, which is functioning as a data storage device for the application.

Usage: This type of application is widely used in the traditional client-server types of applications.

Architectural Pattern : Tiered Architecture : Three tiered architecture (State-Logic-Display)

State-Logic-Display is a three tier architectural pattern where

- The client (user interface), business layer and the database are separated
- The clients access business functionalities through a user "display."
- The business functionalities are provided through the "logic" part
- The data store keeps all the persistent data or "states," usually in a DB
- In this pattern, the client would never access the data storage directly



Usage:

- This type of system allows for any part of the system to be modified without having to change the other two parts.

Architectural Pattern : Client Server Pattern

Generic : The client/server architectural pattern describes distributed systems that involve a separate client and server system, and a connecting network. This could be looked at as a two tier architecture too

Client - A client is a single-user workstation that provides presentation services and the appropriate computing, connectivity and the database services and the interfaces relevant to the business need

Server- A server is one or more multi-user processors, with shared memory providing, computing, connectivity and the database services and the interfaces relevant to the business needs

Design Pattern : Procedural Patterns

These design patterns could be procedural or object oriented.
Procedural patterns are common solutions for analyzing a problem prior to & during construction.

Structural decomposition pattern	Breaks down a large system into subsystems and complex components into co-operating parts, such as ***a product breakdown structure***
Organization of work pattern	defines how components work together to solve a problem, such as ***master-slave and peer-to-peer***
Access control pattern	describes how access to services and components is controlled, such as through a ***proxy***
Management pattern	defines how to handle homogeneous collections in their entirety, such as a ***command processor and view handler***
Communication pattern	defines how to organize communication among components, such as a ***forwarder-receiver, dispatcher-server, and publisher-subscriber***

Design Pattern: Objected Oriented Pattern

These are common, reusable solutions to object-oriented programming problems. Some of these are also called Gang of Four (GOF) patterns :

- Creational patterns that focus on the creation of objects

  E.g. Singleton, Builder, Factory …

- Structural patterns that deal with the composition of objects

  E.g. Adapter, Bride, Façade, Proxy …

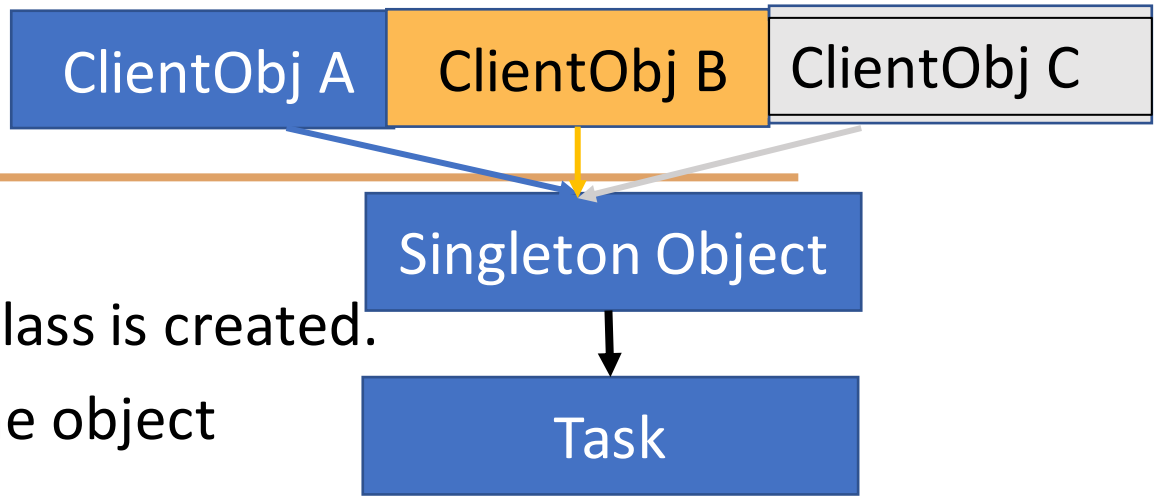
- Behavioural patterns that describe how objects interact

  E.g. Command, Interpreter, Iterator

- Distribution patterns that deal with interfaces for distributed systems

  E.g. Class Stubs and skeletons

Singleton Pattern



Intent

- Ensures that only one instance of a class is created.
- Provide a global point of access to the object

Motivation

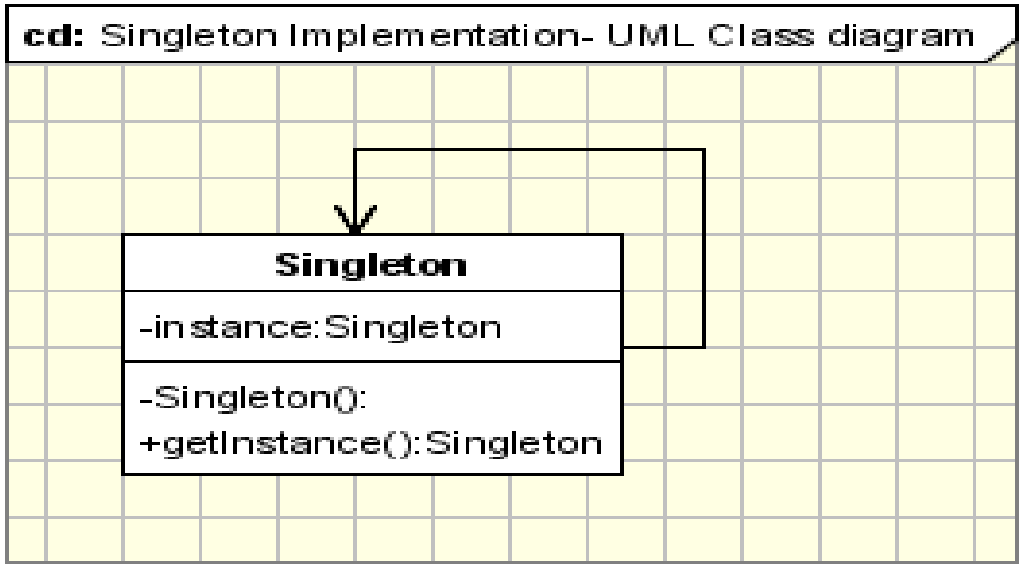
- This is useful when only one object is needed to coordinate actions across the system
- It involves only one class which is responsible to instantiate itself
- To make sure it creates not more than one instance; in the same time it provides a global point of access to that instance.
- In this case the same instance can be used from everywhere, being impossible to invoke directly the constructor each time.

Example

- Centralized management of global resources
- Examples: Logger classes, configuration classes, Print Spooler, Window Manager, Database Connection Manager

Singleton Pattern

The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member




```
class Abc
{
    static Abc obj = new Abc();
     private Abc()
     {

     }
     public static Abc getInstance()
     {
         return obj;
     }
}
```

The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is responsible for creating its class unique instance in case it is not created yet and to return that instance

Anti Pattern

- Patterns describe desirable behavior and Antipatterns describe situations one had better avoid
- In agile approaches (XP), refactoring is applied whenever an antipattern has been introduced

Eg.

God class: class that holds most responsibilities

Lava flow: dead code which gets carried forward indefinitely

..

THANK YOU

Prof. Phalachandra H.L.
Department of Computer Science and Engineering
phalachandra@pes.edu

OOAD-SE

CLASS DIAGRAM

Slide 1- 16 : Lecture 18
Slide 17 – End Lecture 19

Dr. H.L. Phalachandra
Leveraging some slides from
Prof. Vinay Joshi

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the

Recapping on the Design Approaches

Procedural Oriented Approach	Object Oriented Approach
System is divided into modules, which are refined into smaller modules and these are then implemented as ***functions***.	System is divided into small parts called ***objects***.
Typically follows a ***top down approach***.	Follows ***bottom up approach***.
Adding new data and function post the initial design is not easy.	Adding new data and function post the initial design is simpler
Procedural approach needs specific implementation for hiding data. If not programmed appropriately can end up being ***less secure***.	Object oriented programming provides data hiding as part of its constructs so easier for making a product to be ***more secure***.
Function or procedure centric rather than being data centric	Data centric rather than function/procedure centric.

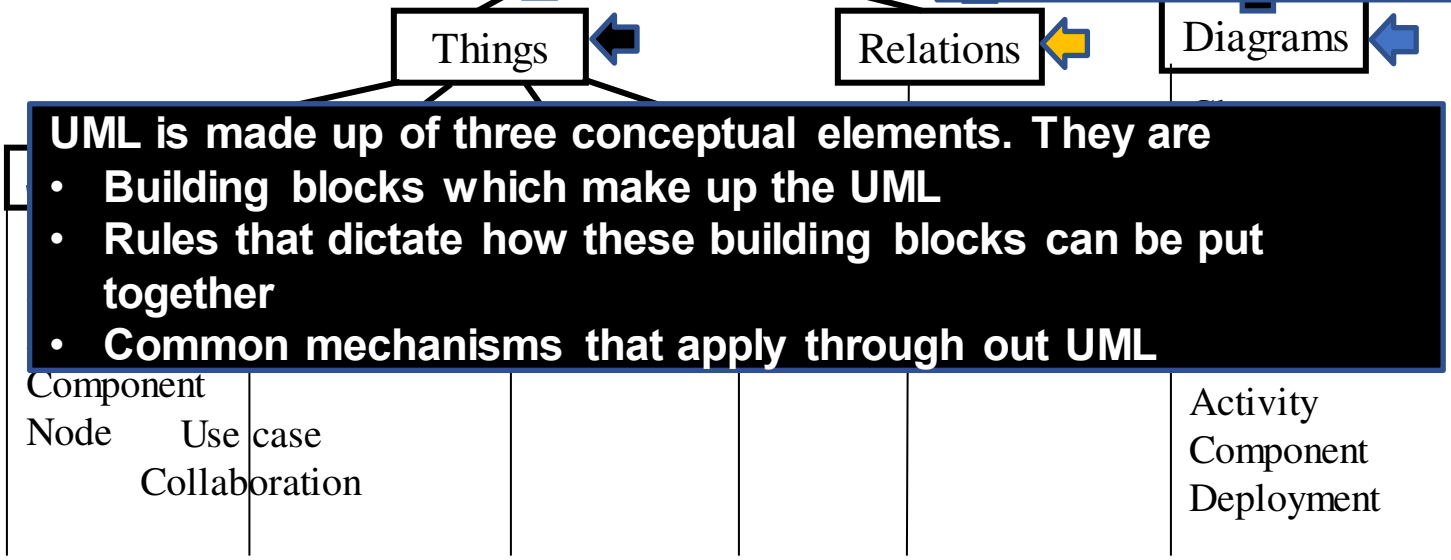
Conceptual model of UML

Things are abstractions in the model.

There are four kinds of things or abstraction

- Relationship ties together the things or abstractions
- There are four different kinds of relationships

Diagrams : Graphical representation of set of elements, often rendered as a set of connected graph of vertices (things) and arcs (relationships)
It's a projection into the system



UML is made up of three conceptual elements. They are

- Building blocks which make up the UML
- Rules that dictate how these building blocks can be put together
- Common mechanisms that apply through out UML

SOFTWARE ENGINEERING : ARCHITECTURE & DESIGN

UML Constructs or Diagrams which will be used for Object Oriented Design

State Diagram
State machine specifies the sequence of states an object or an interaction
g its lifetime in response to events and the responses to
s, transitions n
d activities wh
itions, events
internal object states

These repres
structural ele
are either cor
physical.

Component Diagram
Shows deployable components, including interfaces, ports and internal structure

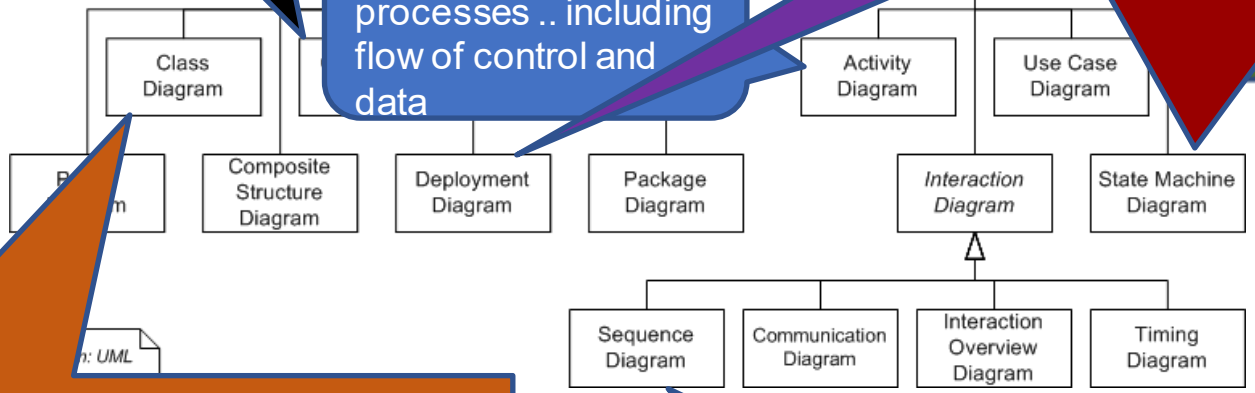
Deployment Diagram
Shows the configuration of run-time processing nodes and the components that are deployed on them

vents that

c view of

Its r d usin n.

Activity Diagram
This Shows processes .. including flow of control and data



Class Diagram
Describes a description of a set of objects that share the same attributes, operations, relationships and semantics.
Shows Classes, Interfaces, relationships

Sequence Diagram
Shows interactions between objects as a time ordered view

SOFTWARE ENGINEERING : ARCHITECTURE & DESIGN

Terms and Concepts

- System is a collection of subsystems organized to accomplish a purpose and is described by a set of models
- Subsystem is grouping of elements of which some constitute a specification of the behavior offered by the other contained elements
- Model is a schematically closed abstraction of the system (It represents a complete and ***self consistent simplification of reality*** created in order to better understand the system
- In Architecture, a ***view*** is a projection into the organization, the structure of a systems model focused on one aspect of the system
- A diagram is a graphical presentation of a set of elements most often rendered as a connected graph with vertices (things) and arcs (relationships)

Or

System represents the thing you are developing ,viewed from different perspectives by different models, with those views presented in the form of a diagram

SOFTWARE ENGINEERING : ARCHITECTURE & DESIGN

Generics of a Class Diagram : Object

Consider this glass as an object

Attributes of the object describes one or more of the characteristics of the object eg.

Summary

An object has

Attributes and values of attributes = state

Operations = behavior

An object is characterized by

Its state and behavior

- **Operation/Behavior**
  - It can be
    - Broken
    - Filled
    - Emptied
- **Attributes**
  - Its height is ..
  - It has/has not a leg
  - It is % full
- **Value/State**
  - Its height is 20 cm
  - It has a leg
  - It is half full

Generics of a Class Diagram : Object (Contd.)

- Object is a concept, abstraction or thing with identity that has a meaning for the application
- Objects appear as proper nouns or specific references in problem descriptions and discussion with users
- Objects can be concrete (person, store, car) or could be conceptual or abstraction (strategy, plan, layout)
- objects have a unique identity that remains unchanged throughout their lifetimes.
- Objects can play the same or different roles in respect of each other. Same role would have multiplicity
- A candidate key is a minimal set of attributes that uniquely identifies an object (or link)

Generics of a Class Diagram : Class

- A class describes a group of objects with the same properties (attributes), behaviors (operations), kinds of relationships and semantics

  Eg. Person – has a name, birthdate and may work in a job

  process – has an owner, priority, list and list of required resources

- Classes appear as common nouns and noun phrases in problem descriptions and discussions with users which needs to be modelled in the software solution
- An object would be an instance of the class. Objects in a class differ in terms of their attribute values and relationship to other objects
- Objects in a class share a common semantic purpose
- A class has responsibilities. A responsibility is something that instances of the class should be able to fulfill
- The purpose of class modelling is to describe objects
- **Class abstracts objects and hence helps in generalizing from a few specific cases to a host of similar cases**

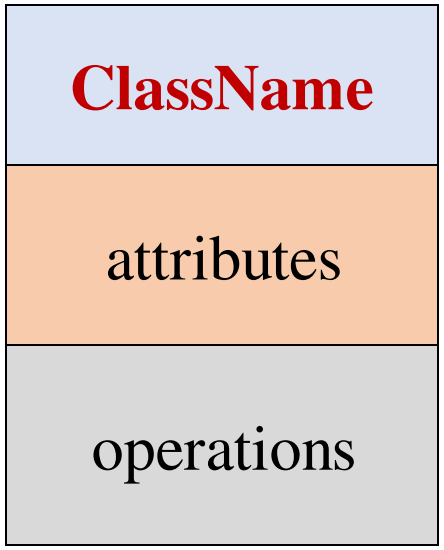
CLASS DIAGRAM

Generics of a Class Model

- A class model captures the static structure of a system by characterizing the objects in the system, the relationships between the objects and the attributes and operations for each class of objects
- ***Class diagram*** graphically represents a Class model which characterizes the objects in a system and provides an intuitive graphic representation of a system
- A class represents an identical set of objects in a system and there are multiple classes which would exist in a system
- Focus is on looking at the problem domain in terms of objects/Classes rather than functionality and Class modelling will need to identify the classes which are needed to be implemented in the system to meet the requirements identified
- A class diagram needs to represent the
  - Classes and the relationship between the classes (how do the classes relate to each other)
  - The attributes (fields) or named properties of these classes
  - The operations which the class will need to support (methods for the class)

CLASS DIAGRAM

Classes



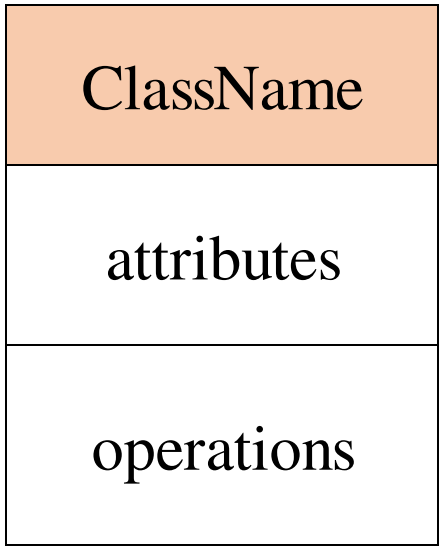
A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

An object would be an instance of the class

CLASS DIAGRAM

Class Names



The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment and is centered.

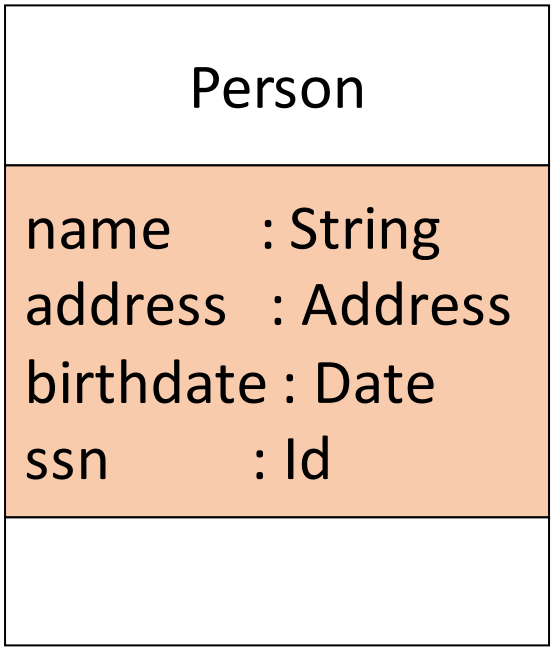
A textual string derived from vocabulary of the system to be modeled – typically a noun

Qualifier might represent inheritance: Savings Account, Current Account

Typically ClassNames are singular and starts with a capital letter and intervening words again starting with capital letters

CLASS DIAGRAM

Class Attributes



An *attribute* is a named property of a class that describes the values the object instance can hold

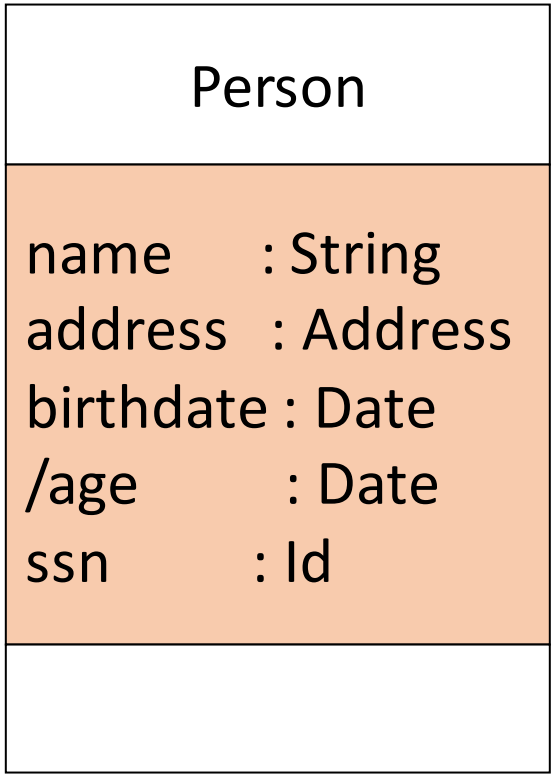
Property of all instances of the class:

Name, address

In the class diagram, attributes appear in the second compartment just below the name-compartment.

CLASS DIAGRAM

Class Attributes (Cont'd)



Attributes are usually listed in the form:

attributeName : dataType

or

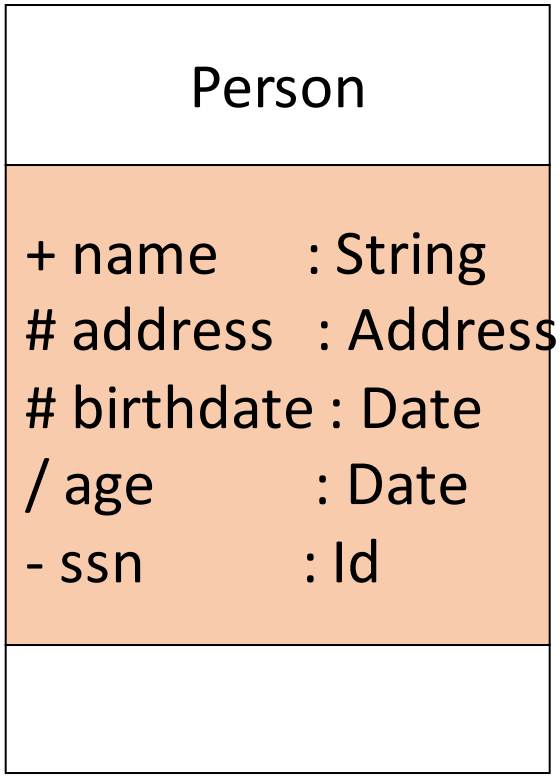
attributeName : dataType = default value

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age (or Order Total).

A derived attribute is designated by a preceding '/'

CLASS DIAGRAM

Class Attributes (Cont'd)



Attributes can have visibility be:

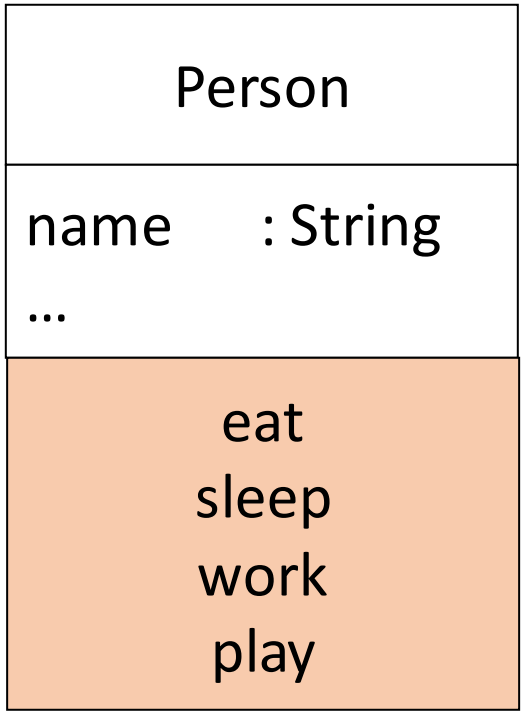
+ public
# protected
- private
/ derived

Some of the attributes which characterize Public, Private and Protected

- Levels of Accessibility
- Provision of encapsulation
- Provision of method overriding

CLASS DIAGRAM

Class Operations



Operations describe the class behavior and appear in the third compartment.

An operation is a function or a procedure that may be applied to or by objects in a class.

Each operation has a target object as an implicit argument

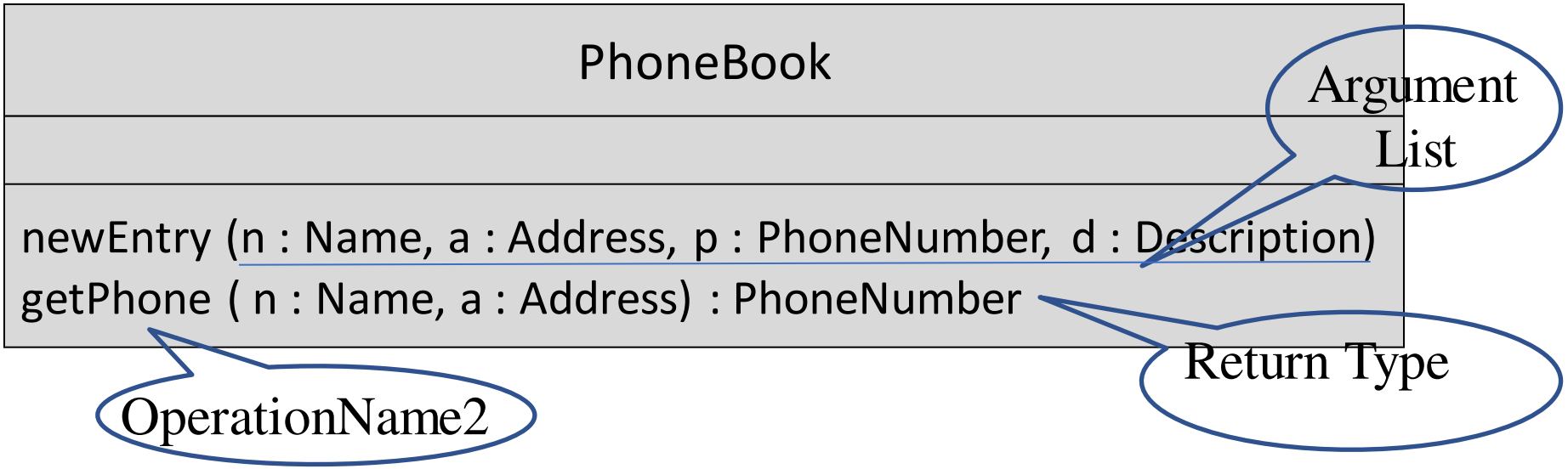
Method is an implementation of an operation on a class

It can also be looked at as an Implementation of a service, invoked by client

Invocation may change state of object

CLASS DIAGRAM

Class Operations (Cont'd)



You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.

There could be a direction for the operation argument which is optional

direction argumentName : type = defaultValue

direction could be in, out or inout (an input which can be modified)

Not all attributes or operations need to be shown. Use … to indicate.

CLASS DIAGRAM

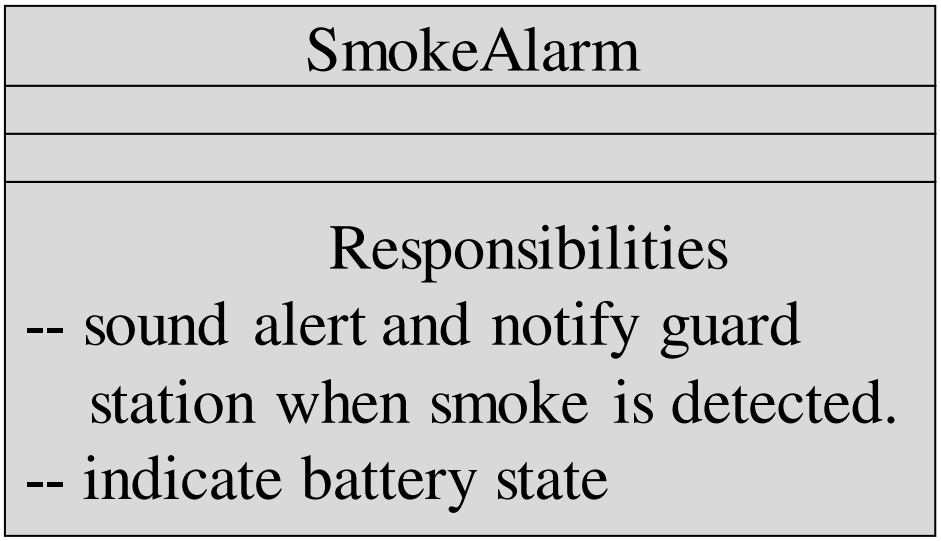
Class Responsibility

Responsibility :

A responsibility is a contract or obligation of a class to perform a particular service.

It's a free form text written as a phrase or a sentence

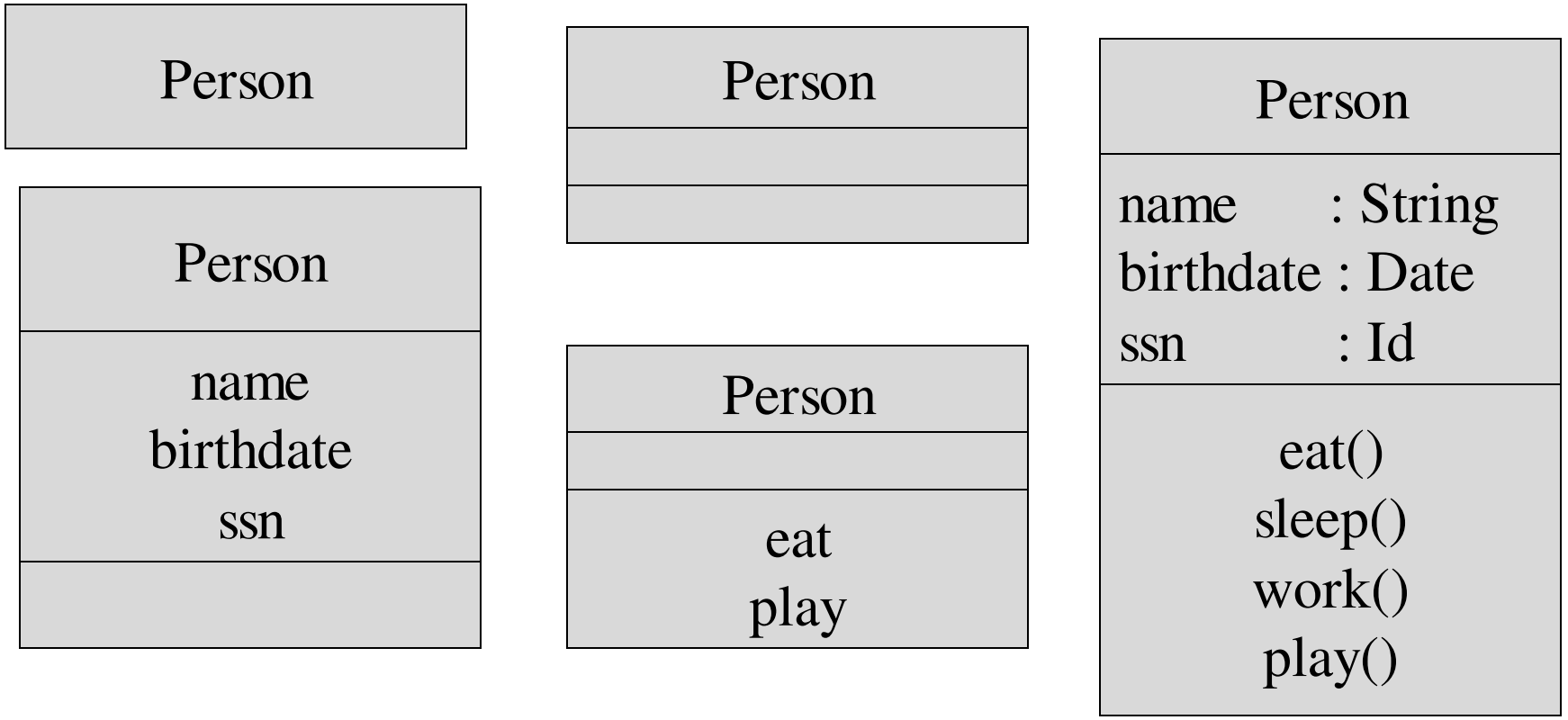
Eg. Determine risk of the customer order



CLASS DIAGRAM

Depicting Classes

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



CLASS DIAGRAM

Relating Classes with Use Cases using CRC

(Class-Responsibility-Collaborators)

CRC cards are tool used for brainstorming in OO design

CRC cards are created from Index cards and each member of the brain storming session with write up one CRC card for each relevant class/object of their design

A typical CRC card would look as shown

Objects need to interact with other objects (Collaborators) in order to fulfill their responsibilities

Since the cards are small, prevents to get into details and give too many responsibilities to a class

They can be easily placed on the table and rearranged to describe and evolve the design

Class Name	
Responsibilities	Collaborators

Class: Account	
Responsibilities	**Collaborators**
Know balance	
Deposit Funds	Transaction
Withdraw Funds	Transaction, Policy
Standing Instructions	Transaction, Standing Instruction, Policy, Account

CLASS DIAGRAM

Modelling the Class Collaborations

- Classes (from the vocabulary of the system) do not stand alone but works with other classes to carry out some semantics which is bigger than the individual classes
- Class diagrams need to depict the visualization, specification and construction of various collaborations with these classes
- Class diagram models one part of the things and hence each class diagram should focus on one collaborations at a time
- To model a collaboration
  - Identify the function or behavior of the part of the system that is represented by the group of classes
  - For each of these functions, identify the classes, interfaces and other collaborations
  - Walk through these using scenarios, which will help discover new classes and also identify semantically wrong ones
  - For each of these classes, identifying their contents and responsibilities help to visualize and balance the responsibilities and as a cascade attributes and operations
  - As part of the course, we would be looking at a scenario and discussing the same

CLASS DIAGRAM

OO Relationships

Relations tie together the different classes/objects/abstractions together

1. **Association**

   This is a structural relationship that describes a set of links which is a common connection among objects

   1. Aggregation
   2. Composition

2. **Generalization/Specialization**

   Its a relationship where objects of specialized elements are substitutable for the objects of the generalized elements. Child shares the structure and behavior of parents

Relations tie together the different classes/objects/abstractions together (cont.)

3. **Dependency**

   Semantic relationship between two things in which a change to one thing may effect the semantics of the other thing.

4. **Realization**

   - Its a relationship between classifiers.
   - One specifies and other carries out.
   - Typically found in interfaces and the classes which realize them.

CLASS DIAGRAM : OO Relationships

1. Association

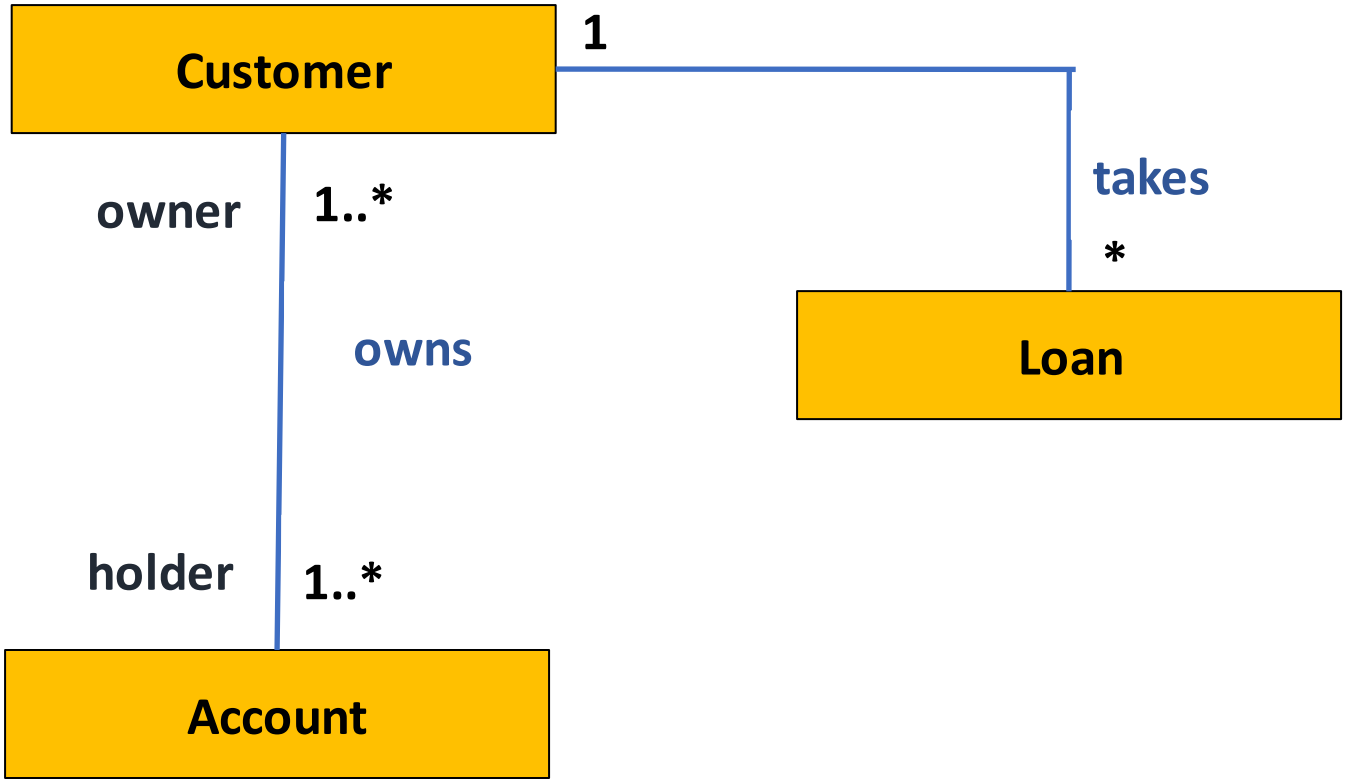
- An *association* describes a relationship between instances of one class and instances of the same or another class represented as a connection
  - Customer *owns* Account
  - Student *enrolls for* Elective
  - Courses have students
  - Courses have exams

  Each of the connections are called links. Typically implemented as pointers
- **Multiplicity**
  - Number of instances or objects of one class that can be associated with one instance of another
  - Constraints the number of related objects and depends on how you define the boundaries of the problem
  - UML diagram explicitly list multiplicity (constraint while cardinality is the count) at the end of the association lines

CLASS DIAGRAM : OO Relationships

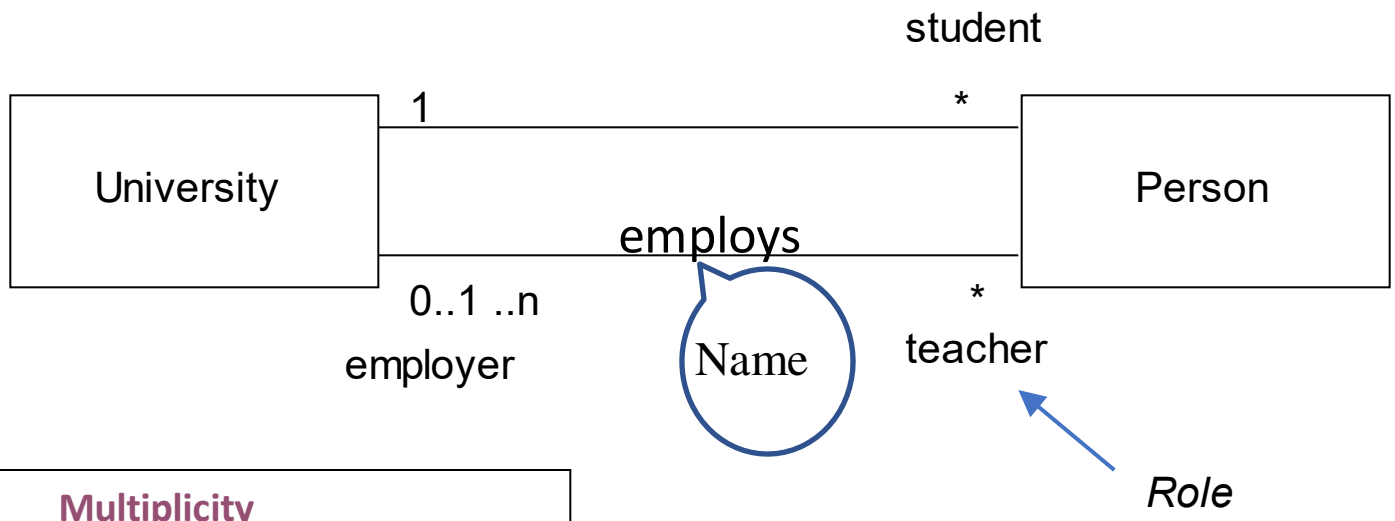
1. Association (Contd.)

- Description of an Association
  - Association has two ends
  - Association ends may be named based on the role or the purpose of the association e.g. Enrolled Student, Course Enrolled
  - Navigability (unidirectional, bidirectional links)



CLASS DIAGRAM : OO Relationships

1. Association : Name, Multiplicity and Roles



Multiplicity

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

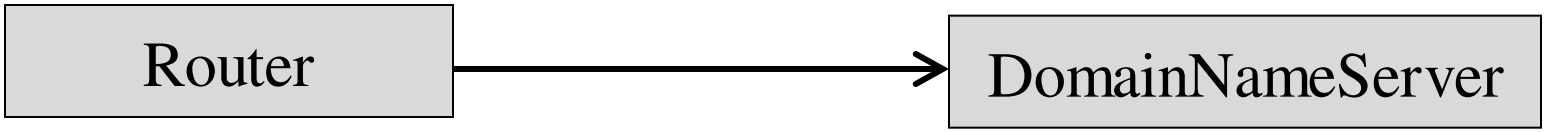
Role

"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

CLASS DIAGRAM : OO Relationships

1. Association Relationships (Contd.)

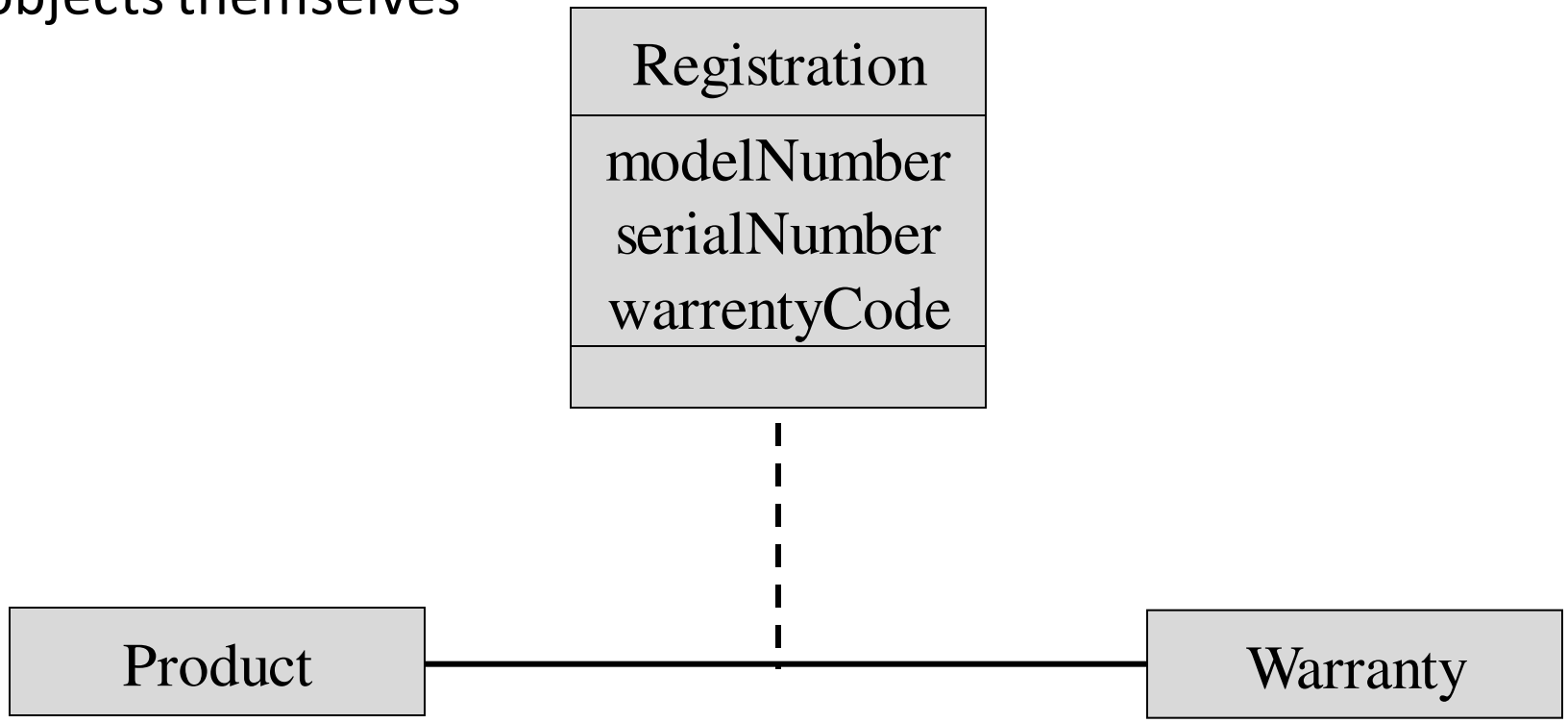
We can constrain the association relationship by defining the *navigability* of the association. Here, a *Router* object requests services from a *DNS* object by sending messages to (invoking the operations of) the server. The direction of the association indicates that the server has no knowledge of the *Router*.



CLASS DIAGRAM : OO Relationships

1. Association Relationships (Contd.)

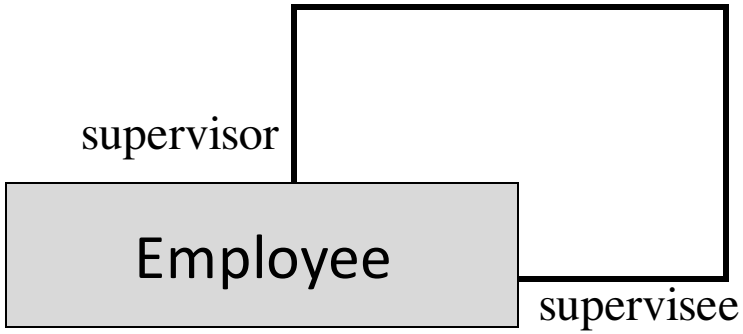
There can be situations where Associations may need a link classes or association classes ..like registration below .. can also be objects themselves



CLASS DIAGRAM : OO Relationships

1. Association Relationships (Contd.)

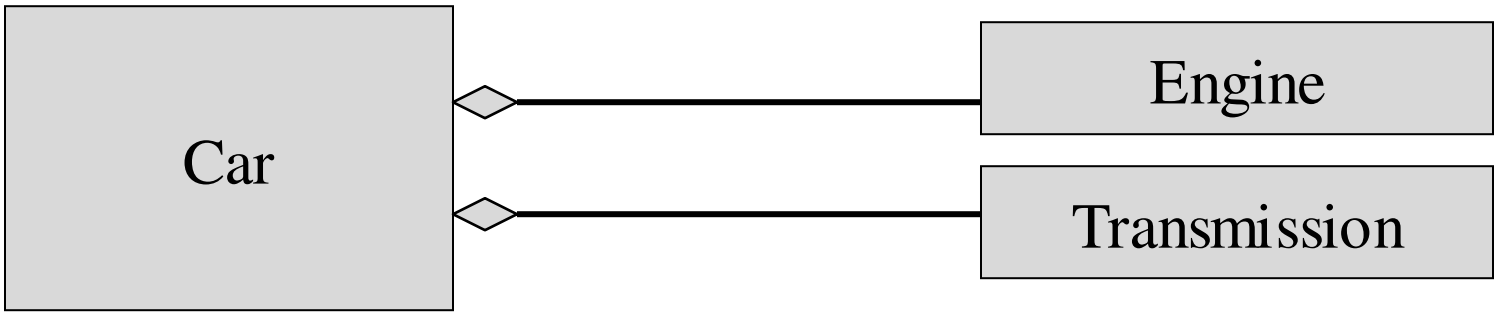
A class can have a *self association*.



CLASS DIAGRAM : OO Relationships

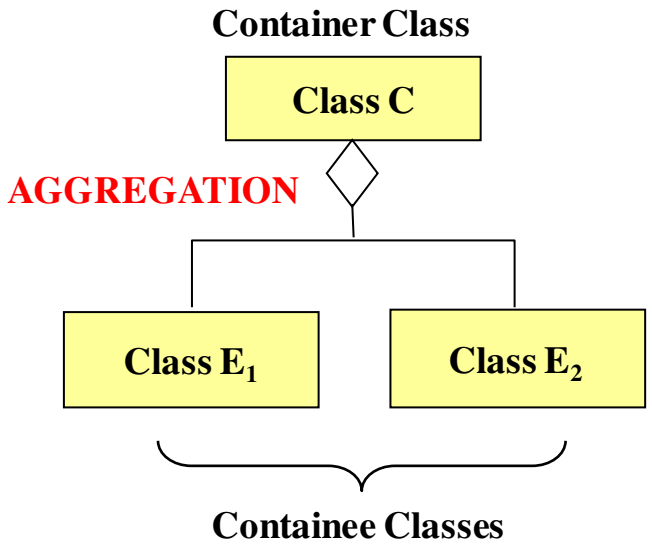
1. Association Relationships : Aggregation

- An aggregation is a strong form of association in which objects are assembled or configured together to create a more complex object.
- An aggregation describes a group of objects and how you interact with them. It defines a single point of control, called the aggregate that represents the assembly and decides on how the assembled objects respond to changes or instructions that might affect the collection.
- It shows the class that takes the role of the whole, is composed (has a) of other classes, which take the role of the parts
- An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment or the association.

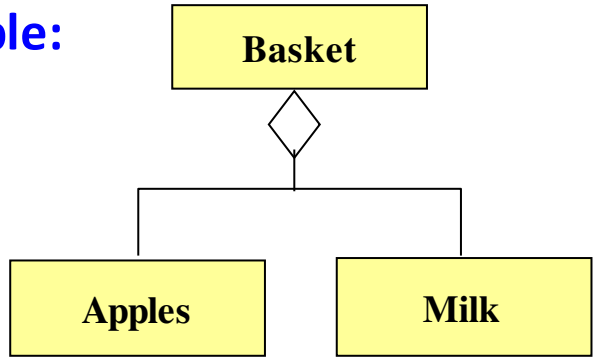


CLASS DIAGRAM : OO Relationships

1. Association Relationships : Aggregation



Example:



[From Dr.David A. Workman]

Aggregation:
expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

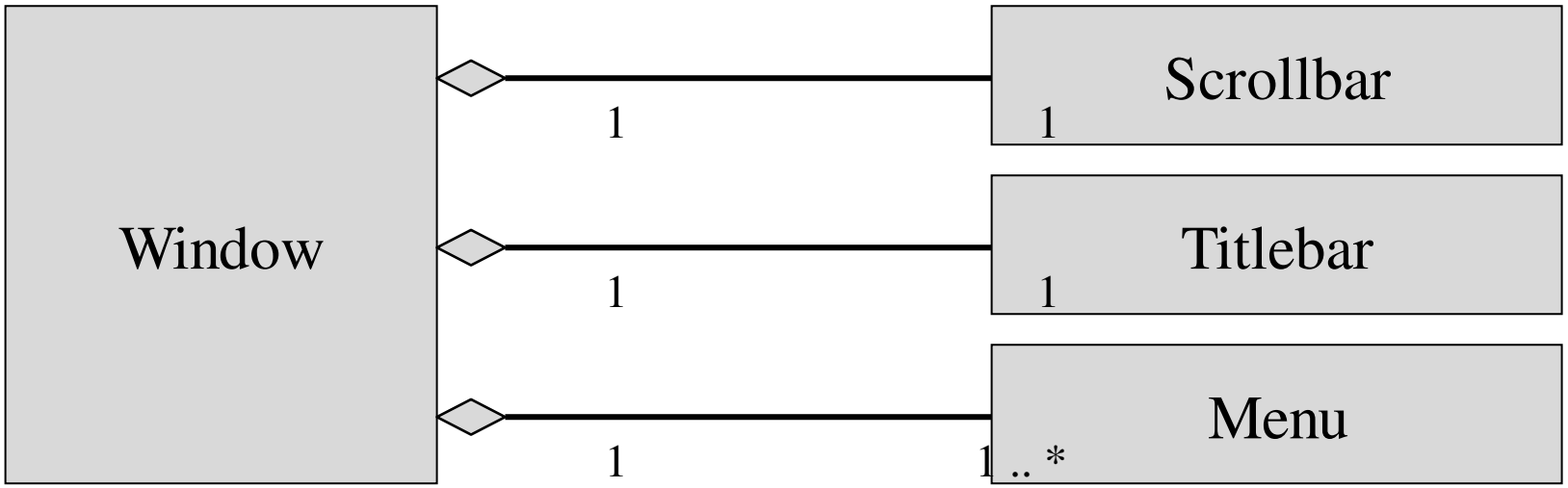
express a more informal relationship than composition expresses.

Aggregation is appropriate when Container and Containees have no special access privileges to each other.

CLASS DIAGRAM : OO Relationships

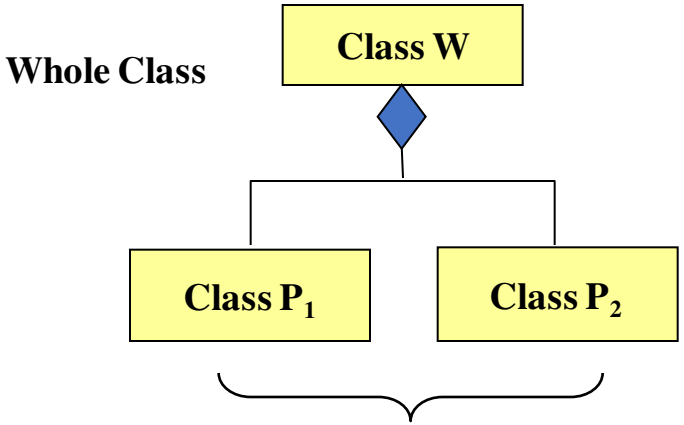
1. Association Relationships : Composition

A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



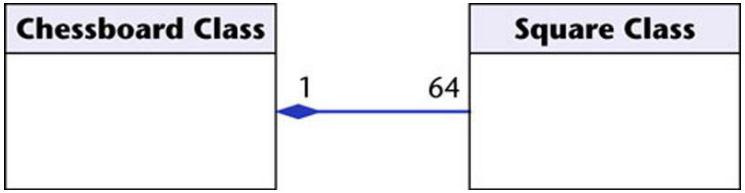
CLASS DIAGRAM : OO Relationships

1. Association Relationships : Composition



[From Dr.David A. Workman]

Example:



Association

Models the part–whole relationship

Composition

Also models the part–whole relationship but, in addition, Every part may belong to only one whole, and If the whole is deleted, so are the parts

Example:

A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.

CLASS DIAGRAM : OO Relationships

1. Association Relationships : Aggregation vs. Composition

- **Composition** is really a strong form of **association**
  - components have only one owner
  - components cannot exist independent of their owner
  - components live or die with their owner

  e.g. Each car has an engine that can not be shared with other cars.

- **Aggregations**
  - May form "part of" the association, but may not be essential to it. They may also exist independent of the aggregate.

  e.g. Apples may exist independent of the bag.

CLASS DIAGRAM : OO Relationships

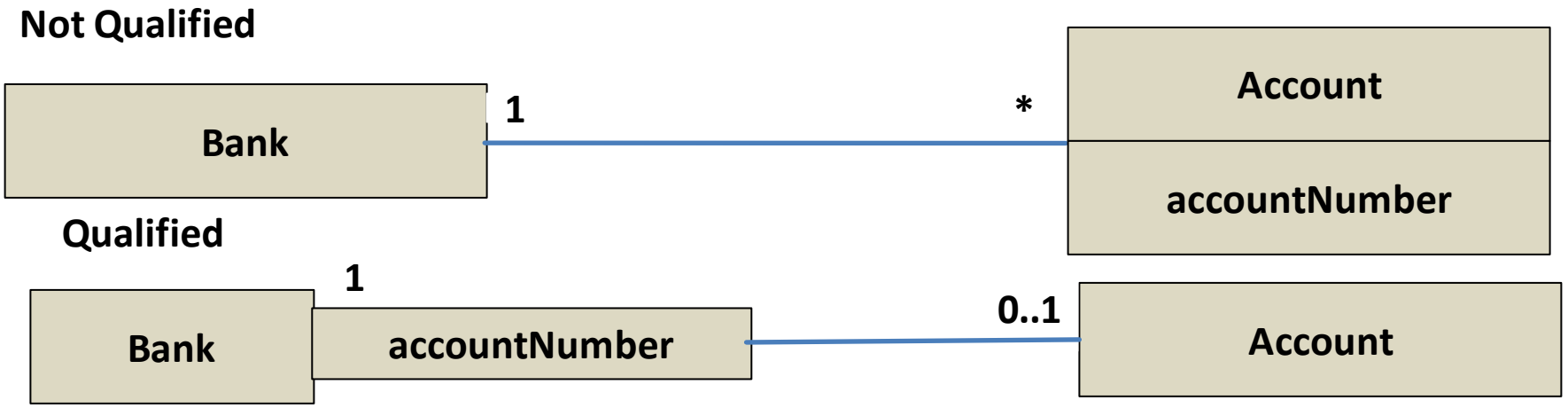
1. Association Relationships : Qualified Association

It is an association in which an attribute called 'qualifier', disambiguates the objects for a "many " association end

- Qualifier reduces the effective multiplicity from many to one ,

  Eg: for association with one to many multiplicity

  - Bank services multiple accounts
  - Account belongs to a single bank
  - Within context of a bank , account number specifies unique account

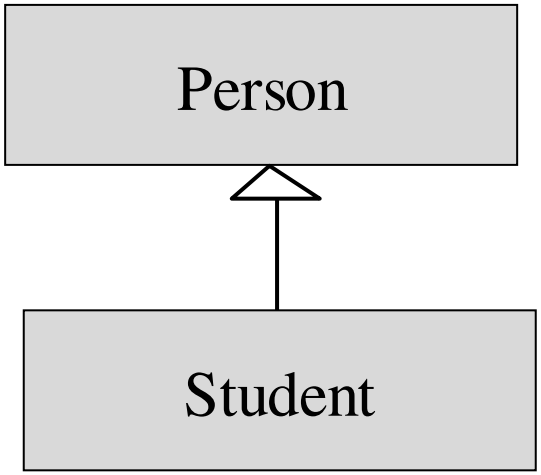


Attribute called ***qualifier*** disambiguates the objects for a "many" association end

CLASS DIAGRAM : OO Relationships

2. Generalization Relationships

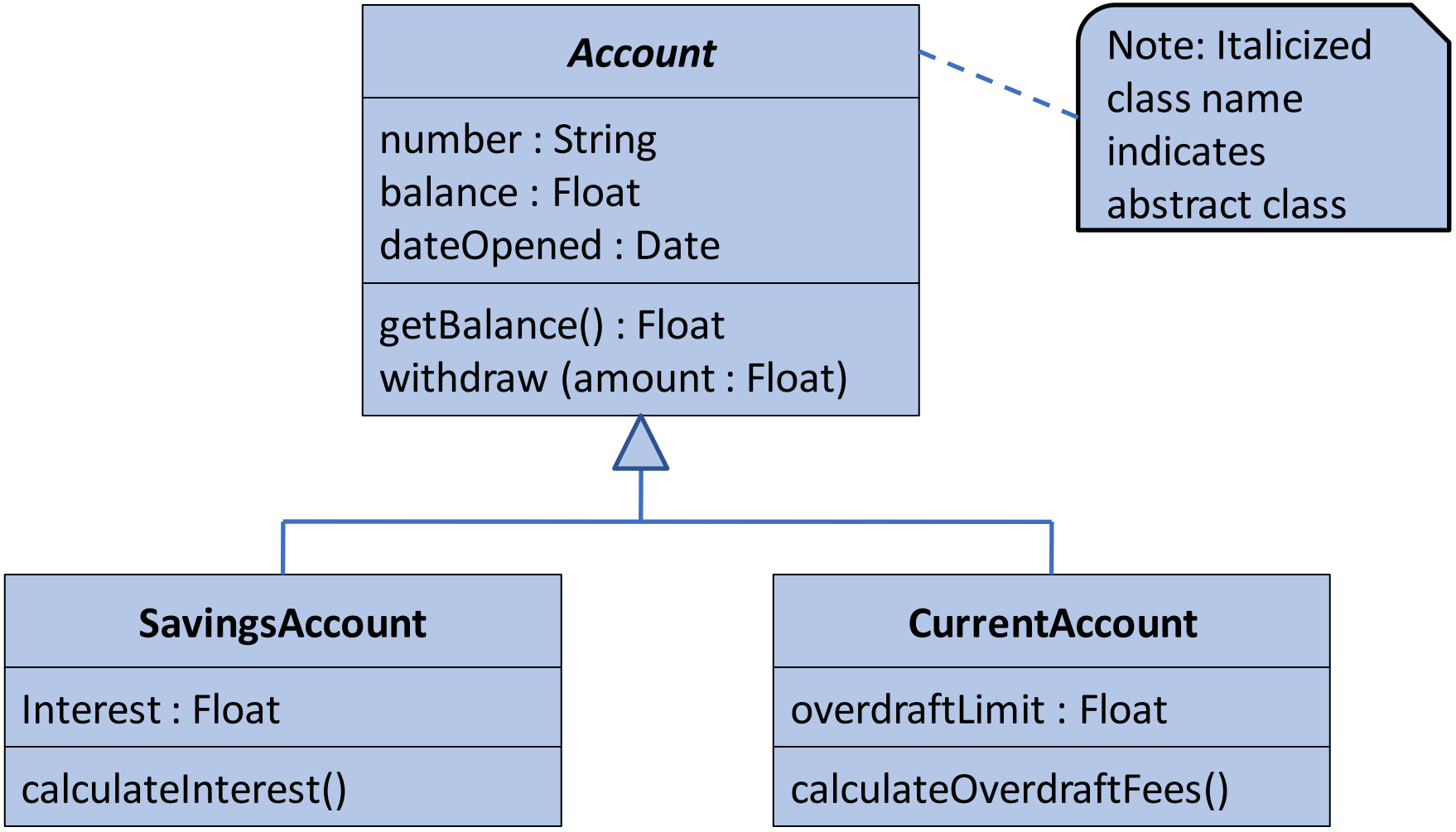
- Generalization (conceptual) in UML represents Inheritance (which is more of a language mechanism) in which a class "gains" all of the attributes and operations of the class it inherits from, and can override/modify some of them, as well as add more attributes and operations of its own
- In UML, a Generalization association between two classes puts them in a hierarchy representing the concept of inheritance of a derived class from a base class
- Inheritance relationship between subclass and superclass is "Is-a" relationship



- Subclass inherits features from superclass
  - Attributes and Operations (unless protected)
  - Operations may be overridden by subclass
  - Is a specialization of the more general superclass.
- Uses of generalization
  - Polymorphism, Abstraction of common elements, Reuse of code

CLASS DIAGRAM : OO Relationships

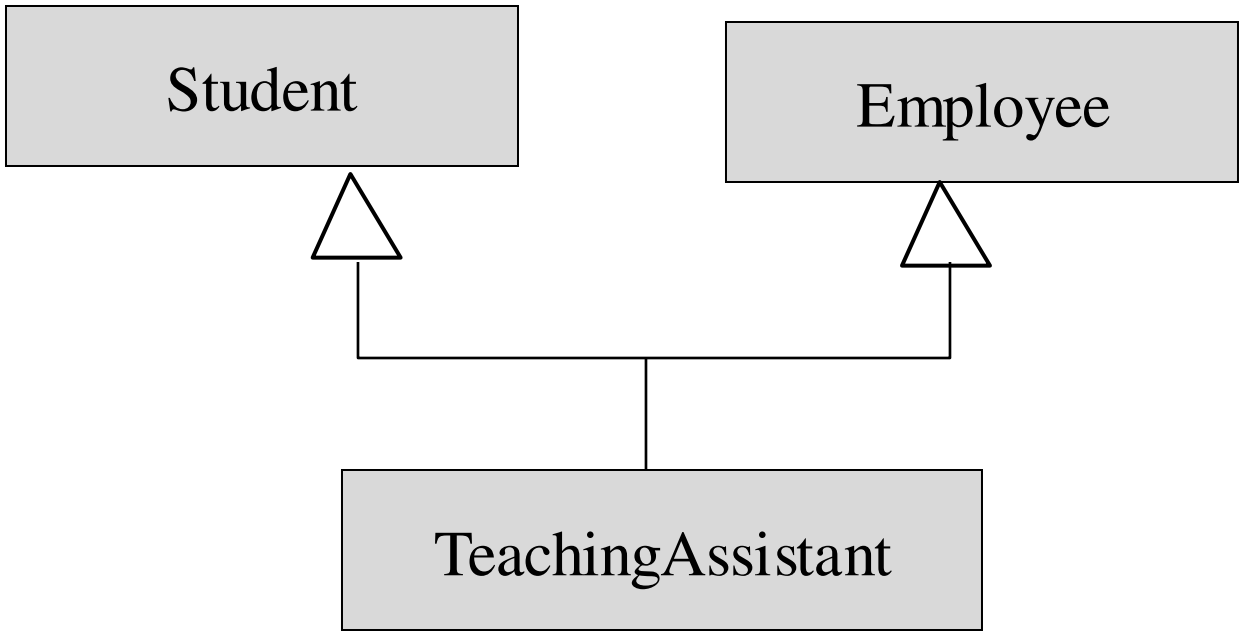
2. Generalization



CLASS DIAGRAM : OO Relationships

2. Generalization (Contd.)

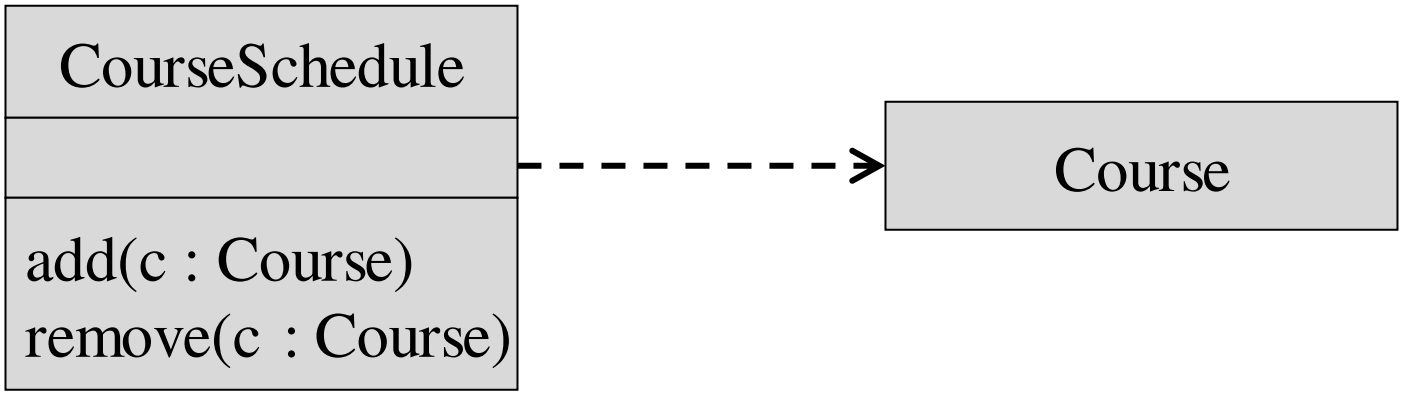
UML permits a class to inherit from multiple superclasses, although some programming languages (*e.g.*, Java) do not permit multiple inheritance.



CLASS DIAGRAM : OO Relationships

3. Dependency

- A dependency relationship is a semantic relationship in which one element, the client, uses or depends on one or more other elements, the supplier(s)
- Dependency relationship is used to represent precedence, where one model element must precede another.
- The dependency from CourseSchedule to Course exists because Course is used in both the add and remove operations of CourseSchedule.
- Dependency is displayed as a dashed line with an open arrow that points from the client to the supplier.



THANK YOU

Dr. H. L. Phalachandra
Department of Computer Science and Engineering
phalachandra@pes.edu

OOAD-SE

CLASS DIAGRAM

Slide 1- 16 : Lecture 18
Slide 17 – End Lecture 19

Dr. H.L. Phalachandra
Leveraging some slides from
Prof. Vinay Joshi

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the

Recapping on the Design Approaches

Procedural Oriented Approach	Object Oriented Approach
System is divided into modules, which are refined into smaller modules and these are then implemented as ***functions***.	System is divided into small parts called ***objects***.
Typically follows a ***top down approach***.	Follows ***bottom up approach***.
Adding new data and function post the initial design is not easy.	Adding new data and function post the initial design is simpler
Procedural approach needs specific implementation for hiding data. If not programmed appropriately can end up being ***less secure***.	Object oriented programming provides data hiding as part of its constructs so easier for making a product to be ***more secure***.
Function or procedure centric rather than being data centric	Data centric rather than function/procedure centric.

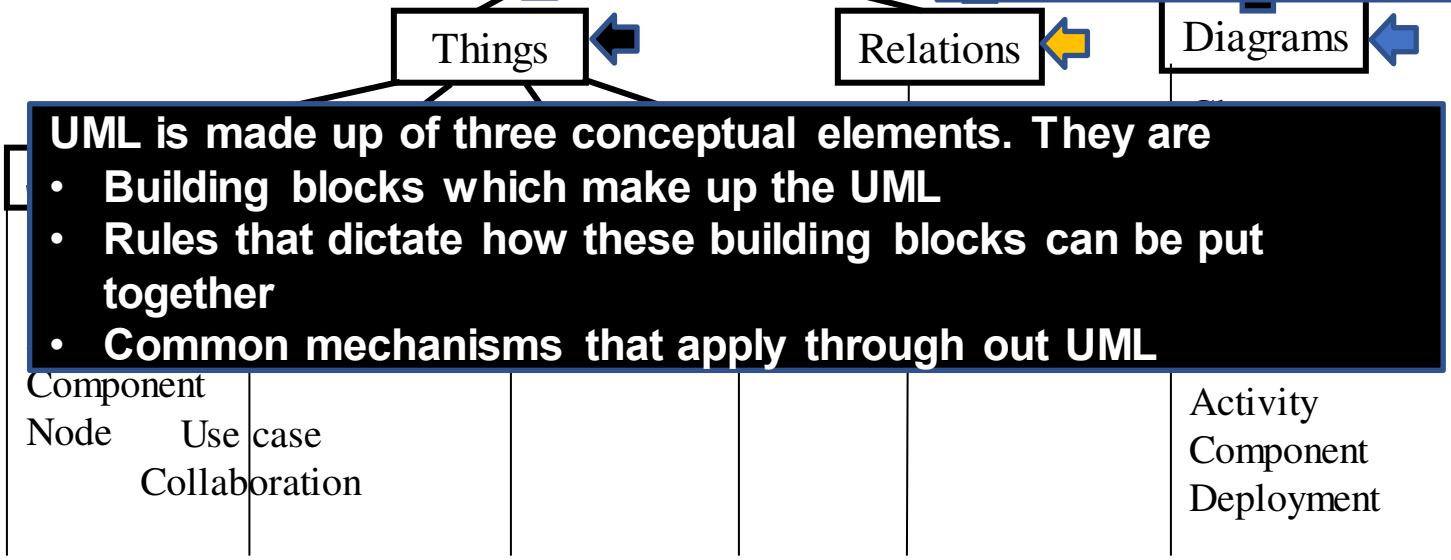
Conceptual model of UML

Things are abstractions in the model.

There are four kinds of things or abstraction

- Relationship ties together the things or abstractions
- There are four different kinds of relationships

Diagrams : Graphical representation of set of elements, often rendered as a set of connected graph of vertices (things) and arcs (relationships)
It's a projection into the system



UML is made up of three conceptual elements. They are

- Building blocks which make up the UML
- Rules that dictate how these building blocks can be put together
- Common mechanisms that apply through out UML

SOFTWARE ENGINEERING : ARCHITECTURE & DESIGN

UML Constructs or Diagrams which will be used for Object Oriented Design

State Diagram
State machine specifies the sequence of states an object or an interaction
g its lifetime in response to events and the responses to
s, transitions n
d activities wh
itions, events
internal object states

These repres
structural ele
are either cor
physical.

Component Diagram
Shows deployable components, including interfaces, ports and internal structure

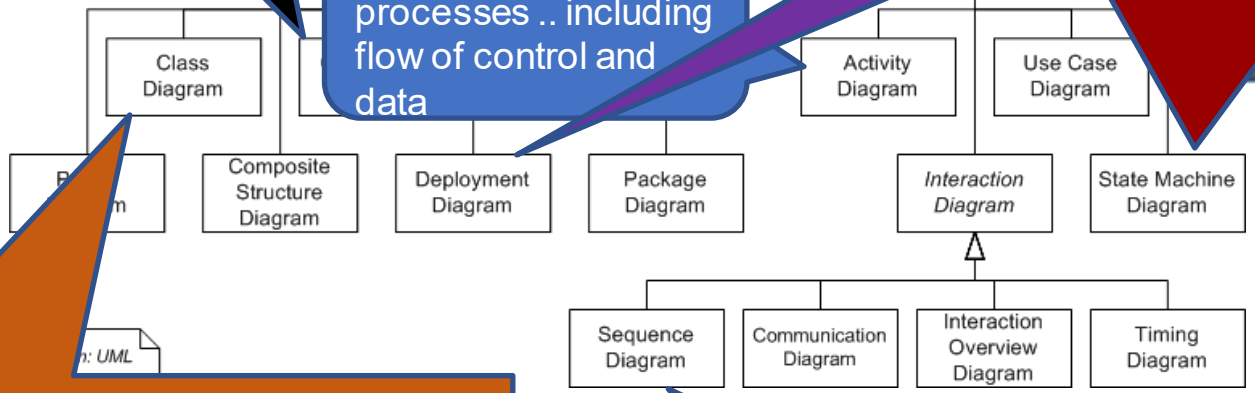
Deployment Diagram
Shows the configuration of run-time processing nodes and the components that are deployed on them

vents that

c view of

Its r d usin n.

Activity Diagram
This Shows processes .. including flow of control and data



Class Diagram
Describes a description of a set of objects that share the same attributes, operations, relationships and semantics.
Shows Classes, Interfaces, relationships

Sequence Diagram
Shows interactions between objects as a time ordered view

SOFTWARE ENGINEERING : ARCHITECTURE & DESIGN

Terms and Concepts

- System is a collection of subsystems organized to accomplish a purpose and is described by a set of models
- Subsystem is grouping of elements of which some constitute a specification of the behavior offered by the other contained elements
- Model is a schematically closed abstraction of the system (It represents a complete and ***self consistent simplification of reality*** created in order to better understand the system
- In Architecture, a ***view*** is a projection into the organization, the structure of a systems model focused on one aspect of the system
- A diagram is a graphical presentation of a set of elements most often rendered as a connected graph with vertices (things) and arcs (relationships)

Or

System represents the thing you are developing ,viewed from different perspectives by different models, with those views presented in the form of a diagram

SOFTWARE ENGINEERING : ARCHITECTURE & DESIGN

Generics of a Class Diagram : Object

Consider this glass as an object

Attributes of the object describes one or more of the characteristics of the object eg.

Summary

An object has

Attributes and values of attributes = state

Operations = behavior

An object is characterized by

Its state and behavior

- **Operation/Behavior**
  - It can be
    - Broken
    - Filled
    - Emptied
- **Attributes**
  - Its height is ..
  - It has/has not a leg
  - It is % full
- **Value/State**
  - Its height is 20 cm
  - It has a leg
  - It is half full

Generics of a Class Diagram : Object (Contd.)

- Object is a concept, abstraction or thing with identity that has a meaning for the application
- Objects appear as proper nouns or specific references in problem descriptions and discussion with users
- Objects can be concrete (person, store, car) or could be conceptual or abstraction (strategy, plan, layout)
- objects have a unique identity that remains unchanged throughout their lifetimes.
- Objects can play the same or different roles in respect of each other. Same role would have multiplicity
- A candidate key is a minimal set of attributes that uniquely identifies an object (or link)

Generics of a Class Diagram : Class

- A class describes a group of objects with the same properties (attributes), behaviors (operations), kinds of relationships and semantics

  Eg. Person – has a name, birthdate and may work in a job

  process – has an owner, priority, list and list of required resources

- Classes appear as common nouns and noun phrases in problem descriptions and discussions with users which needs to be modelled in the software solution
- An object would be an instance of the class. Objects in a class differ in terms of their attribute values and relationship to other objects
- Objects in a class share a common semantic purpose
- A class has responsibilities. A responsibility is something that instances of the class should be able to fulfill
- The purpose of class modelling is to describe objects
- **Class abstracts objects and hence helps in generalizing from a few specific cases to a host of similar cases**

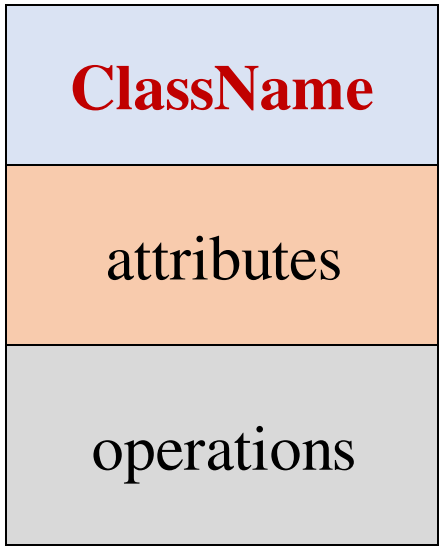
CLASS DIAGRAM

Generics of a Class Model

- A class model captures the static structure of a system by characterizing the objects in the system, the relationships between the objects and the attributes and operations for each class of objects
- ***Class diagram*** graphically represents a Class model which characterizes the objects in a system and provides an intuitive graphic representation of a system
- A class represents an identical set of objects in a system and there are multiple classes which would exist in a system
- Focus is on looking at the problem domain in terms of objects/Classes rather than functionality and Class modelling will need to identify the classes which are needed to be implemented in the system to meet the requirements identified
- A class diagram needs to represent the
  - Classes and the relationship between the classes (how do the classes relate to each other)
  - The attributes (fields) or named properties of these classes
  - The operations which the class will need to support (methods for the class)

CLASS DIAGRAM

Classes



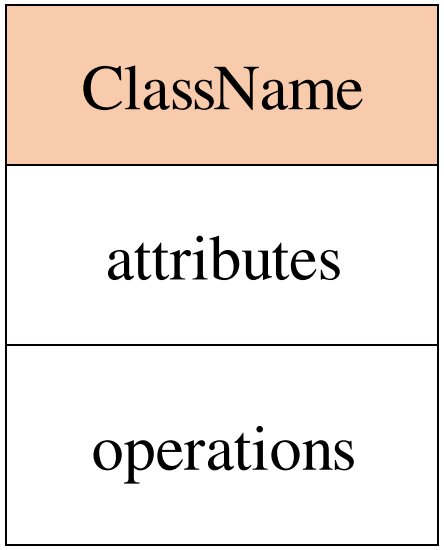
A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

An object would be an instance of the class

CLASS DIAGRAM

Class Names



The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment and is centered.

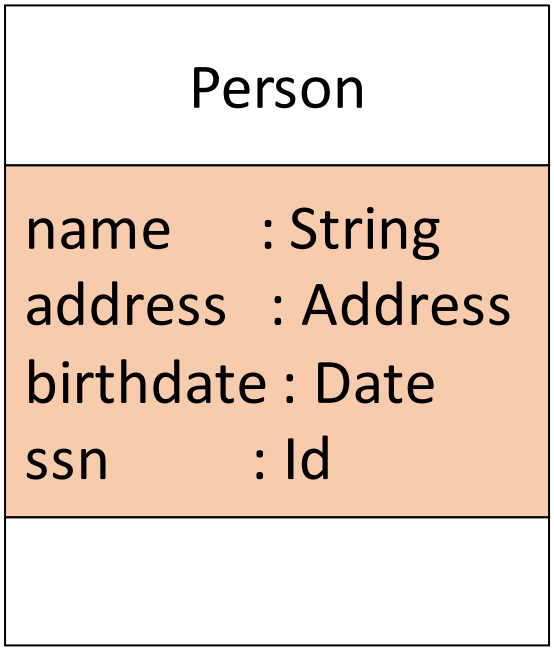
A textual string derived from vocabulary of the system to be modeled – typically a noun

Qualifier might represent inheritance: Savings Account, Current Account

Typically ClassNames are singular and starts with a capital letter and intervening words again starting with capital letters

CLASS DIAGRAM

Class Attributes



An *attribute* is a named property of a class that describes the values the object instance can hold

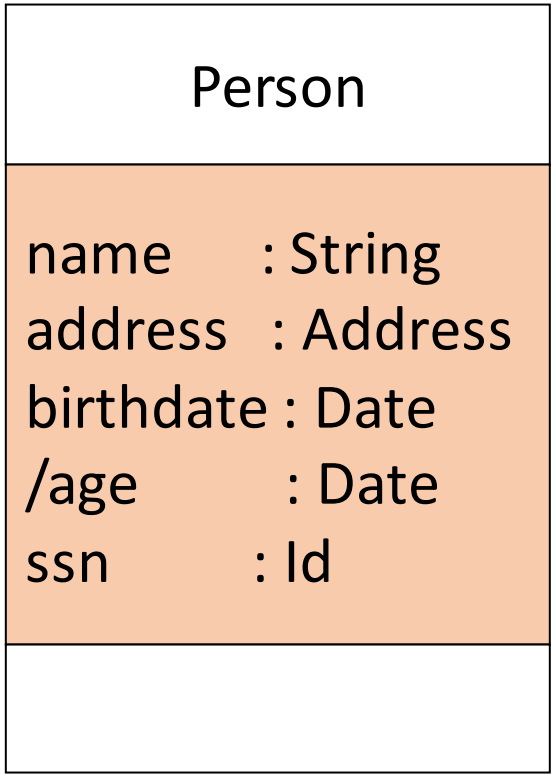
Property of all instances of the class:

Name, address

In the class diagram, attributes appear in the second compartment just below the name-compartment.

CLASS DIAGRAM

Class Attributes (Cont'd)



Attributes are usually listed in the form:

attributeName : dataType

or

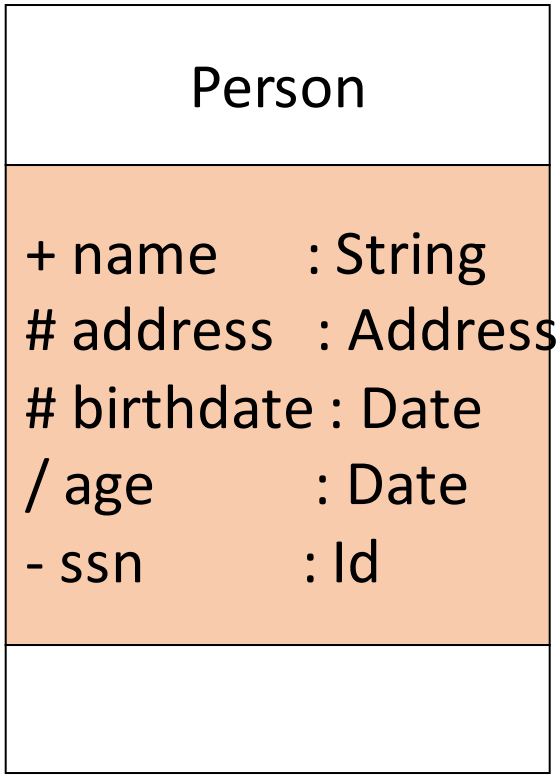
attributeName : dataType = default value

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age (or Order Total).

A derived attribute is designated by a preceding '/'

CLASS DIAGRAM

Class Attributes (Cont'd)



Attributes can have visibility be:

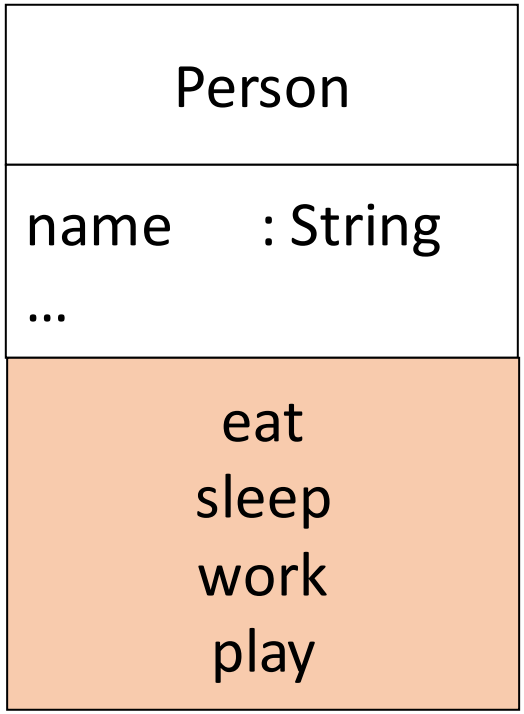
+ public
# protected
- private
/ derived

Some of the attributes which characterize Public, Private and Protected

- Levels of Accessibility
- Provision of encapsulation
- Provision of method overriding

CLASS DIAGRAM

Class Operations



Operations describe the class behavior and appear in the third compartment.

An operation is a function or a procedure that may be applied to or by objects in a class.

Each operation has a target object as an implicit argument

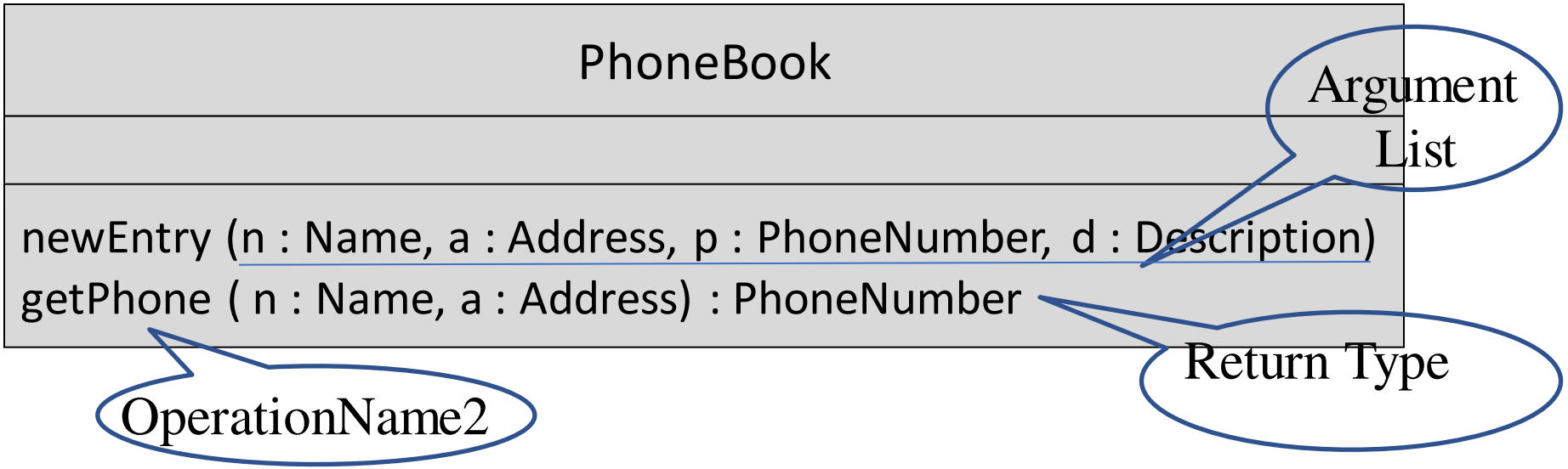
Method is an implementation of an operation on a class

It can also be looked at as an Implementation of a service, invoked by client

Invocation may change state of object

CLASS DIAGRAM

Class Operations (Cont'd)



You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.

There could be a direction for the operation argument which is optional

direction argumentName : type = defaultValue

direction could be in, out or inout (an input which can be modified)

Not all attributes or operations need to be shown. Use … to indicate.

CLASS DIAGRAM

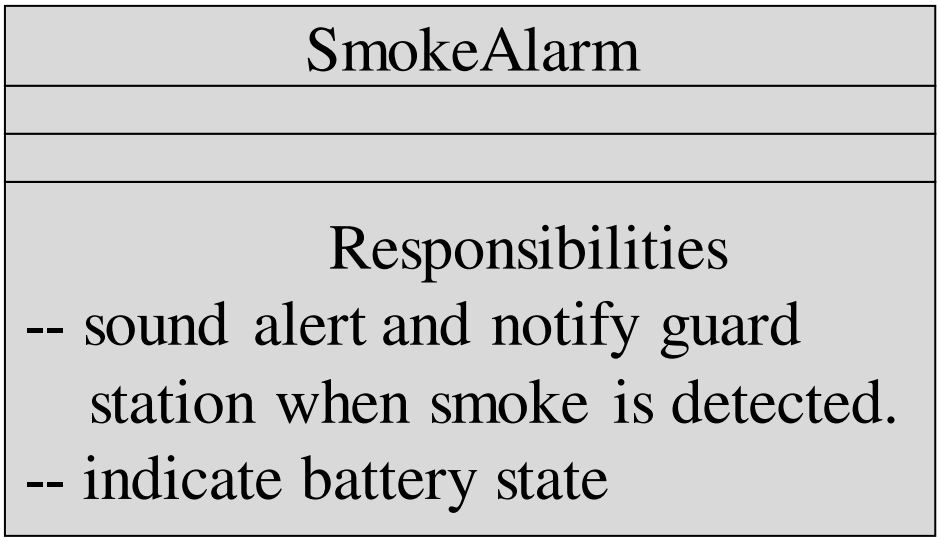
Class Responsibility

Responsibility :

A responsibility is a contract or obligation of a class to perform a particular service.

It's a free form text written as a phrase or a sentence

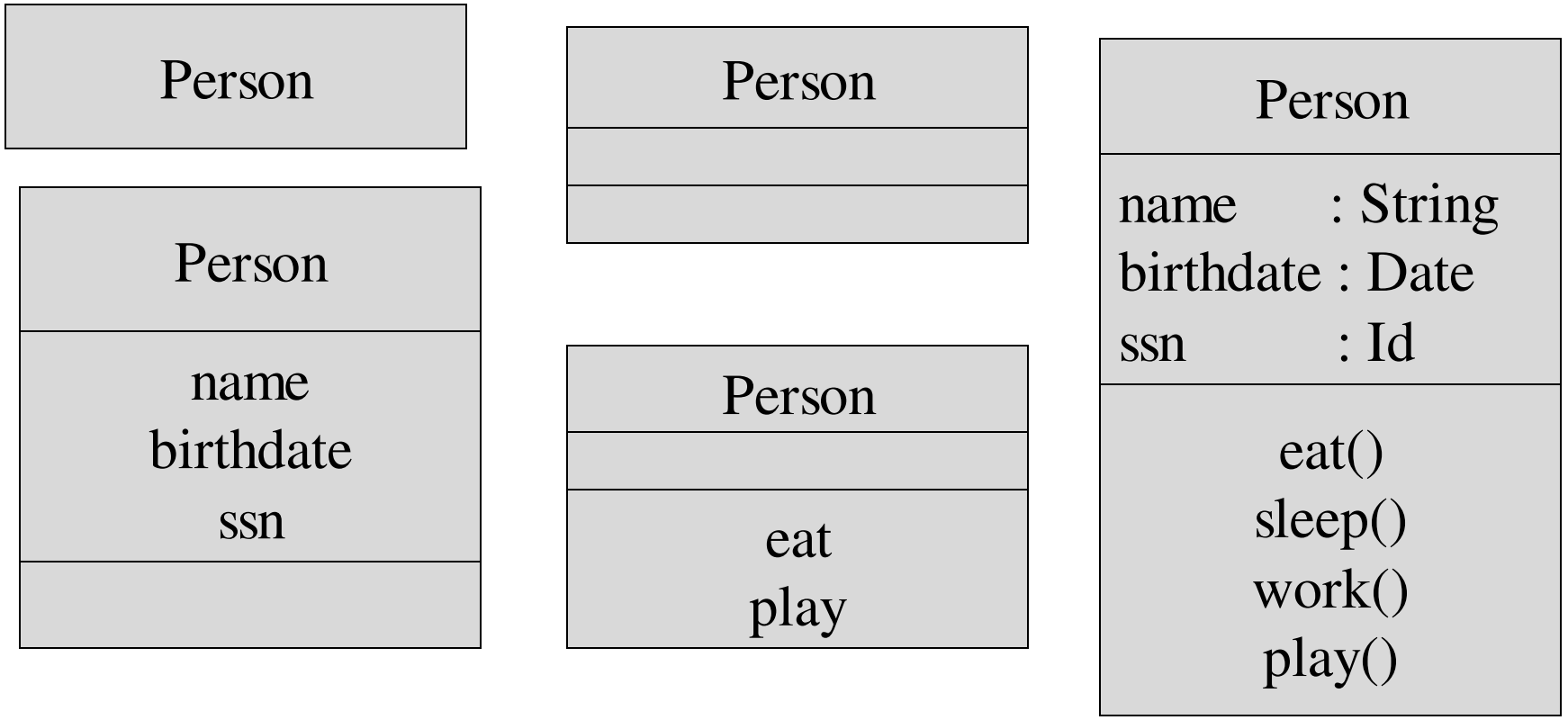
Eg. Determine risk of the customer order



CLASS DIAGRAM

Depicting Classes

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



CLASS DIAGRAM

Relating Classes with Use Cases using CRC

(Class-Responsibility-Collaborators)

CRC cards are tool used for brainstorming in OO design

CRC cards are created from Index cards and each member of the brain storming session with write up one CRC card for each relevant class/object of their design

A typical CRC card would look as shown

Objects need to interact with other objects (Collaborators) in order to fulfill their responsibilities

Since the cards are small, prevents to get into details and give too many responsibilities to a class

They can be easily placed on the table and rearranged to describe and evolve the design

Class Name	
Responsibilities	Collaborators

Class: Account	
Responsibilities	**Collaborators**
Know balance	
Deposit Funds	Transaction
Withdraw Funds	Transaction, Policy
Standing Instructions	Transaction, Standing Instruction, Policy, Account

CLASS DIAGRAM

Modelling the Class Collaborations

- Classes (from the vocabulary of the system) do not stand alone but works with other classes to carry out some semantics which is bigger than the individual classes
- Class diagrams need to depict the visualization, specification and construction of various collaborations with these classes
- Class diagram models one part of the things and hence each class diagram should focus on one collaborations at a time
- To model a collaboration
  - Identify the function or behavior of the part of the system that is represented by the group of classes
  - For each of these functions, identify the classes, interfaces and other collaborations
  - Walk through these using scenarios, which will help discover new classes and also identify semantically wrong ones
  - For each of these classes, identifying their contents and responsibilities help to visualize and balance the responsibilities and as a cascade attributes and operations
  - As part of the course, we would be looking at a scenario and discussing the same

CLASS DIAGRAM

OO Relationships

Relations tie together the different classes/objects/abstractions together

1. **Association**

   This is a structural relationship that describes a set of links which is a common connection among objects

   1. Aggregation
   2. Composition

2. **Generalization/Specialization**

   Its a relationship where objects of specialized elements are substitutable for the objects of the generalized elements. Child shares the structure and behavior of parents

Relations tie together the different classes/objects/abstractions together (cont.)

3. **Dependency**

   Semantic relationship between two things in which a change to one thing may effect the semantics of the other thing.

4. **Realization**

   - Its a relationship between classifiers.
   - One specifies and other carries out.
   - Typically found in interfaces and the classes which realize them.

CLASS DIAGRAM : OO Relationships

1. Association

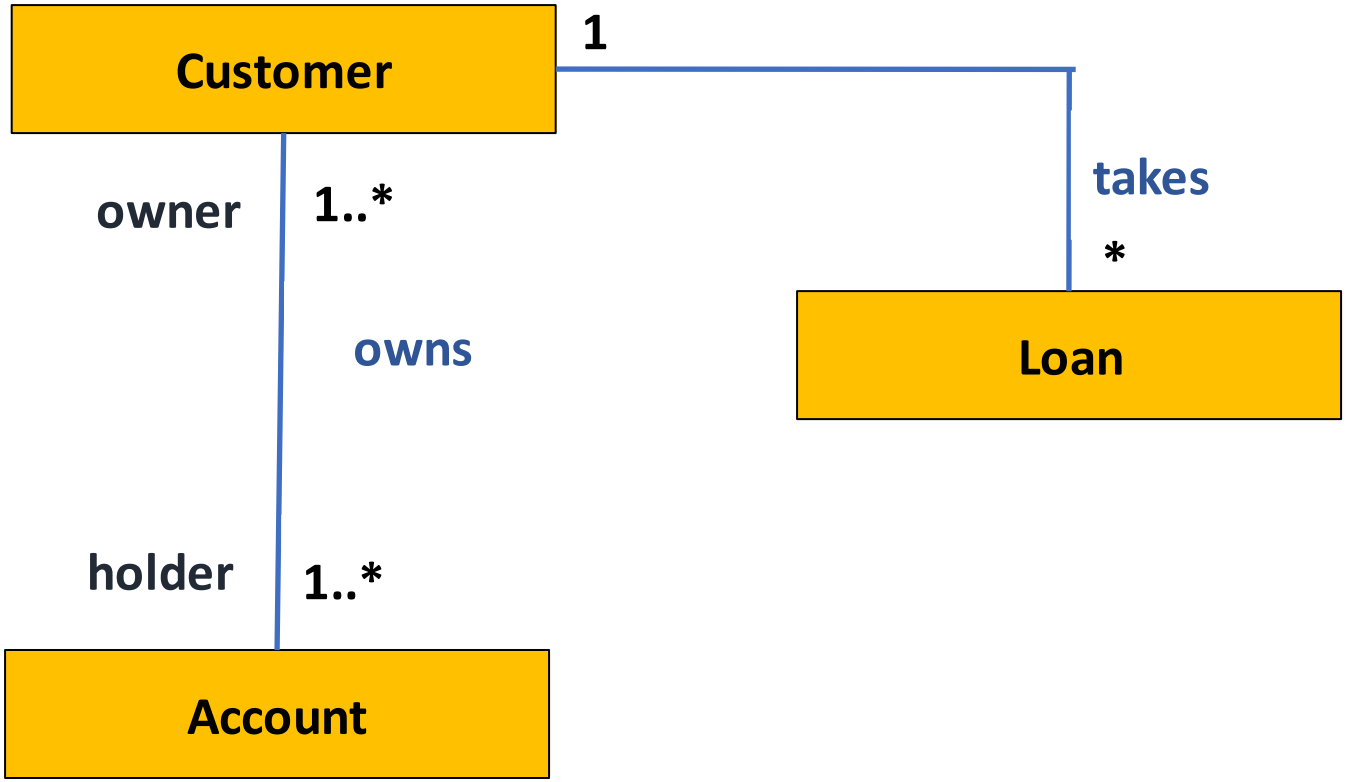
- An *association* describes a relationship between instances of one class and instances of the same or another class represented as a connection
  - Customer *owns* Account
  - Student *enrolls for* Elective
  - Courses have students
  - Courses have exams

  Each of the connections are called links. Typically implemented as pointers
- **Multiplicity**
  - Number of instances or objects of one class that can be associated with one instance of another
  - Constraints the number of related objects and depends on how you define the boundaries of the problem
  - UML diagram explicitly list multiplicity (constraint while cardinality is the count) at the end of the association lines

CLASS DIAGRAM : OO Relationships

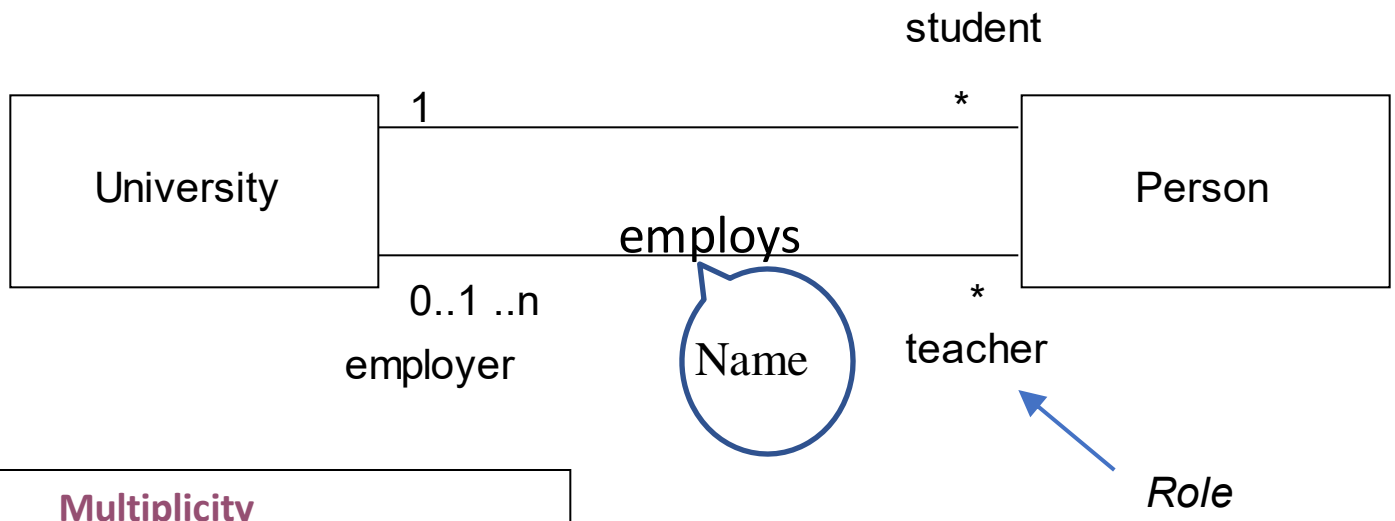
1. Association (Contd.)

- Description of an Association
  - Association has two ends
  - Association ends may be named based on the role or the purpose of the association e.g. Enrolled Student, Course Enrolled
  - Navigability (unidirectional, bidirectional links)



CLASS DIAGRAM : OO Relationships

1. Association : Name, Multiplicity and Roles



Multiplicity

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

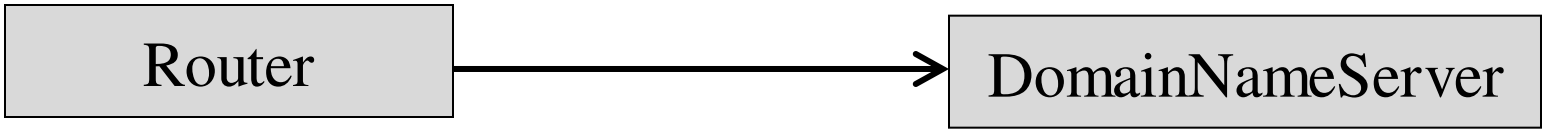
Role

"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

CLASS DIAGRAM : OO Relationships

1. Association Relationships (Contd.)

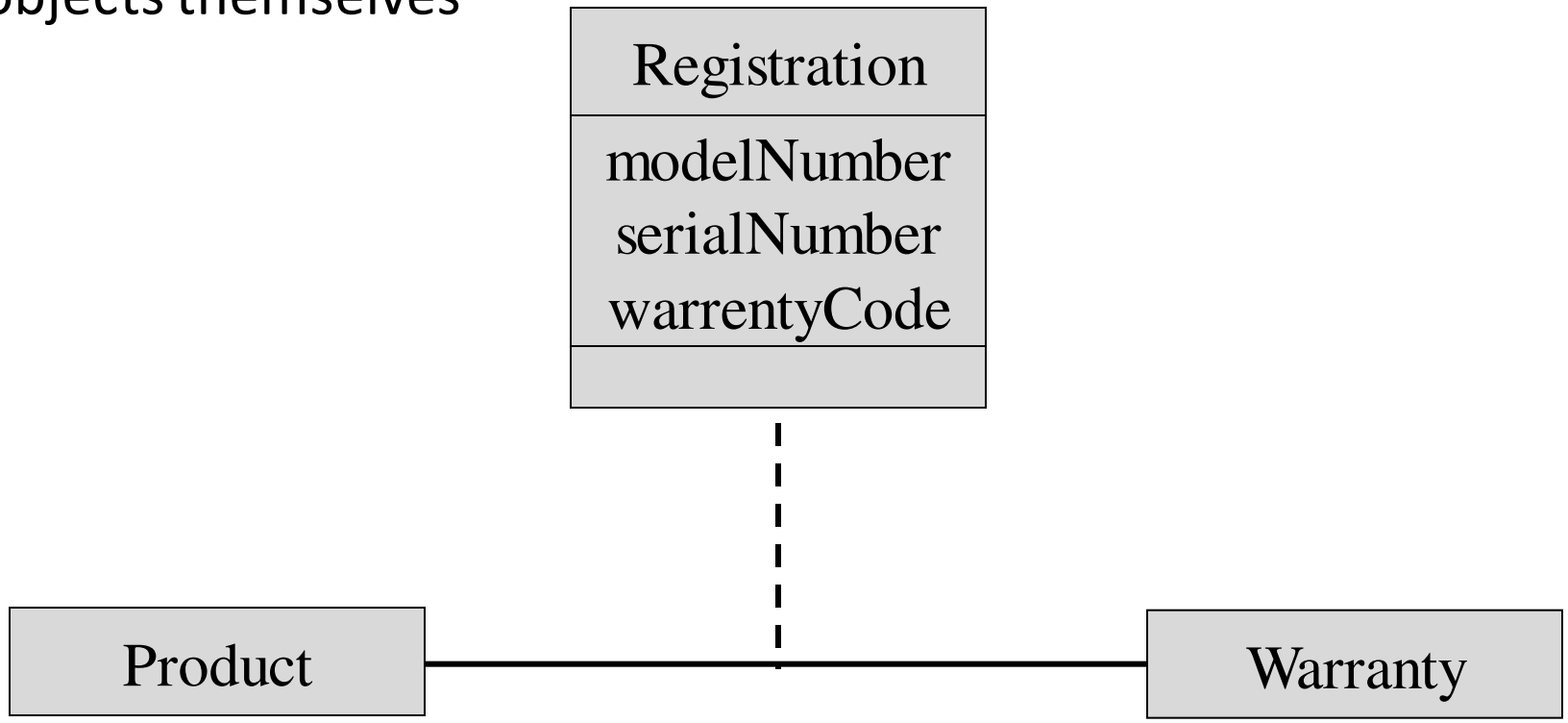
We can constrain the association relationship by defining the *navigability* of the association. Here, a *Router* object requests services from a *DNS* object by sending messages to (invoking the operations of) the server. The direction of the association indicates that the server has no knowledge of the *Router*.



CLASS DIAGRAM : OO Relationships

1. Association Relationships (Contd.)

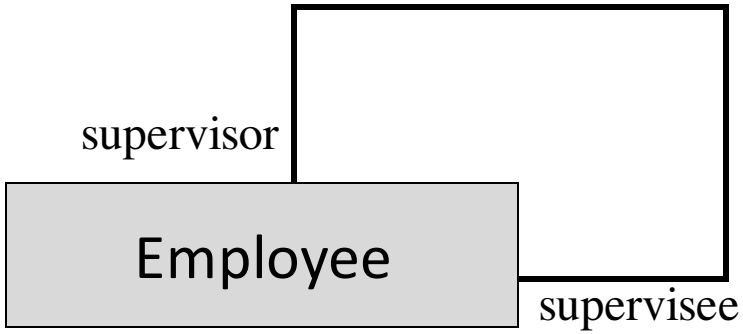
There can be situations where Associations may need a link classes or association classes ..like registration below .. can also be objects themselves



CLASS DIAGRAM : OO Relationships

1. Association Relationships (Contd.)

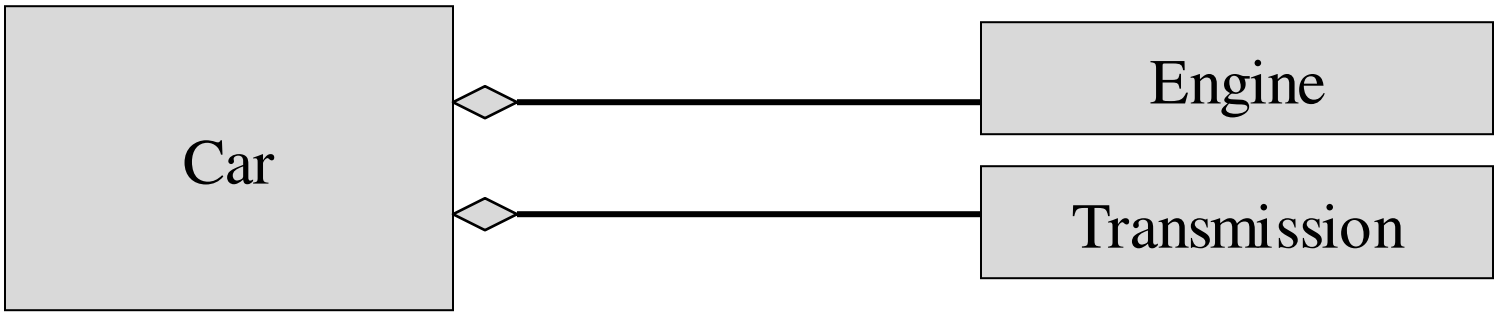
A class can have a *self association*.



CLASS DIAGRAM : OO Relationships

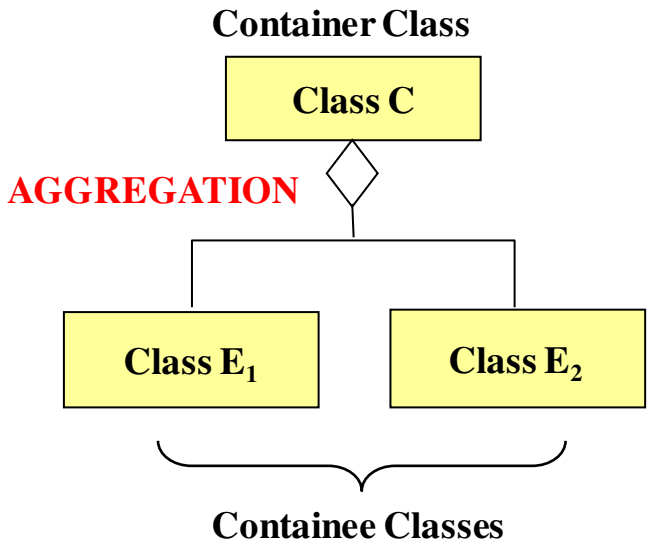
1. Association Relationships : Aggregation

- An aggregation is a strong form of association in which objects are assembled or configured together to create a more complex object.
- An aggregation describes a group of objects and how you interact with them. It defines a single point of control, called the aggregate that represents the assembly and decides on how the assembled objects respond to changes or instructions that might affect the collection.
- It shows the class that takes the role of the whole, is composed (has a) of other classes, which take the role of the parts
- An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment or the association.

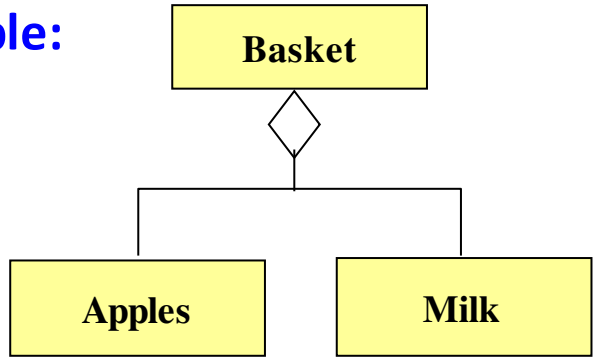


CLASS DIAGRAM : OO Relationships

1. Association Relationships : Aggregation



Example:



[From Dr.David A. Workman]

Aggregation:
expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

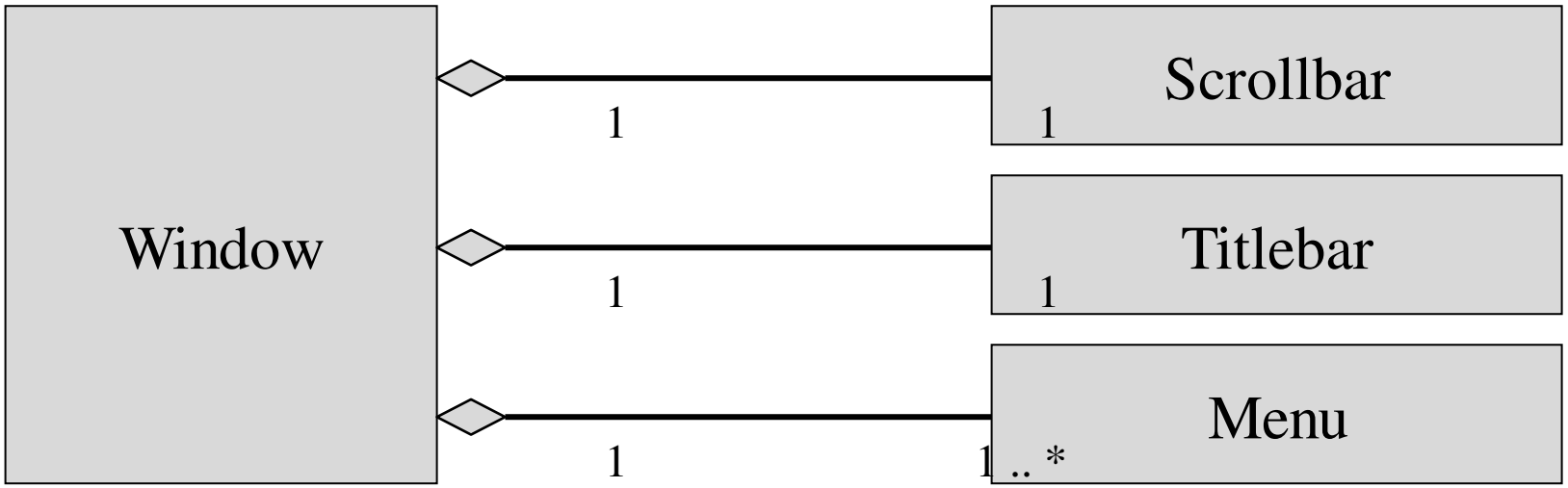
express a more informal relationship than composition expresses.

Aggregation is appropriate when Container and Containees have no special access privileges to each other.

CLASS DIAGRAM : OO Relationships

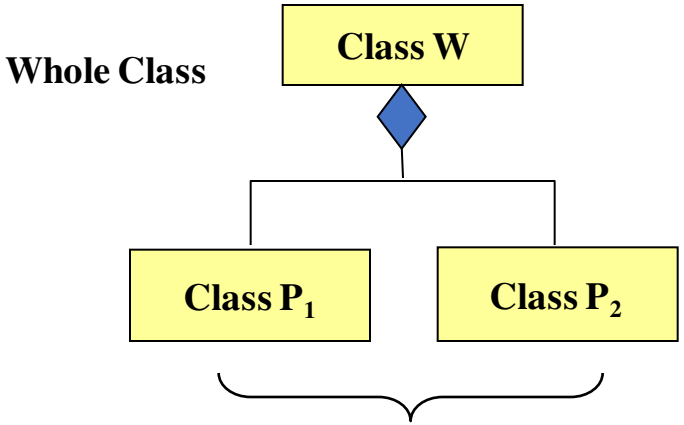
1. Association Relationships : Composition

A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



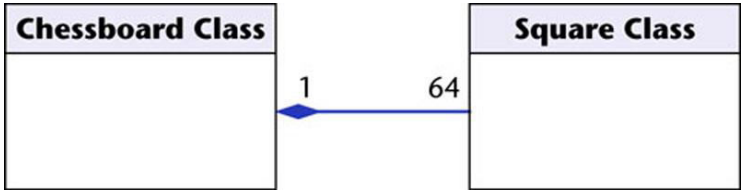
CLASS DIAGRAM : OO Relationships

1. Association Relationships : Composition



[From Dr.David A. Workman]

Example:



Association

Models the part–whole relationship

Composition

Also models the part–whole relationship but, in addition, Every part may belong to only one whole, and If the whole is deleted, so are the parts

Example:

A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.

CLASS DIAGRAM : OO Relationships

1. Association Relationships : Aggregation vs. Composition

- **Composition** is really a strong form of **association**
  - components have only one owner
  - components cannot exist independent of their owner
  - components live or die with their owner

  e.g. Each car has an engine that can not be shared with other cars.

- **Aggregations**
  - May form "part of" the association, but may not be essential to it. They may also exist independent of the aggregate.

  e.g. Apples may exist independent of the bag.

CLASS DIAGRAM : OO Relationships

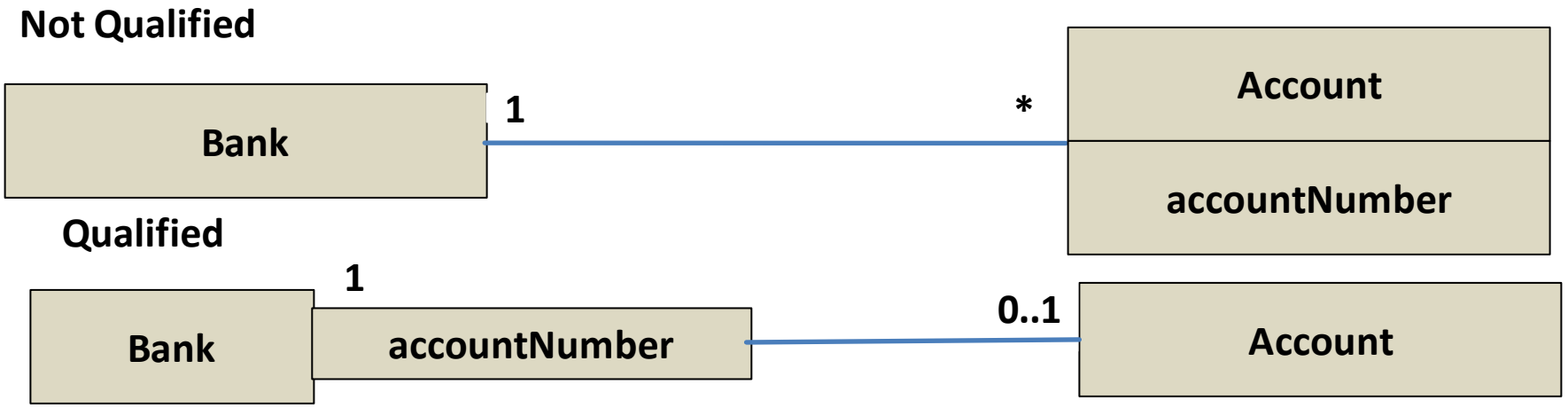
1. Association Relationships : Qualified Association

It is an association in which an attribute called 'qualifier', disambiguates the objects for a "many " association end

- Qualifier reduces the effective multiplicity from many to one ,

  Eg: for association with one to many multiplicity

  - Bank services multiple accounts
  - Account belongs to a single bank
  - Within context of a bank , account number specifies unique account

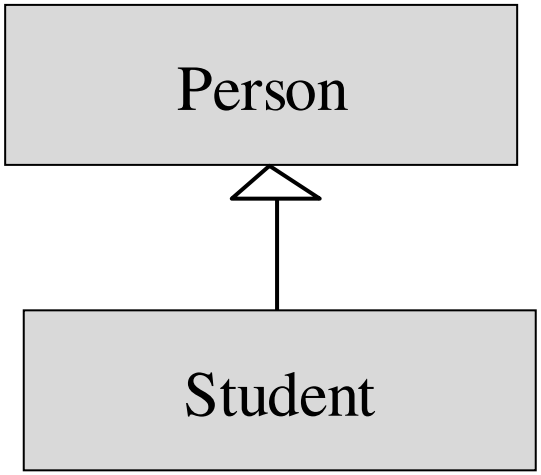


Attribute called ***qualifier*** disambiguates the objects for a "many" association end

CLASS DIAGRAM : OO Relationships

2. Generalization Relationships

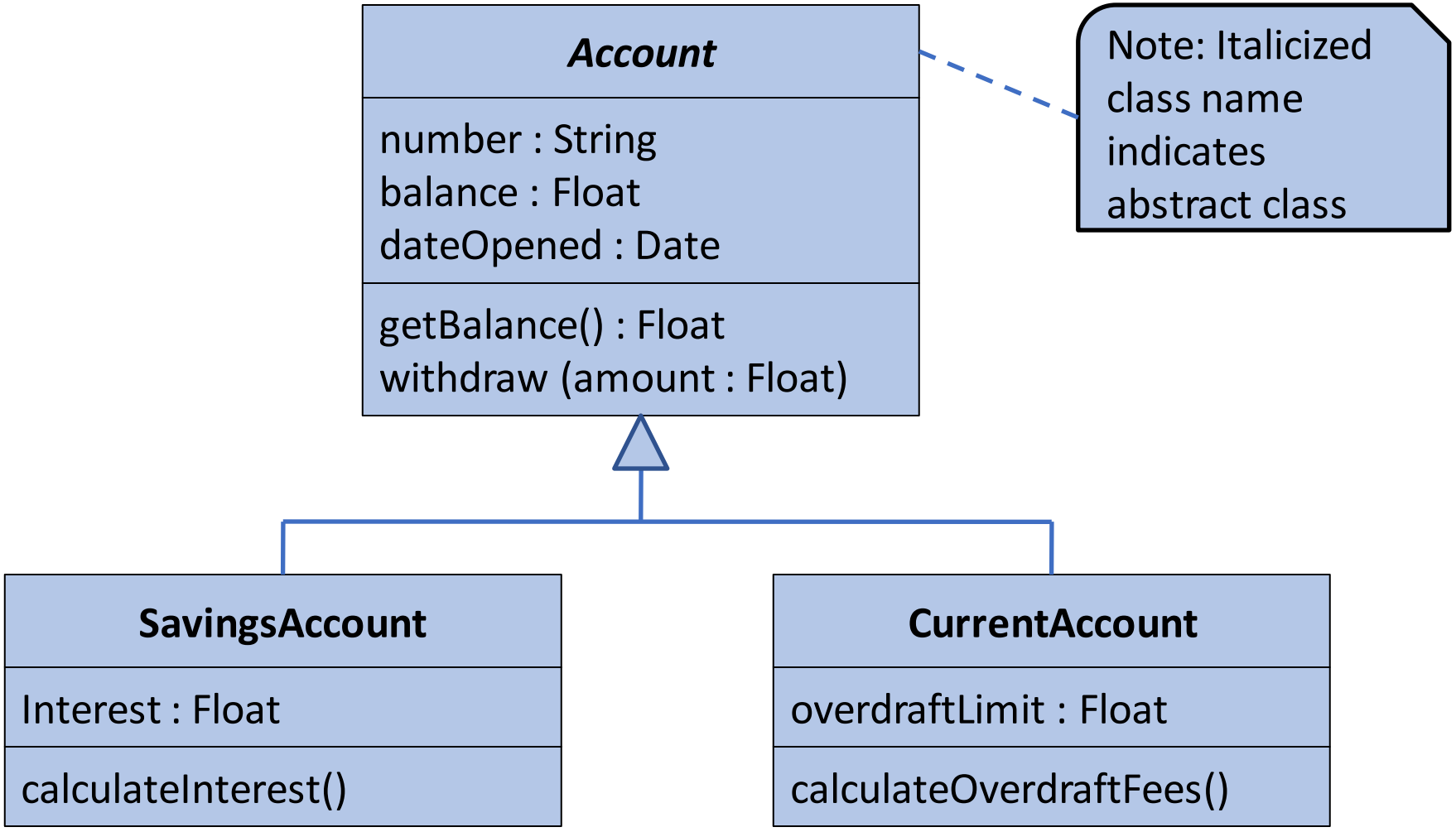
- Generalization (conceptual) in UML represents Inheritance (which is more of a language mechanism) in which a class "gains" all of the attributes and operations of the class it inherits from, and can override/modify some of them, as well as add more attributes and operations of its own
- In UML, a Generalization association between two classes puts them in a hierarchy representing the concept of inheritance of a derived class from a base class
- Inheritance relationship between subclass and superclass is "Is-a" relationship



- Subclass inherits features from superclass
  - Attributes and Operations (unless protected)
  - Operations may be overridden by subclass
  - Is a specialization of the more general superclass.
- Uses of generalization
  - Polymorphism, Abstraction of common elements, Reuse of code

CLASS DIAGRAM : OO Relationships

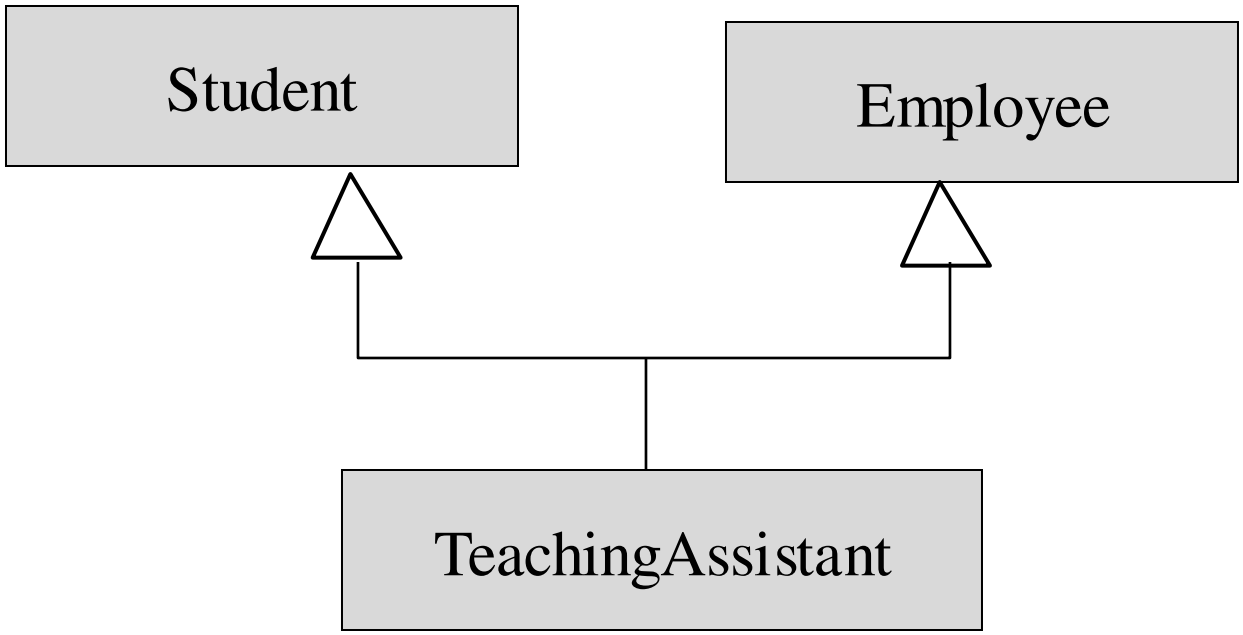
2. Generalization



CLASS DIAGRAM : OO Relationships

2. Generalization (Contd.)

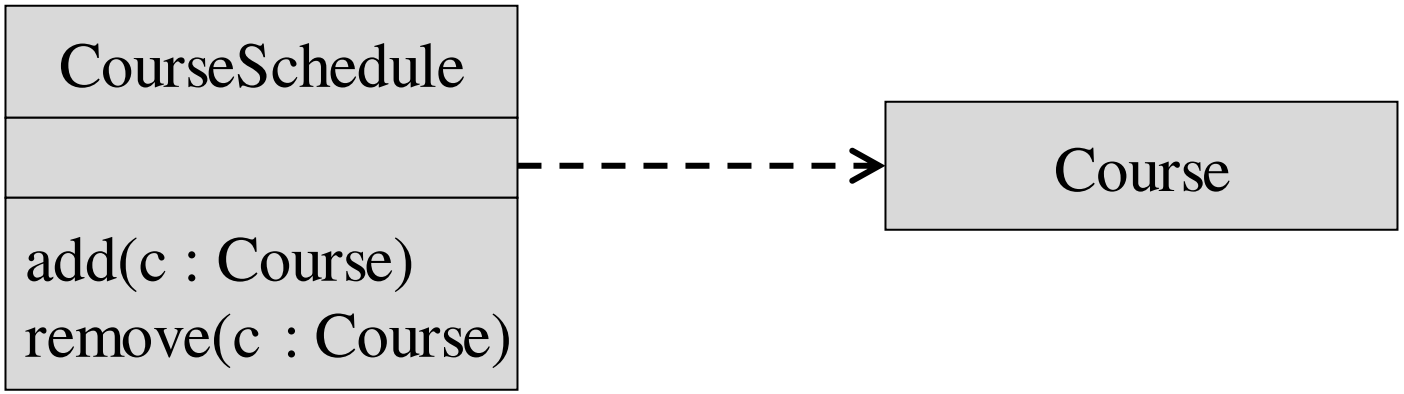
UML permits a class to inherit from multiple superclasses, although some programming languages (*e.g.*, Java) do not permit multiple inheritance.



CLASS DIAGRAM : OO Relationships

3. Dependency

- A dependency relationship is a semantic relationship in which one element, the client, uses or depends on one or more other elements, the supplier(s)
- Dependency relationship is used to represent precedence, where one model element must precede another.
- The dependency from CourseSchedule to Course exists because Course is used in both the add and remove operations of CourseSchedule.
- Dependency is displayed as a dashed line with an open arrow that points from the client to the supplier.



THANK YOU

Dr. H. L. Phalachandra
Department of Computer Science and Engineering
phalachandra@pes.edu

OOAD-SE

CLASS DIAGRAM

Dr. H.L. Phalachandra

Leveraging some content from

Prof. Vinay Joshi

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the

OOAD-SE

Class Modelling to Solve a Problem

Dr. H.L. Phalachandra

Leveraging some content from

Prof. Vinay Joshi

Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the

Modelling to Solve a Problem

Class Models Recap :

- We discussed on the context of using UML and designing a system using Object Oriented approach
- We discussed on the different models which we would consider as part of the same.
  - Static models - Class Model, Component Model and Deployment Model
  - Behavioral models - Activity Model, Sequence Model and the State Model
- We discussed terminologies regarding system, models, views and diagrams
- We then discussed on the objects in the problem space its characteristics, class and its characteristics, Class diagrams and relationships between classes as Associations whether Aggregation or composition, Generalization, Dependency and Realization
- We used some examples to understand these concepts

Modelling to Solve a Problem

Consider an ATM Case Study

- Design the software to support a computerized banking network including both human cashiers and automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data.
- Automatic teller machines communicate with a central computer that clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires appropriate recordkeeping and security provisions. The system must handle concurrent accesses to the same account correctly.
- The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.

Modelling to Solve a Problem

Building this Class model for an application would have activities as

1. Use the problem description/use cases and find the Classes
2. Prepare a data dictionary
3. Find associations
4. Find attributes of objects and links
5. Organize and simplify classes using inheritance
6. Verify that access paths exist for likely queries
7. Iterate and refine the model
8. Reconsider the level of abstraction
9. Group classes into packages

Modelling to Solve a Problem

1. Classes : Finding Classes

- List candidate classes found in written description of the problem
- Guidelines
  - Classes need to be extracted from problem domain
  - Some may be concreter and others may be abstract
  - Some may be explicitly stated by users/requestors/existing documents and others may be implicit
  - General rule start looking for Nouns for identifying classes

Modelling to Solve a Problem

1. Classes : Finding Classes : Identified Classes for the ATM Example

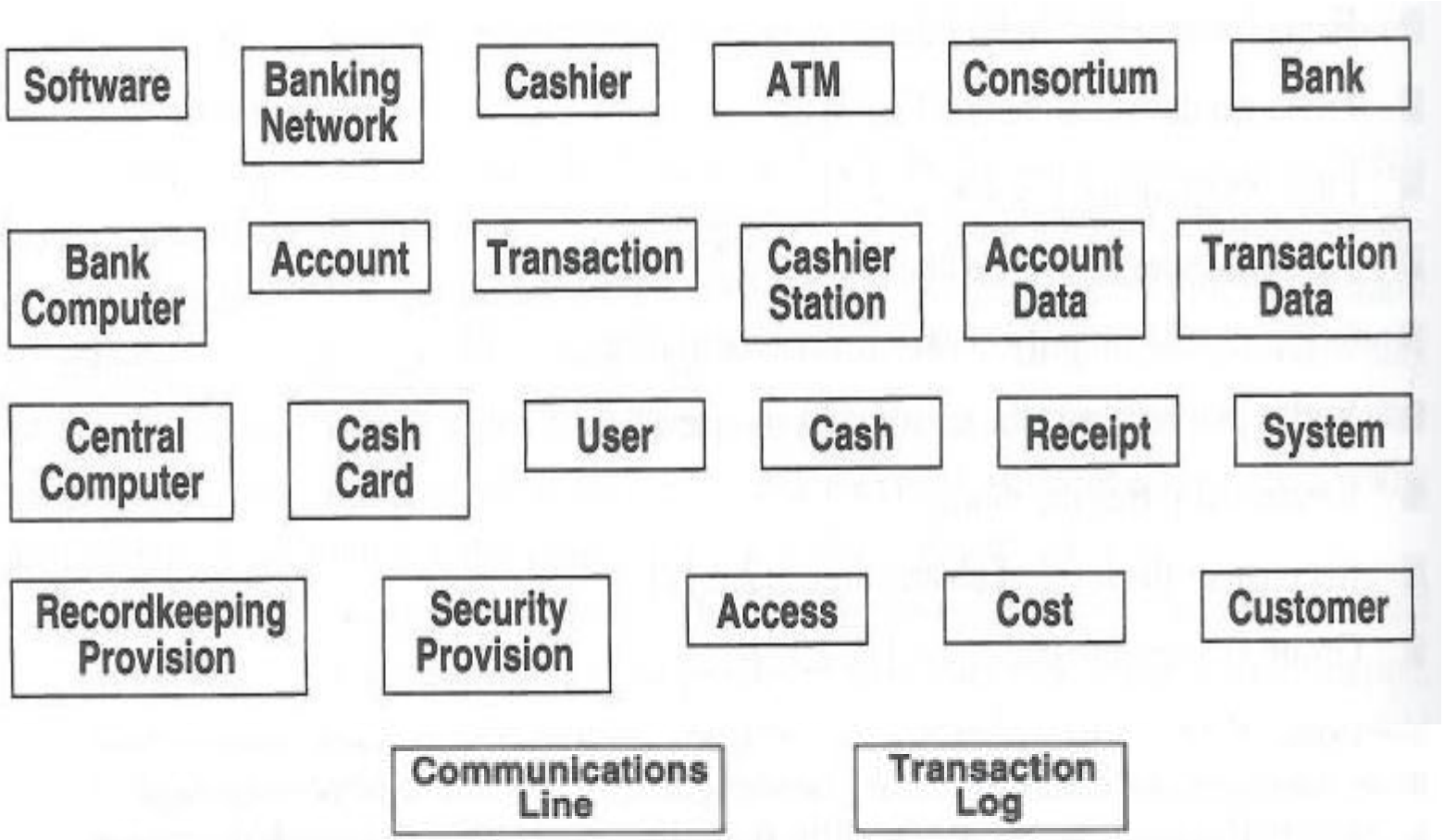


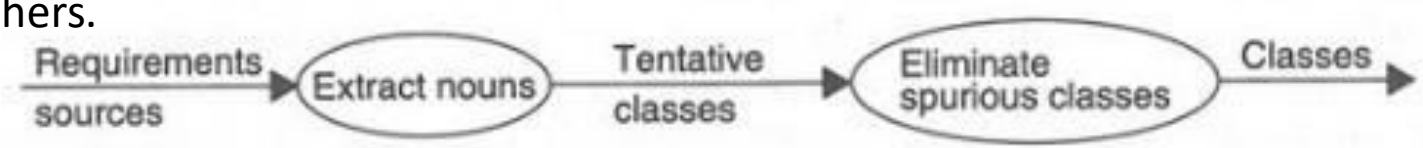
Figure 12.3 ATM classes extracted from problem statement nouns

Modelling to Solve a Problem

1. Classes : Keeping the right Classes

Identify and Eliminate classes that are

- Descriptive names which can give an indication Eg. ~~Security Provision~~
- Redundant Eg. For ATM example .. Customer ~~and User~~
- Irrelevant Eg. ~~Cost~~ say of an ATM machines (outside Scope)
- Vague Eg. Transaction record inlieu of ~~Record keeping provision~~
- Attributes – these are things identified with nouns and are attributes of classes but not classes. Eg. ~~AccountData~~ is an attribute of Account
- Operations applied to objects not manipulated on its own right. Eg. Call .. Date, time, origin destination
- Role - The name should indicate an intrinsic nature and not a role that is played. Consider subtypes of say a person .. Could be ~~employee~~, ~~boss~~, ~~spouse~~ (a person being a person and not a role)
- Eliminate Implementation constructs - Avoid things like data structure, trees. Arrays, process, algorithm, interrupts etc. Eg. ~~Communication lines~~ and ~~Transaction log~~
- Derived Classes - As a generic rule we can avoid those classes which can be derived from others.



Modelling to Solve a Problem

1. Classes : Keeping the Right Classes for the ATM system

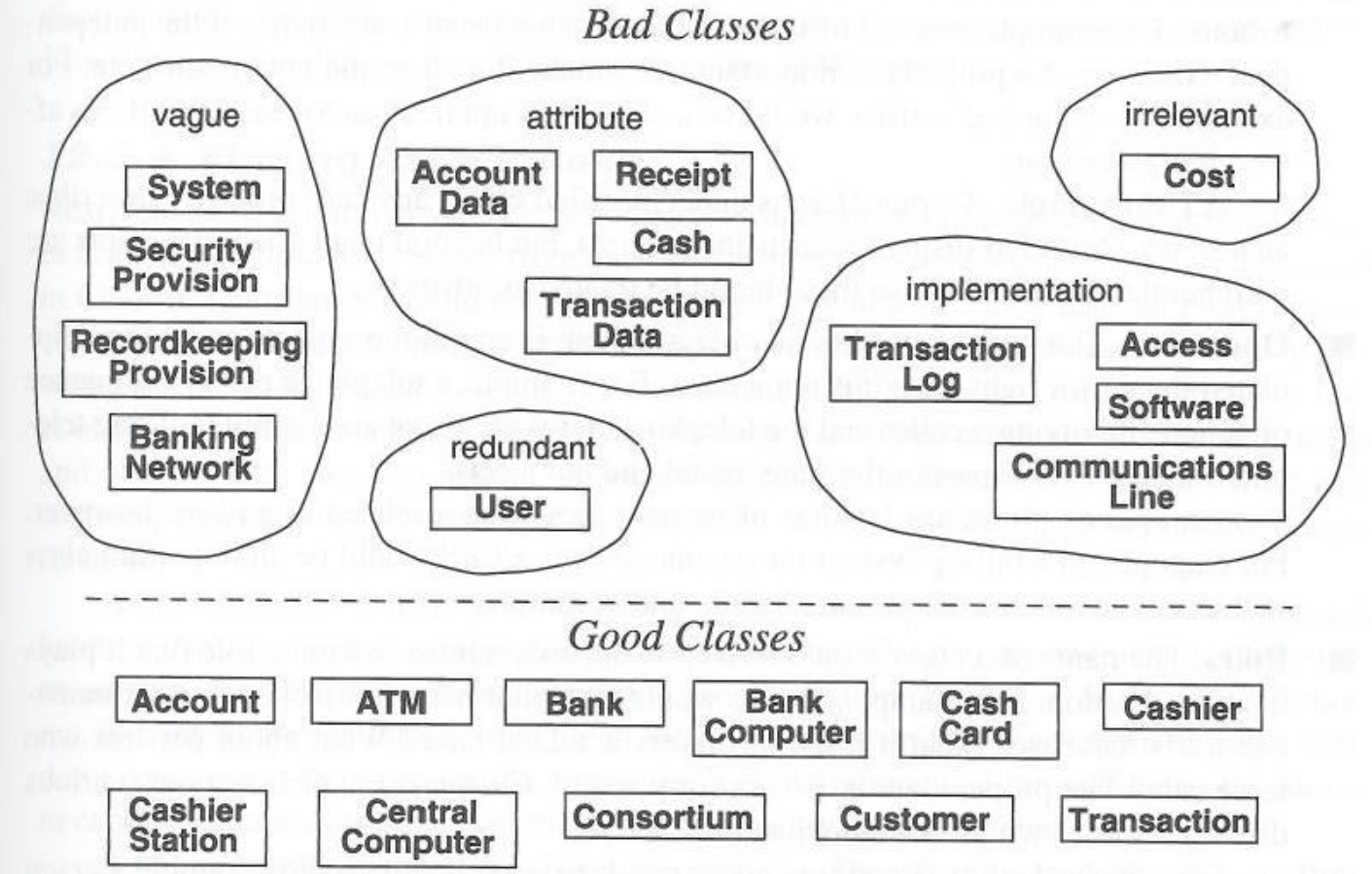


Figure 12.5 Eliminating unnecessary classes from ATM problem

Modelling to Solve a Problem

1. Classes : Identified good Classes for the ATM system

Account	ATM	Bank	Bank Computer
Cash Card	Cashier	Cashier Station	Central Computer
Consortium	Customer	Transaction	

2. Data Dictionary : Preparing a Data Dictionary

- Given that independent words used for classes and can be interpreted differently and can have different meanings, a dictionary is prepared with a description as will be interpreted in the example context.
- From the O-O perspective it should be possible to write a concise description of all valid classes
- This will need to describe the scope of the class within the current problem including assumptions or restrictions on its use
- This also shows the associations, attributes, operations and enumeration values
- The relational requirement is more concrete specially to make a list of tables and all fields of those tables. Thus this will identify the meanings of the fields and what domains they are on.

Modelling to Solve a Problem

2. Data Dictionary : Data Dictionary (Example)

Account—a single account at a bank against which transactions can be applied. Accounts may be of various types, such as checking or savings. A customer can hold more than one account.

ATM—a station that allows customers to enter their own transactions using cash cards as identification. The ATM interacts with the customer to gather transaction information, sends the transaction information to the central computer for validation and processing, and dispenses cash to the user. We assume that an ATM need not operate independently of the network.

Bank—a financial institution that holds accounts for customers and issues cash cards authorizing access to accounts over the ATM network.

BankComputer—the computer owned by a bank that interfaces with the ATM network and the bank's own cashier stations. A bank may have its own internal computers to process accounts, but we are concerned only with the one that talks to the ATM network.

CashCard—a card assigned to a bank customer that authorizes access of accounts using an ATM machine. Each card contains a bank code and a card number. The bank code uniquely identifies the bank within the consortium. The card number determines the accounts that the card can access. A card does not necessarily access all of a customer's accounts. Each cash card is owned by a single customer, but multiple copies of it may exist, so the possibility of simultaneous use of the same card from different machines must be considered.

Cashier—an employee of a bank who is authorized to enter transactions into cashier stations and accept and dispense cash and checks to customers. Transactions, cash, and checks handled by each cashier must be logged and properly accounted for.

CashierStation—a station on which cashiers enter transactions for customers. Cashiers dispense and accept cash and checks; the station prints receipts. The cashier station communicates with the bank computer to validate and process the transactions.

CentralComputer—a computer operated by the consortium that dispatches transactions between the ATMs and the bank computers. The central computer validates bank codes but does not process transactions directly.

Consortium—an organization of banks that commissions and operates the ATM network. The network handles transactions only for banks in the consortium.

Customer—the holder of one or more accounts in a bank. A customer can consist of one or more persons or corporations; the correspondence is not relevant to this problem. The same person holding an account at a different bank is considered a different customer.

Transaction—a single integral request for operations on the accounts of a single customer. We specified only that ATMs must dispense cash, but we should not preclude the possibility of printing checks or accepting cash or checks. We may also want to provide the flexibility to operate on accounts of different customers, although it is not required yet.

Data dictionary for ATM classes. Prepare a data dictionary for all modeling elements.

Modelling to Solve a Problem

3. Class Associations : Finding the Associations

- Association should describe structural property of application domain and not a transient event
- Associations in a problem domain may be identified by verbal phrases e.g. NextTo, part of, ContainedIn or Drives, TalkTo, Has, WorksFor, Manages, includes, shares, provides etc.
- Its to find a specific “has-a” relationship rather than “is-a”
- A reference from one class to another is an association

  Eg. class Person should not have an attribute employer .. There needs to be a relationship between class Person and class Company with association “worksfor”
- Find and extract from the problem statement the explicit and implicit verb phrases and list them. Use the Knowledge of problem domain to help.

Modelling to Solve a Problem

3. Class Associations : ATM System:

Verbal Phrases from ATM System domain for identification of Associations

Verb phrases

Banking network includes cashier stations and ATMs
Consortium shares ATMs
Bank provides bank computer
Bank computer maintains accounts
Bank computer processes transaction against account
Bank owns cashier station
Cashier station communicates with bank computer
Cashier enters transaction for account
ATMs communicate with central computer about transaction
Central computer clears transaction with bank
ATM accepts cash card
ATM interacts with user
ATM dispenses cash
ATM prints receipts
System handles concurrent access
Banks provide software
Cost apportioned to banks

Implicit verb phrases

Consortium consists of banks
Bank holds account
Consortium owns central computer
System provides recordkeeping
System provides security
Customers have cash cards

Knowledge of problem domain

Cash card accesses accounts
Bank employs cashiers

Modelling to Solve a Problem

3. Class Associations : Keeping the Right Associations

1. Eliminate associations between eliminated classes
   - Associations with eliminated classes has to be eliminated
     eg. Relationship with Banking Network, ATM interacts with User
2. Eliminating Irrelevant or implementation associations
   - Implementation concerns to be eliminated
     eg. System handles concurrent access
3. Actions which handle transient processing events
   eg. ATM accepts cash Card. ATM interacts with User etc.
4. Ternary Associations
   - As much as possible, decompose associations among 3 or more classes into binary associations (could be as qualified association)
     eg. Bank computer processes transactions against accounts to
     Bank computer processes transactions and Transactions concerns account
5. Derived Associations
   - Omit associations that can be defined in terms of other associations
     eg. Consortium shares ATMs can be Consortium owns central computer and Central computer communicates with ATMs

Modelling to Solve a Problem

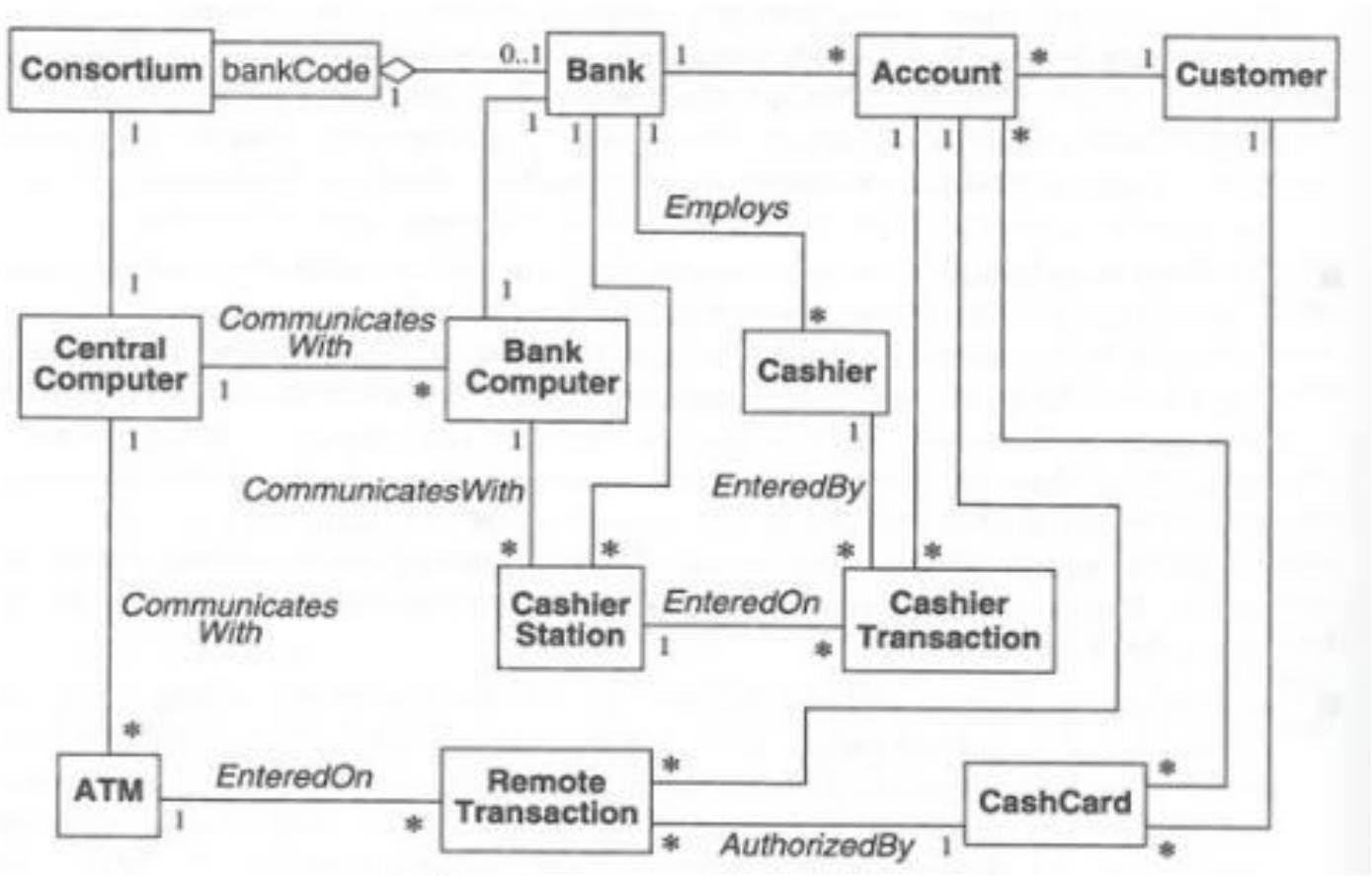
3. Class Associations : Pruning the Associations based on the Semantics

1. Misnamed associations
   - You eliminate them if they are redundant.
   - You rename them if they're valid and not redundant.
   - Names should state what a relationship is, not be based on how it came about or some other extraneous description.

     Eg. "Bank computer maintains accounts" describes an action can be renamed to "Bank holds accounts"
2. Association end names
   - Consider end names and it can be eliminated in case of it being obvious
     Eg. ATMs communicate with central computer .. the end names are obvious
3. Qualified associations
   - Qualifying an association ensures names which can be repeated can be identified in that specific context
     Eg. The qualifier bankCode distinguishes the different banks in a consortium.
4. Multiplicity
   - Ensure that cardinality of 1 is challenged and relooked

Modelling to Solve a Problem

3. Class Associations : Initial class diagram for ATM System



Initial class diagram for ATM system

Modelling to Solve a Problem

4. Class Attributes : Finding Attributes

- Attributes are data properties of individual objects
- Attributes can be identified as nouns followed by possessive phrases eg. color of the car or position of the cursor.
  If we can say the noun as car, the color of car.
- Attributes may not be completely described in the problem description and will need to be drawn from the knowledge of the domain and the application.
- Attributes can be found from similar application or systems
- Those which are not needed to be independently existing should be an attribute eg. birth date, weight
- Omit derived attributes (e.g. age)
- And others like Qualifiers, Identifiers, Association Attributes
  Eg. ATM example: AccountData is likely an attribute of an account

Modelling to Solve a Problem

A Class Model with attributes

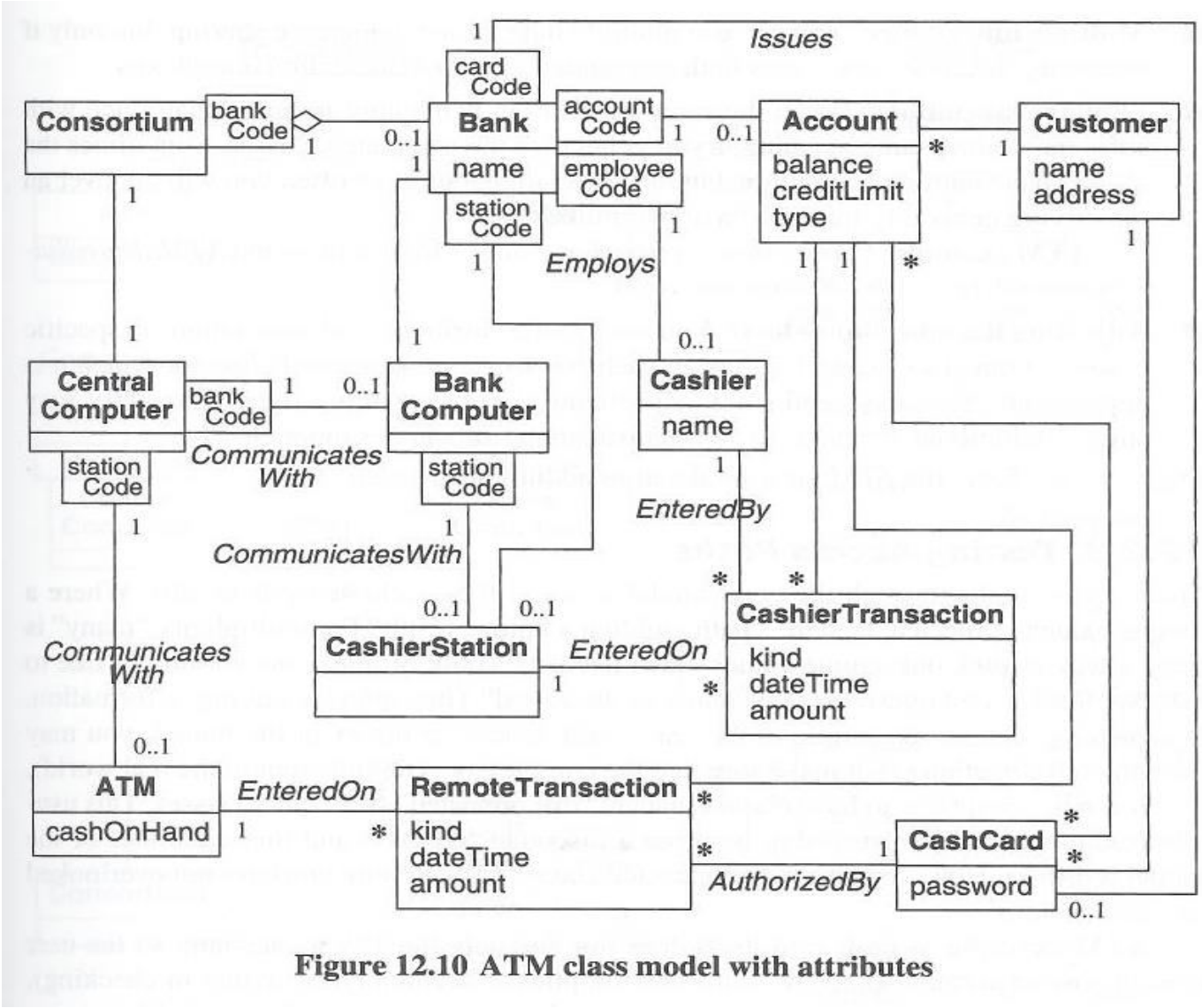


Figure 12.10 ATM class model with attributes

Modelling to Solve a Problem

Refining Class diagram with Inheritance

The class diagram can be further organized to share common structures through inheritance as generalizing common aspects of existing classes or specializing existing classes into multiple subclasses. These are typically as

- Bottom-up generalization
- Top-down specialization

Eg. Remote transaction and Cashier transaction are similar which can be generalized into Transaction

Modelling to Solve a Problem

Refined Class Diagram

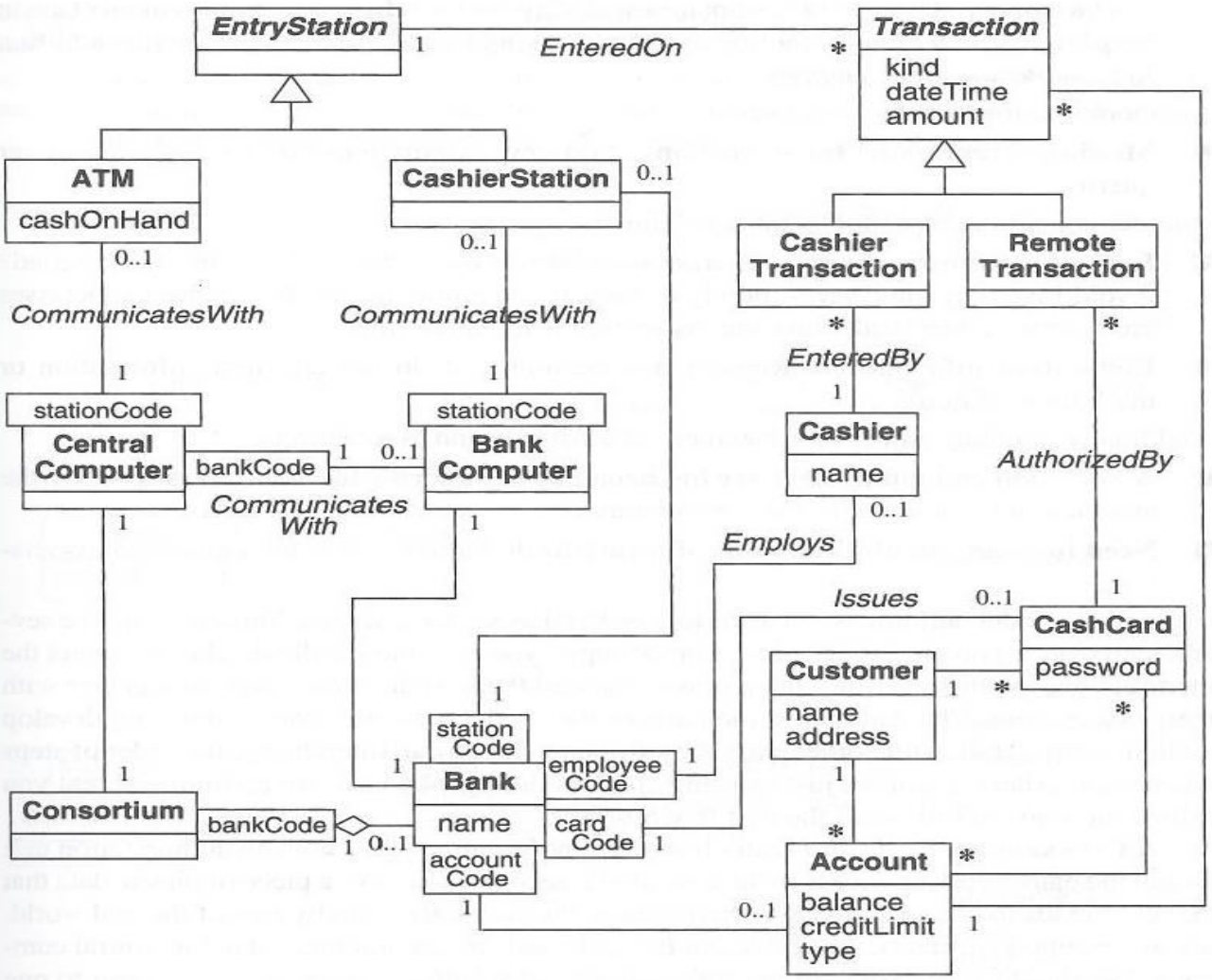


Figure 12.11 ATM class model with attributes and inheritance

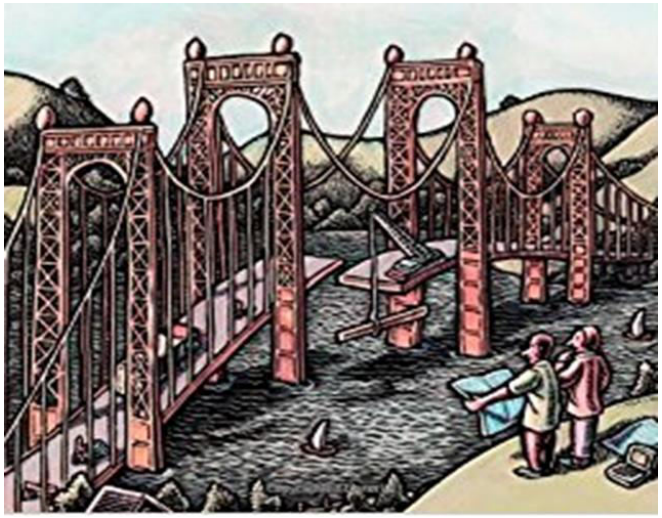
THANK YOU

Dr. H. L. Phalachandra
Department of Computer Science and Engineering
phalachandra@pes.edu

OOAD and SE

Implementation Models

(Component Model)



Dr. H.L. Phalachandra
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Static Models for Solving a Problem

Component Diagram

- We discussed on the class diagram and details on how they would structurally represent the objects of the problem space. We also discussed on how a Class Model depicted as Class diagrams represented the relationship between these objects as part of the object oriented approach of designing a system
- There are two dimensions for a software intensive system
  - Logical dimension

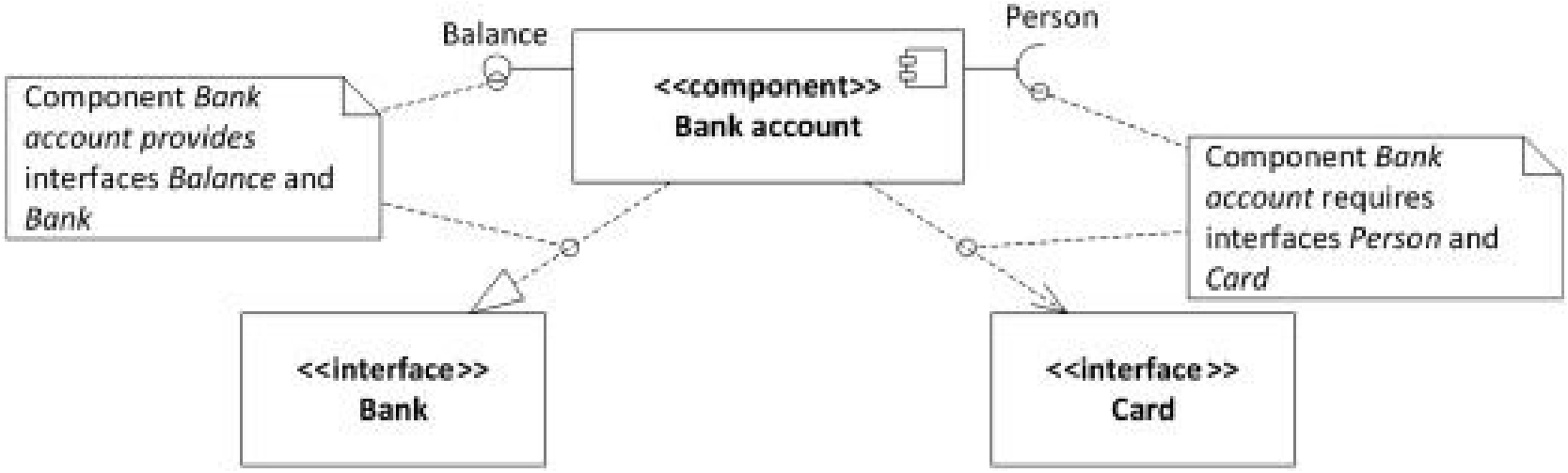
    Addressing the classes, Interfaces, Collaborations, Interactions, States etc.
  - Physical dimension or Implementation dimension which is represented by
    - Components which represents the physical packaging of these logical things
    - Nodes which represent the hardware on which these components are deployed and executed

Thus, couple of other static models which we will consider to complete the physical perspective would be Component Model and Deployment Model

Static Models for Solving a Problem

Component Diagram

- When modelling large object-oriented systems, it is necessary to break down the system into manageable subsystems.
- A component diagram represents the actual physical software components and their dependencies.
- The UML component diagram shows how a software system will be composed of a set of deployable components (represented as diagrams) —dynamic-link library (DLL) files, executable files, or web services—that interact through well-defined interfaces and which have their internal details hidden. Thus



Static Models for Solving a Problem

Component Diagram

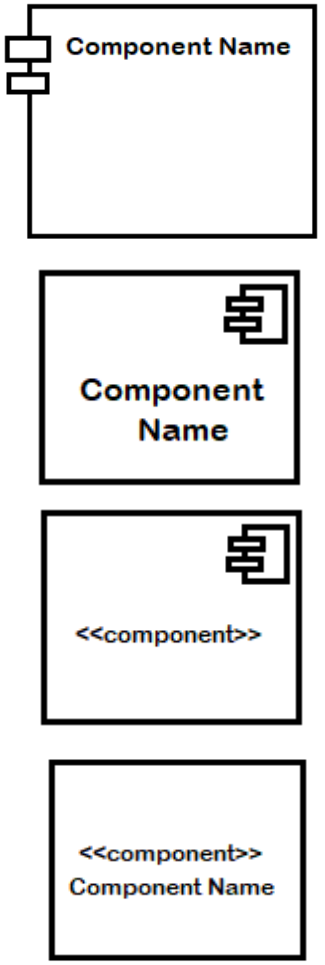
- Components are autonomous units within a system, which are replaceable and executable piece of a system whose implementation details are hidden
- Components can be used to define software systems or subsystems of any size and complexity
- Component diagram describes the organization and wiring of the physical and the conceptual stand-alone components in a system (hence its also called a wiring diagram)
- A component diagram shows the internal parts, connectors, and ports that implement a component. When the component is instantiated, copies of its internal parts are also instantiated.
- Component diagrams are essentially class diagrams which are organized to focus on a system's components that often are used to model the implementation view of a system..

Static Models for Solving a Problem

Component Diagram Notations : Component Notation

Typically a component is represented as below

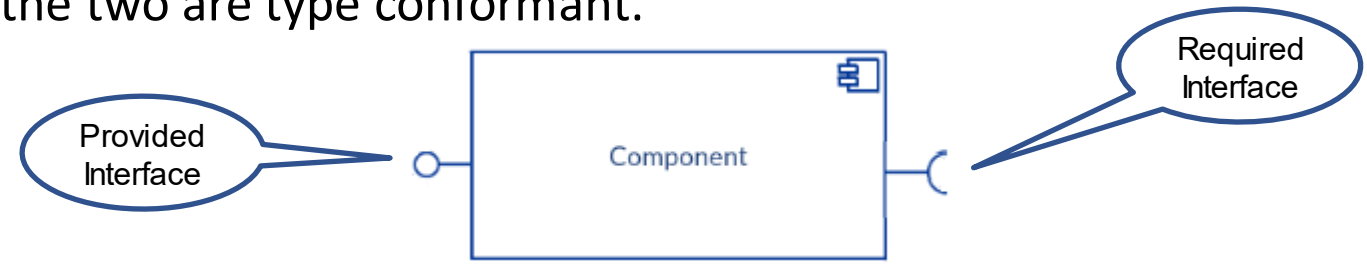
1. Rectangle with the component stereotype (the text <<component>>). The component stereotype is usually used above the component name to avoid confusing the shape with a class icon.
2. Rectangle with the component icon in the top right corner and the name of the component
3. Rectangle with the component icon and the component stereotype.

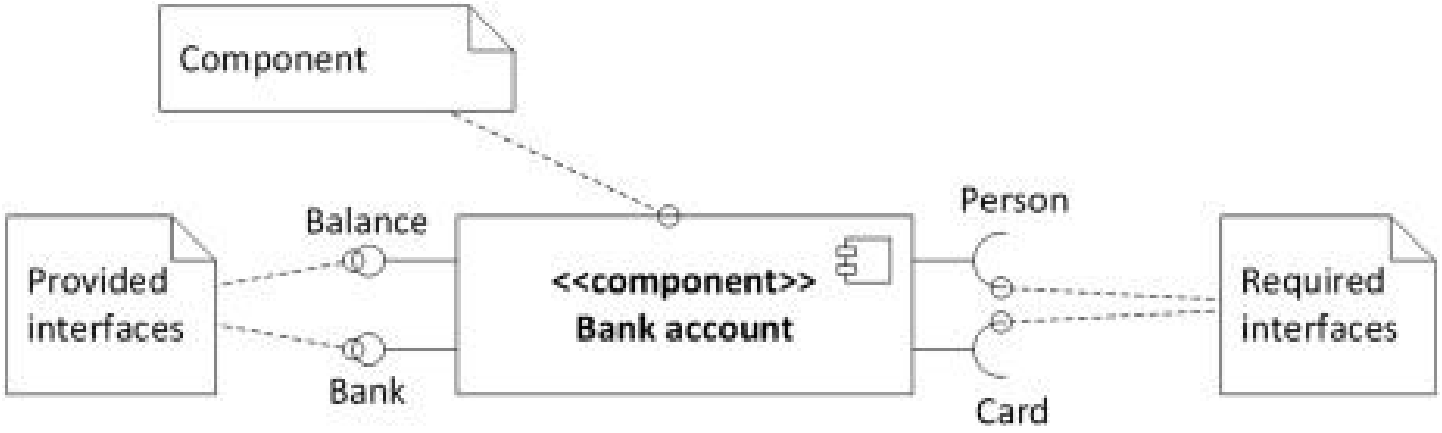


Static Models for Solving a Problem

Component Notation/Representation

- Thus a component represents a modular part of a system that encapsulates its contents, it defines it behavior in terms of provided and required interfaces.
- It serves as a type whose conformance is defined by the provided and required interfaces (encompassing both their static as well as dynamic semantics) as a basil and a socket. One component may therefore be substituted by another only if the two are type conformant.

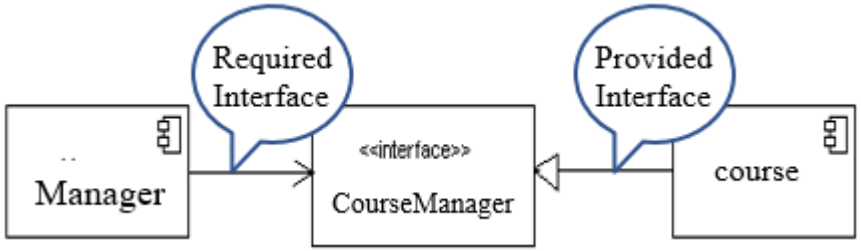




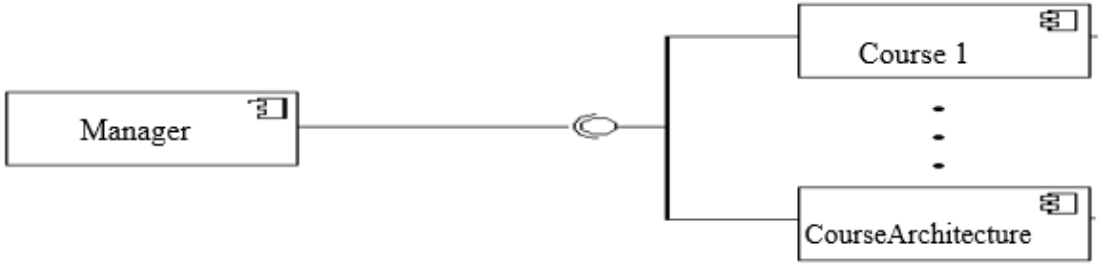
Static Models for Solving a Problem

Component Diagram : Interface

An interface can also be shown using a rectangle symbol along with dependency arrows

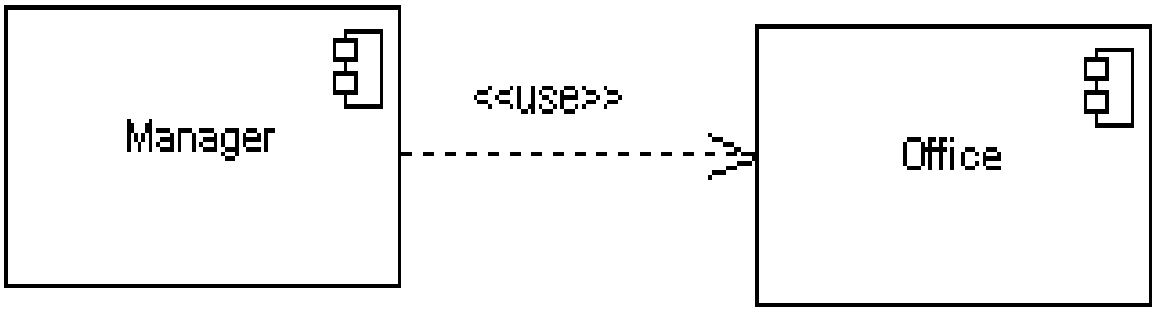


In a system context where there are multiple components that require or provide a particular interface, a notation abstraction can be used that combines by joining the interfaces



Static Models for Solving a Problem

Component Diagram : Usage Dependency



Components can be connected by usage dependencies.

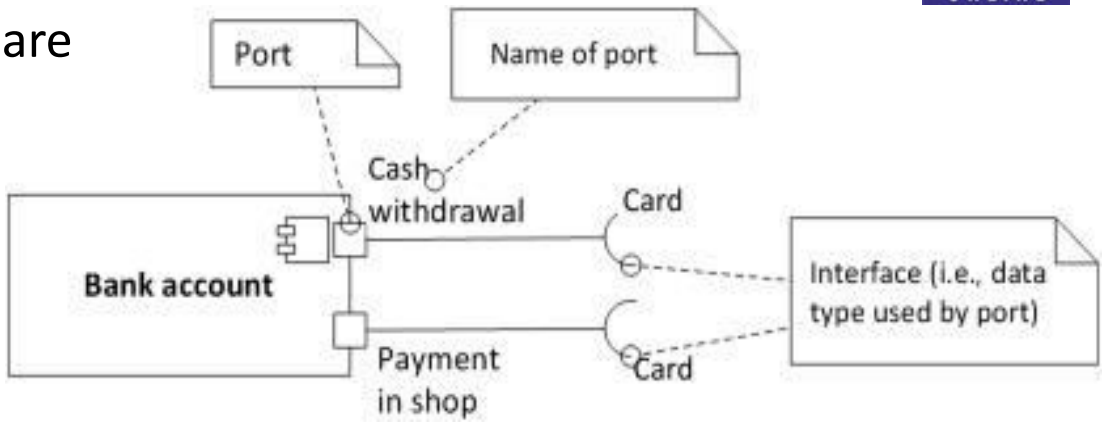
Usage Dependency

- A usage dependency is relationship which one element requires another element for its full implementation
- It is a dependency in which the client requires the presence of the supplier
- It is shown as a dashed arrow with a <<use>> keyword
- The arrowhead point from the dependent component to the one of which it is dependent on

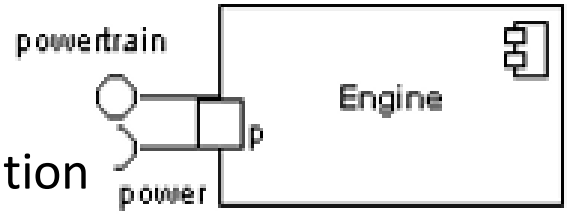
Static Models for Solving a Problem

Component Diagram : Port

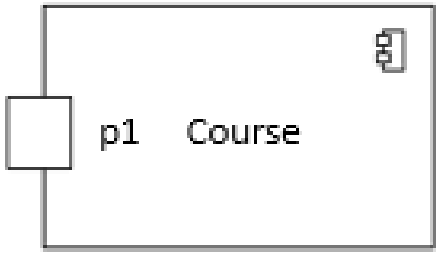
- Port specifies a distinct interaction point or a window into an encapsulated component shown as a small square
- Each port provides or requires one or more specific interfaces.
- Ports can be named, and the name is placed near the square symbol



- There can be multiple ports providing or requiring the same interface. It allows greater control over implementation and interaction with other components.



- Can support uni directional communication or bi-directional communication

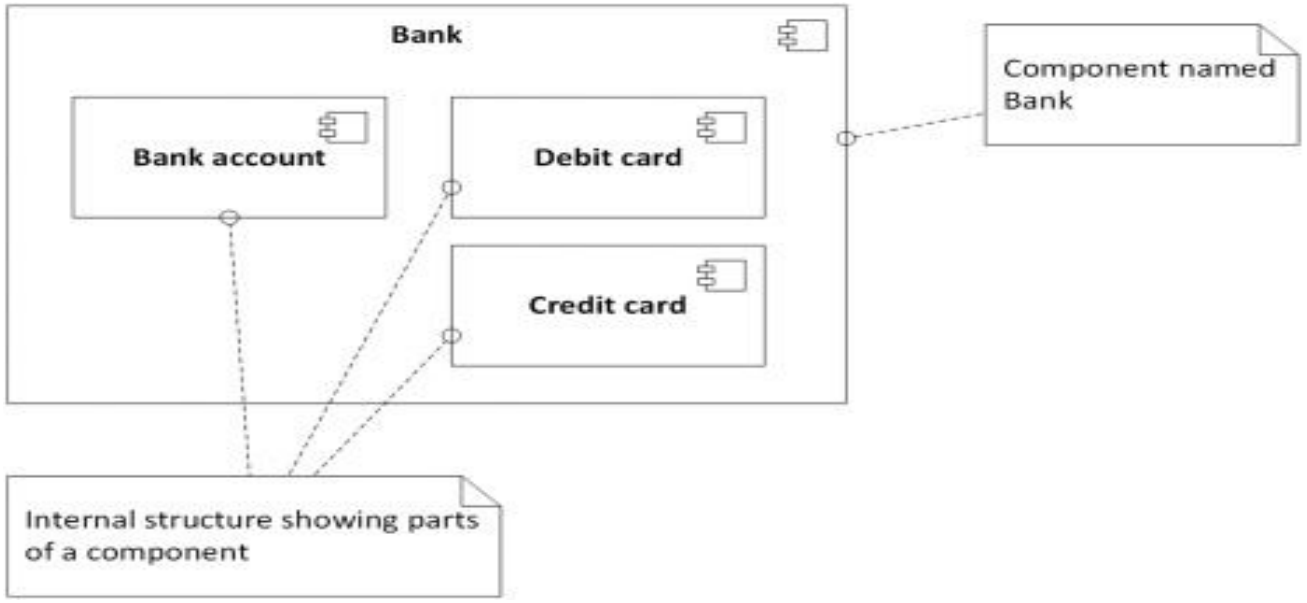


- The internals are fully isolated from the environment
- This allows such a component to be used in any context that satisfies the constraints specified by its ports

Static Models for Solving a Problem

Component Internal Structure

- Used to specify structure or white box view of a complex component, i.e., typically smaller components building up the larger component and the system.

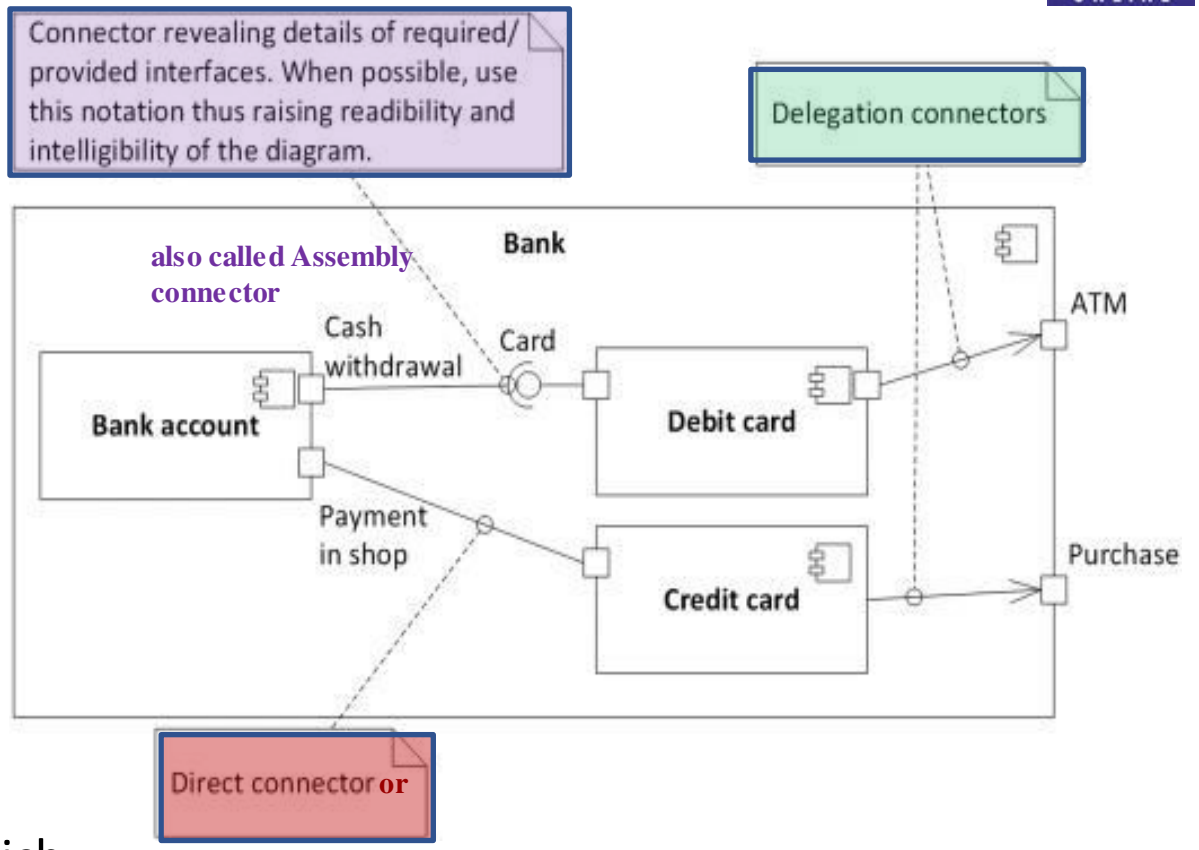


- **Part :** A component that builds up internal structure of a more complex component. You can consider part as a subcomponent.
- **Connector:** A relation between ports of components. If one port provides interface required by the other port, they can be linked together.

Static Models for Solving a Problem

Component Diagram : Connector

- There are several types of connectors that we can draw between parts and components:
- A **direct connector** links together two ports of parts
- A **connector by interfaces or assembly connector** links together two ports by relating together the required and the provided interfaces using ball and socket notation

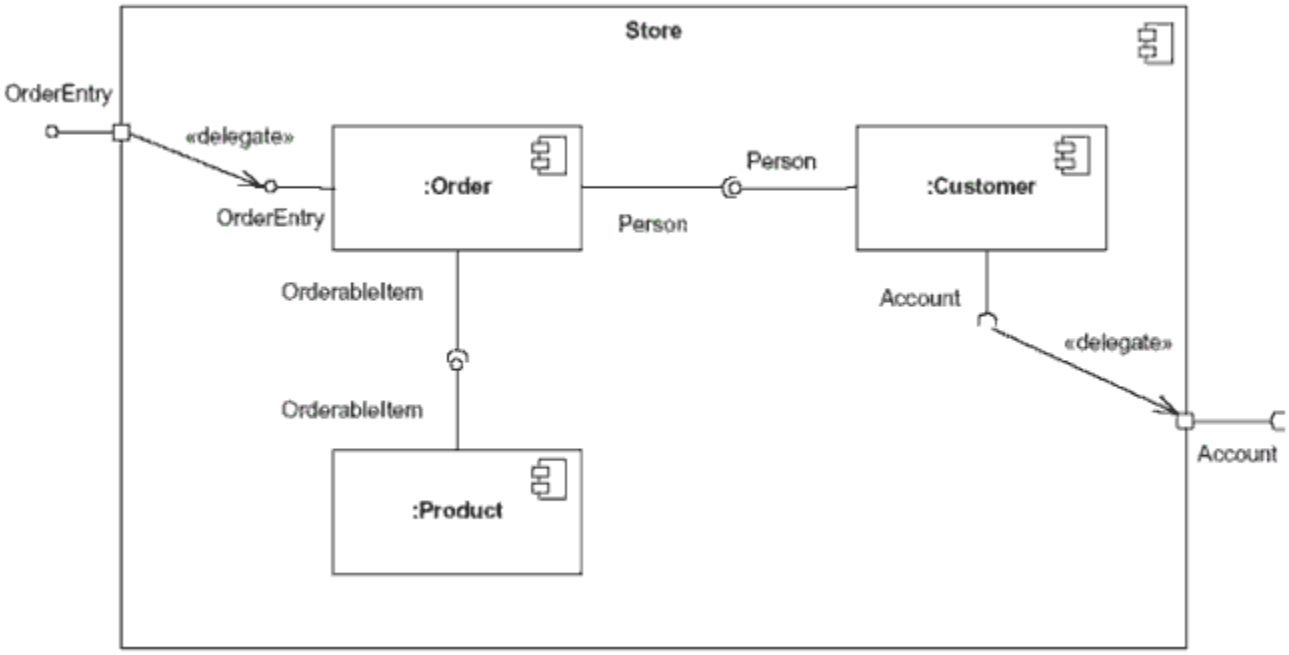


- A **delegation connector** which links together port of a part and port of a component thus providing interface (i.e., provides services to other components) or requiring interface (i.e., consuming services of another component)

Static Models for Solving a Problem

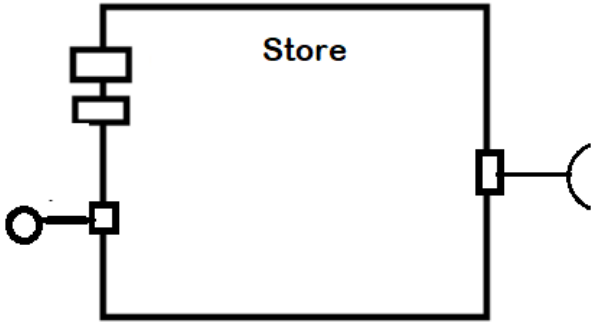
Internal or External View of the Component

Internal View



An internal, or white box view of a component is where the realizing classes/components are nested within the component shape

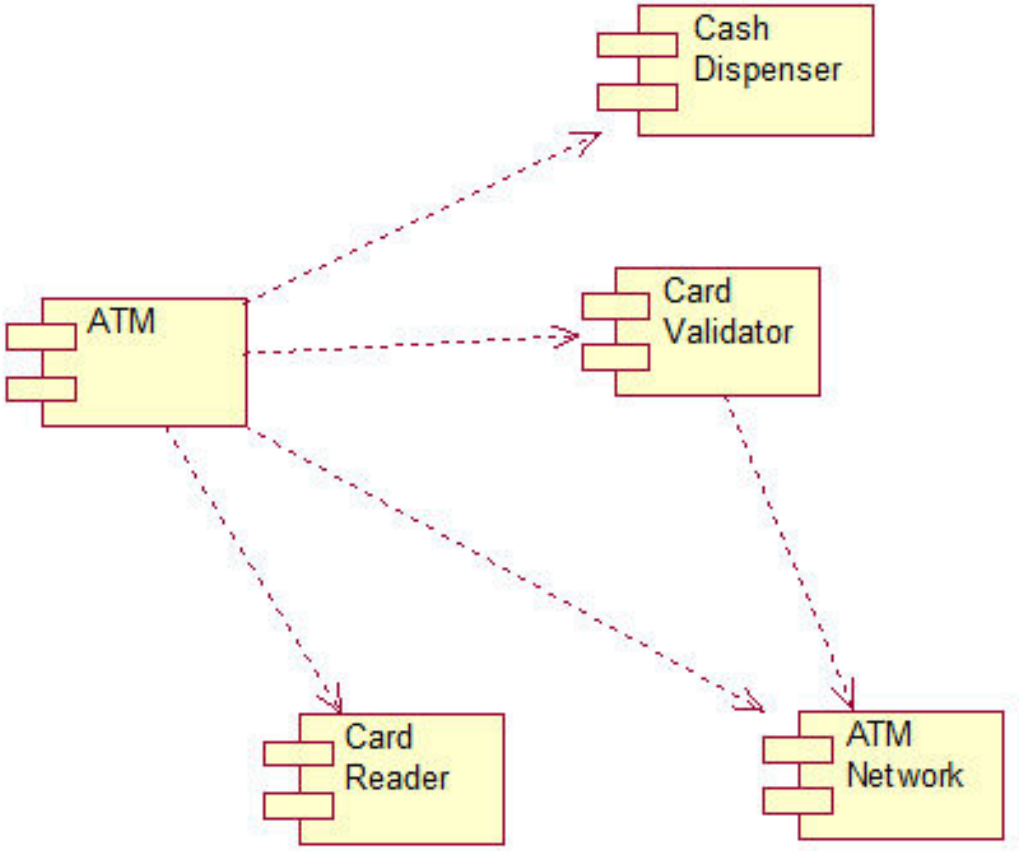
External View



An external view (or black box view) shows publicly visible properties and operations

Static Models for Solving a Problem

E.g. Component Diagram for an ATM component



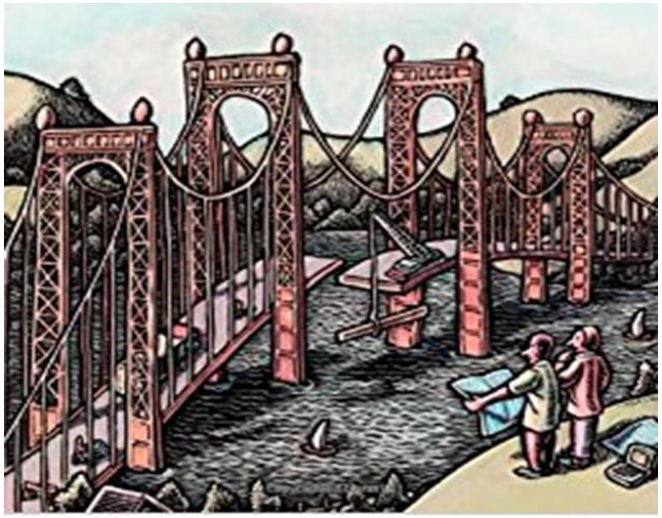
THANK YOU

Dr. H. L. Phalachandra
Department of Computer Science and Engineering
phalachandra@pes.edu

OOAD and SE

Implementation Models

(Deployment Model)



Dr. H.L. Phalachandra
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Static Models for Solving a Problem

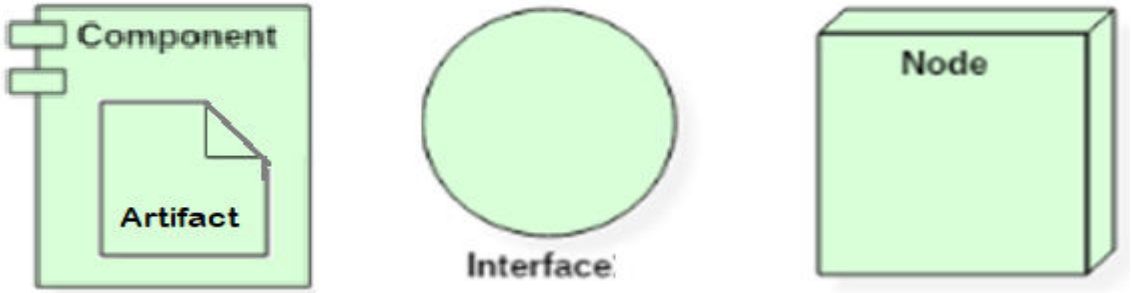
Deployment Diagram

- Software components developed as part of the development process would need to be deployed on some set of hardware or software to be executed
- Deployment diagram maps the software architecture created in design to the physical system architecture that executes it. In distributed systems, it models the distribution of the software across the physical nodes.
- Deployment diagrams shows how the components described in component diagrams are deployed in hardware.
- Deployment diagrams are used to visualize
  - The topology of the physical components of a system, where the software components are deployed.
  - Describe the hardware components used to deploy software components.
  - Describe the runtime processing nodes (physical hardware) .

Static Models for Solving a Problem

Deployment Diagram

- Deployment diagrams are useful for system engineers. An efficient deployment diagram is very important as it controls the following parameters –
  - Performance
  - Scalability
  - Maintainability
  - Portability
  - Understandability
- The software systems are manifested using various artifacts, and then they are mapped to the execution environment like servers called nodes that is going to be execute the software. Since there can be many nodes, the relation between then is represented as communication links



Some of the deployment diagram symbols and notations

Static Models for Solving a Problem

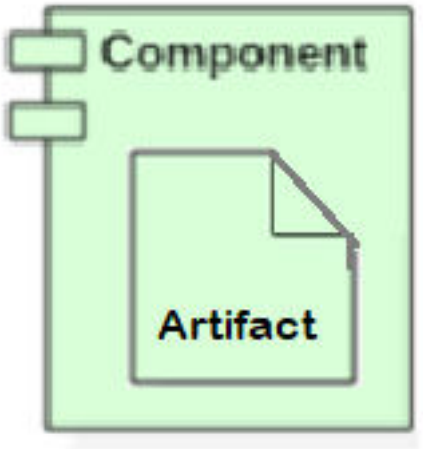
Deployment Diagram

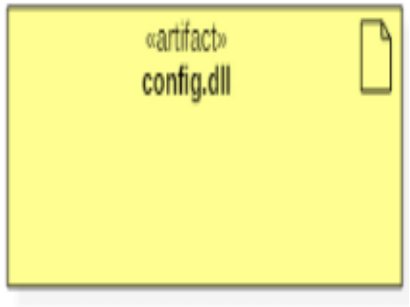
- Component with Artifacts : represents the concrete real-world entity related to software development.
- Things that participate in the execution of a system or executable entities that reside in nodes
- Represent the physical packaging of the logical elements
- Artifacts are deployed on the nodes.

The most common artifacts are as follows,

- Source files
- Executable files
- libraries
- Scripts
- Configuration files
- User manuals or documentation
- Output files

Each artifact has a filename in its specification that indicates the physical location of the artifact.



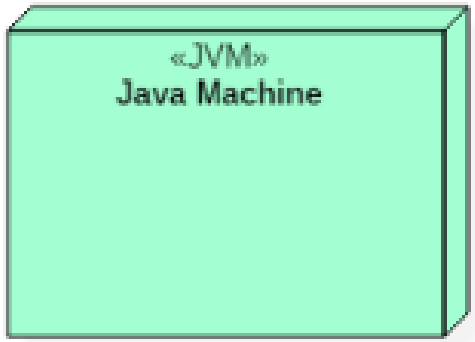


Static Models for Solving a Problem

Deployment Diagram

- Node : Node is a computational resource upon which component artifacts are deployed for execution.

device node



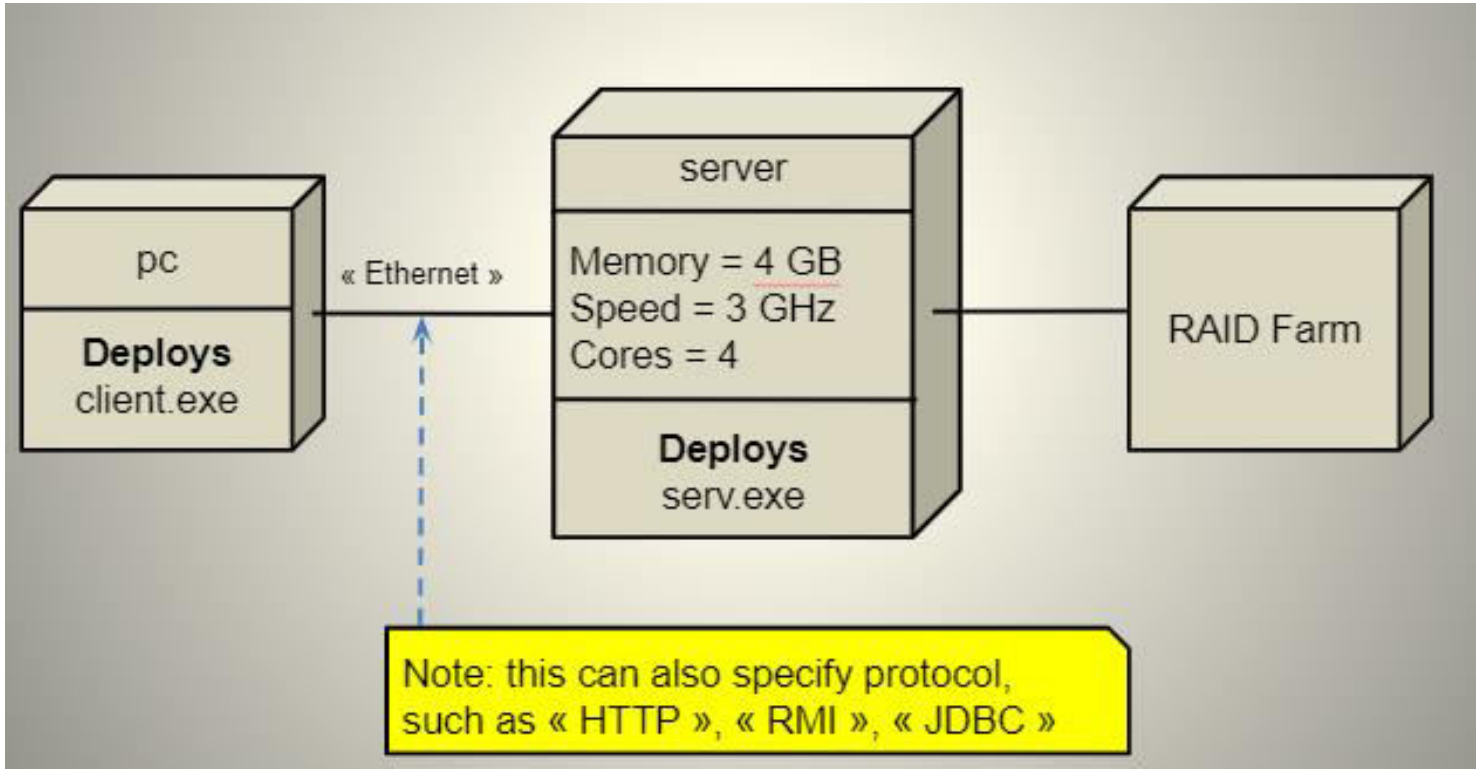
execution environment node

- Executes components or Artifacts that are deployed on nodes
- Physical element in the run-time environment that represents a computational resource
- Can be organized by grouping them in packages
- Set of artifacts or components allocated to a node is a distribution unit
- Device is a node that is used to represent a physical computational resource in a system. An example of the devices is like an application Server
- Connection represents an association among nodes eg. Ethernet

Static Models for Solving a Problem

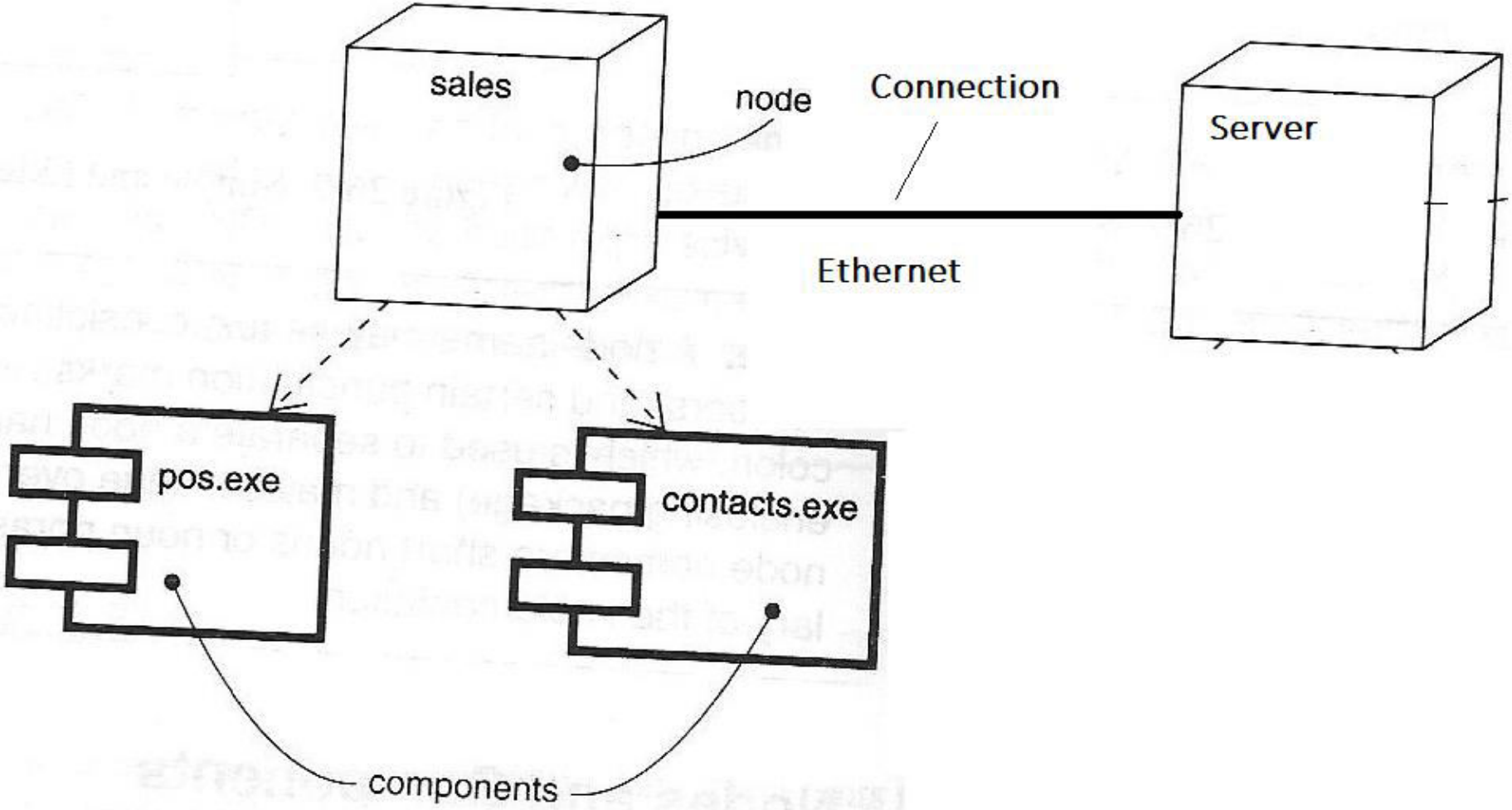
Illustration of a Deployment Diagram

Client Server



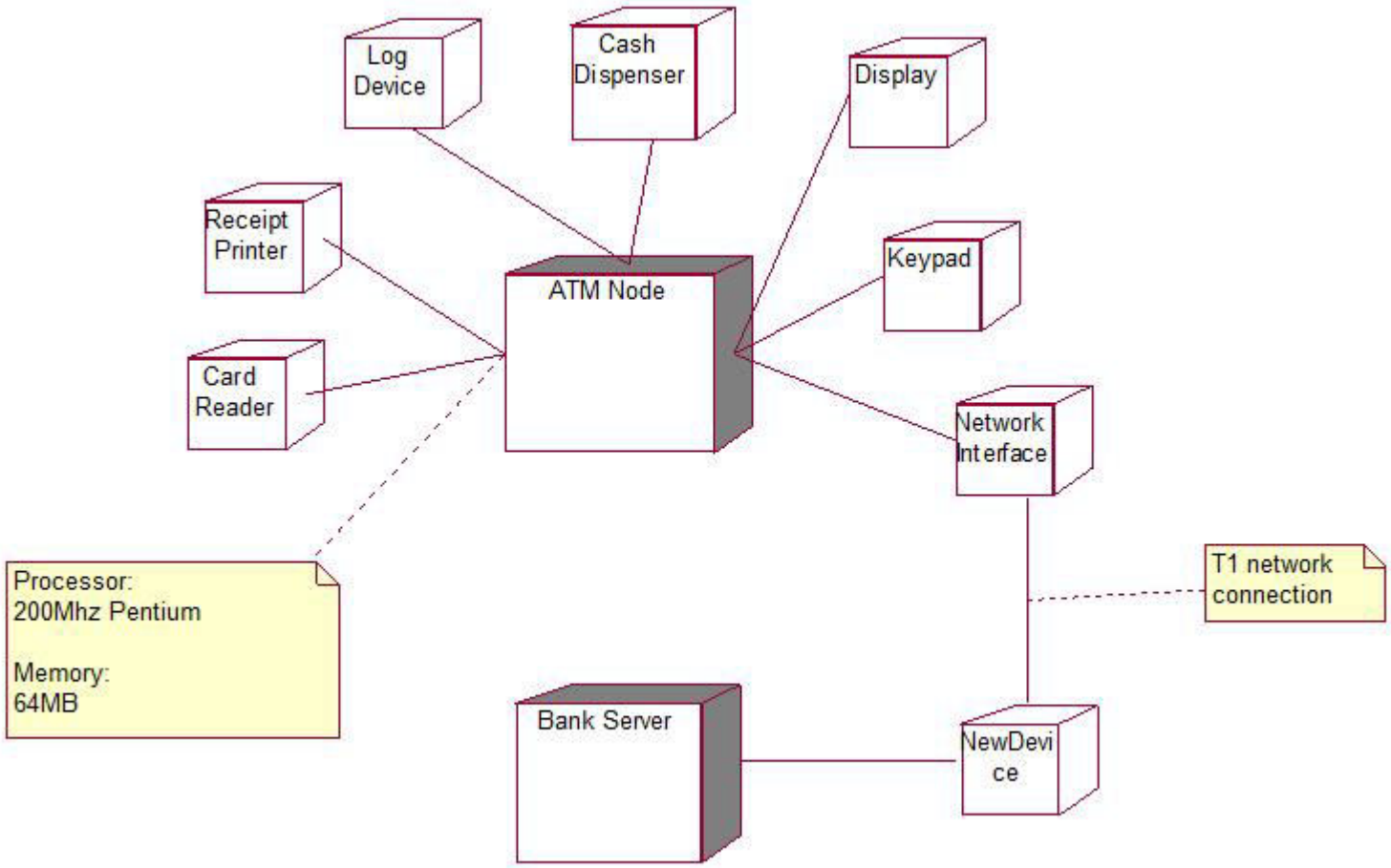
Static Models for Solving a Problem

Illustration of a Deployment Diagram



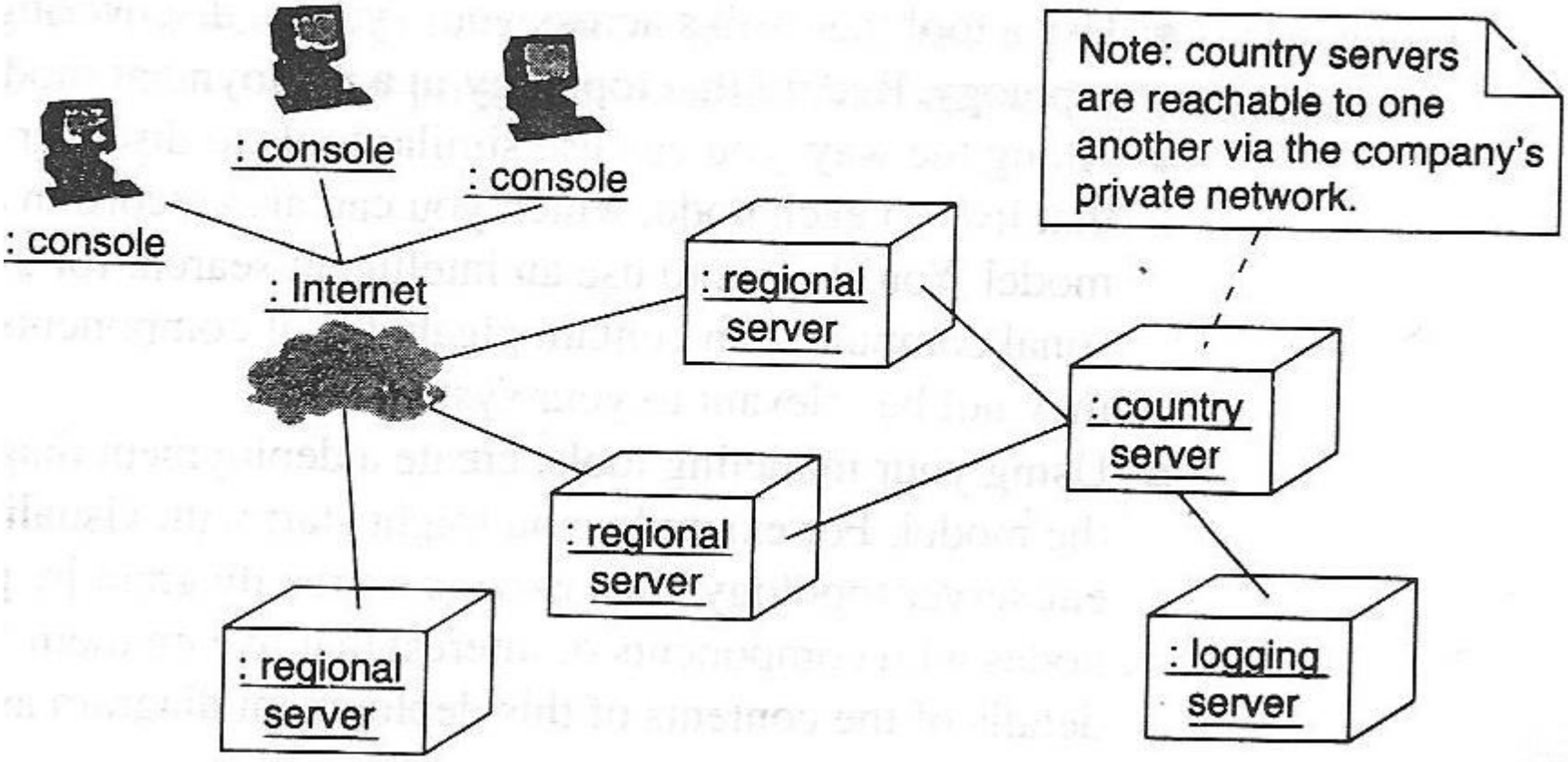
Static Models for Solving a Problem

Illustration of a Deployment Diagram



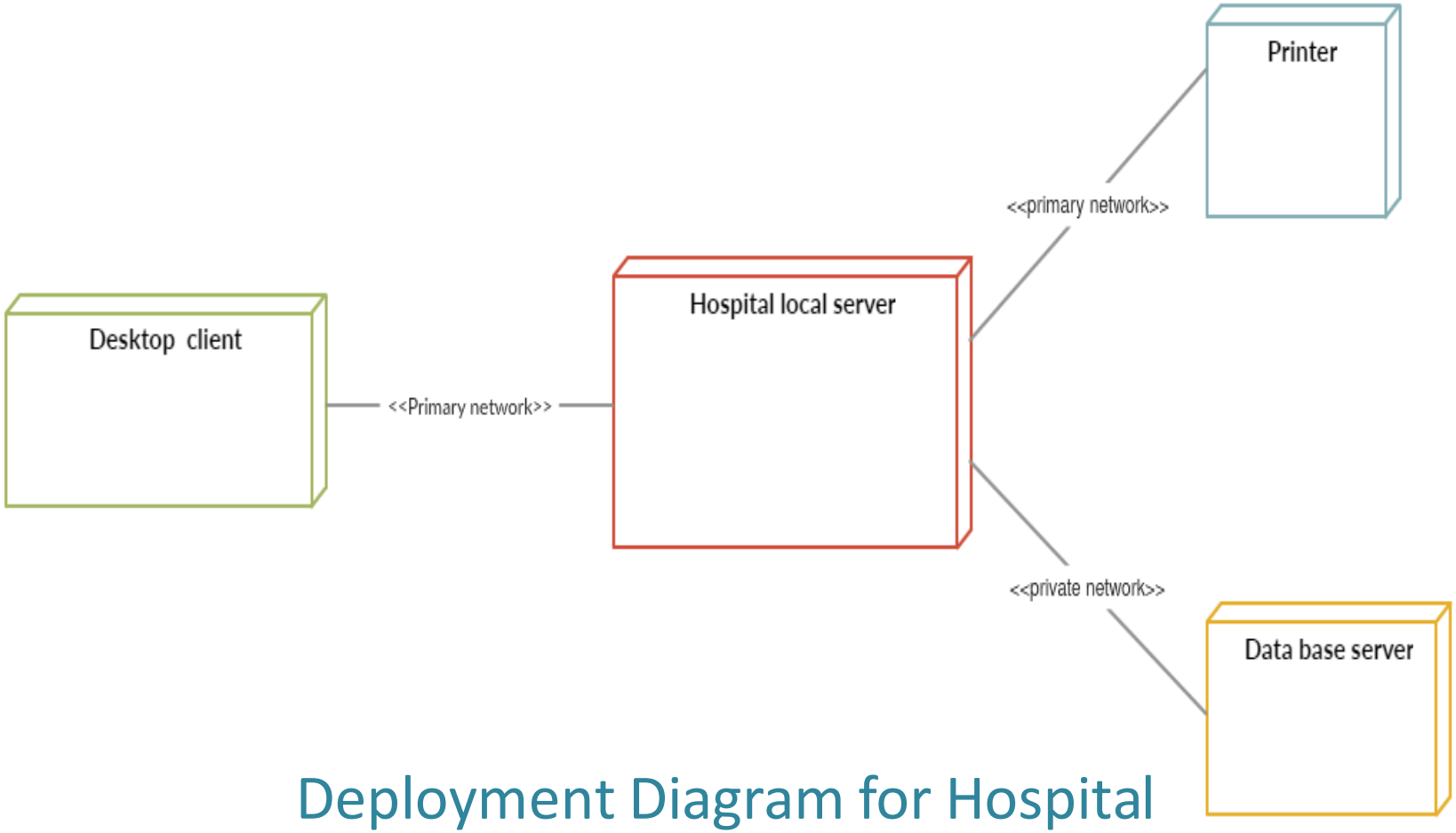
Static Models for Solving a Problem

Illustration of a Deployment Diagram



Static Models for Solving a Problem

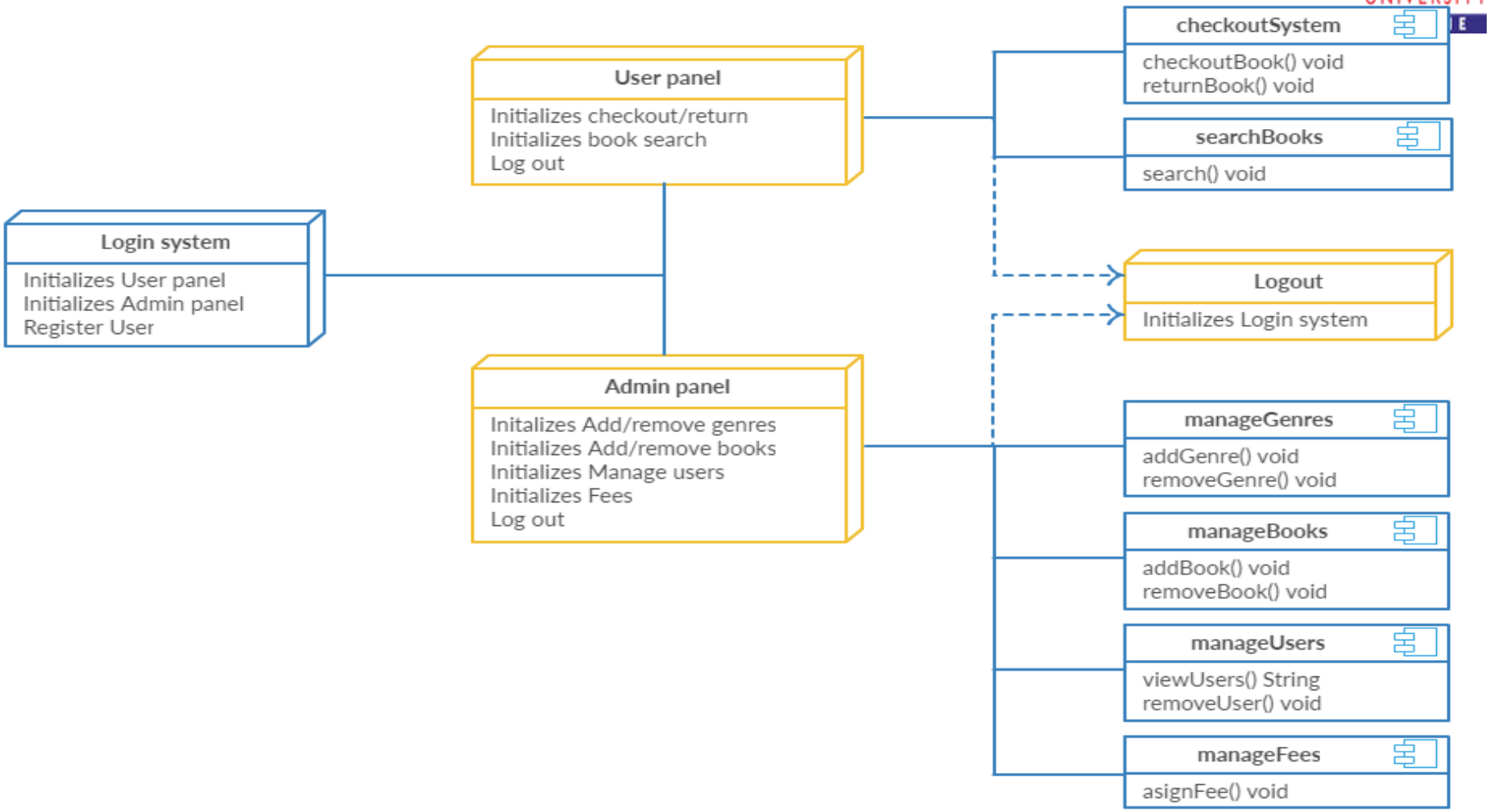
Illustration of a Deployment Diagram



Deployment Diagram for Hospital Management System Template

Static Models for Solving a Problem

Illustration of a Deployment Diagram



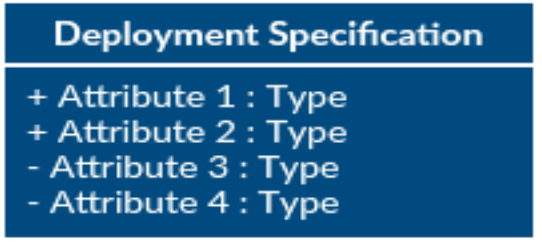
Deployment Diagram for Library Management System Template

Static Models for Solving a Problem

Deployment Diagram :

Deployment Specification

- Deployment specifications is a configuration file, such as a text file or an XML document. It describes how an artifact is deployed on a node.



How to Draw deployment Diagram

Step 1: Identify the purpose of your deployment diagram. And to do so, you need to identify the nodes and devices within the system you'll be visualizing with the diagram.

Step 2: Figure out the relationships between the nodes and devices. Once you know how they are connected, proceed to add the communication associations to the diagram.

Step 3 : Identify what other elements like components, active objects you need to add to complete the diagram.

Step 4: Add dependencies between components and objects as required.

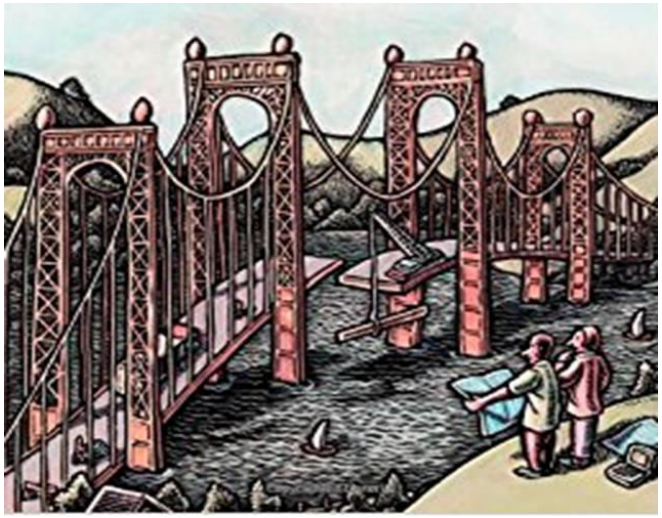
THANK YOU

Dr. H. L. Phalachandra
Department of Computer Science and Engineering
phalachandra@pes.edu

OOAD and SE

Behavioral Models

(Activity Model)



Dr. H.L. Phalachandra
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Behavioral Models for Solving a Problem

Activity Model

- We have looked at the structural models which describe the solution structure in terms of classes/objects their characteristics, their relationships.
- We discussed their representation as a set of component diagrams as part of breaking up the system into implementable smaller manageable submodules.
- We also discussed the deployment diagram which associated these components to physical hosts/devices where these components would be executed.
- In the set of following sessions we will discuss on bringing in the interactions between these classes/objects and components into a set of dynamic or behavioral models.
- There are two basic models which we will discuss, for bringing in the dynamic behavior or interactions of the classes to produce useful results.
  - Activity Models
  - Sequence Models

Behavioral Models for Solving a Problem

Activity Diagrams

- It's a flow chart showing flow of control from activity to activity with the focus on operations rather than on objects
- It involves modelling and showing the ***sequential*** (and possibly ***concurrent***) steps in the computational process (which could be a like a algorithm or workflow) thus factoring in essential dependencies between activities as part of operation
- Actions which are being represented in the Activity diagram, encompass calling another operation, sending a signal, creating or destroying an object or some pure computation such as evaluating an expression
- Activity diagram summarily represents some action which is made up of executable atomic computations that results in change in the state of the system, or the return of a value
- Overall it represents the workflow of the process
- Can be used at any level of abstraction
  - Algorithm
  - Business Process

Behavioral Models for Solving a Problem

Constituents or Notions of an Activity Diagrams

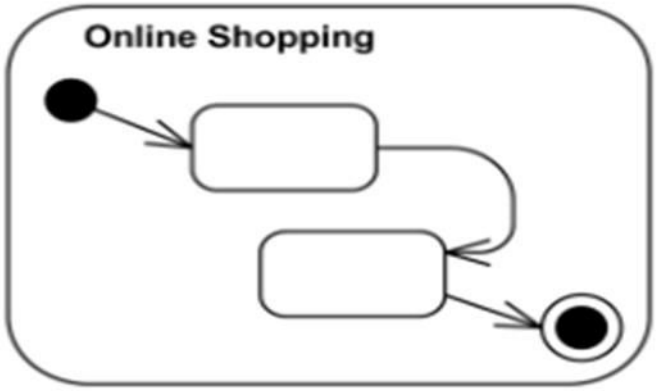
Activity

Activity is a parameterized behavior represented as coordinated flow of actions.

Action

Action is a named element which represents a single atomic step within activity, i.e. that is not further decomposed within the activity or Activity represents a behavior that is composed of individual elements that are actions.

- Activities may contain **actions** of various kinds:
  - Occurrences of primitive functions, such as arithmetic functions.
  - Invocations of behavior
  - Communication actions, such as sending of signals.
  - Manipulations of objects, such as reading or v associations.
- There are actions that invoke other activities - either directly using call behavior or indirectly with call **operation action**



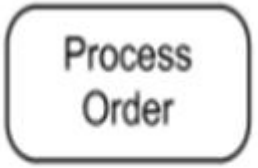
Online Shopping activity.

Behavioral Models for Solving a Problem

Constituents or Notions of an Activity Diagrams

Action (Cont.)

- An action is indicated on the activity diagram by a "capsule" shape – a rectangular object with semi circular left and right ends.
- Name of the action is usually action verb or noun for the action with some explanation
  E.g. Fill Order, Review Document, Receive Order etc.



*The **Process Order** action.*

- An action will not begin execution until all of its **input conditions** are satisfied. The completion of the execution of an action may enable the execution of a set of successor nodes and actions that take their inputs from the outputs of the action

```
for (Account a: accounts)
   a.verifyBalance();
end_for
```

*Example of action expressed with tool-dependent **action language**.*

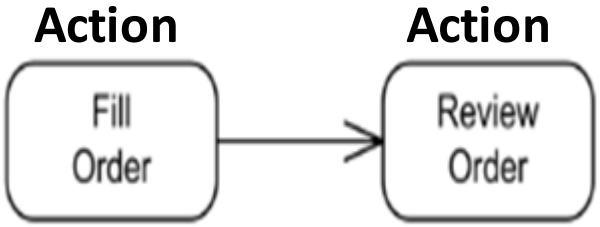
Behavioral Models for Solving a Problem

Constituents or Notions of an Activity Diagrams

Transition (Activity Edge or Transition Line)

- Transition is represented by a directed line and could be named or not

  When the action or an activity state completes, flow of control passes immediately to the next action or activity state E.g.



Activity edge connects Fill Order and Review Order.

- Activity Edge is an abstract class for the directed connections along which tokens or data objects flow between activity nodes. It includes control edges and object flow edges.

  The source and target of an edge must be in the same activity as the edge

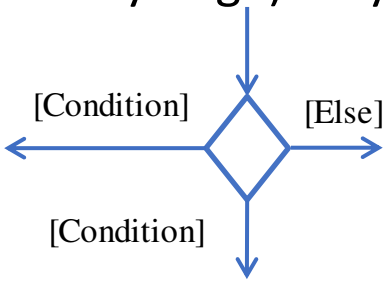
- Activity edge can have a **guard** - specification evaluated at runtime to determine if the edge can be traversed. The guard must evaluate to true for every token that is offered to pass along the edge [Eg. Evaluation of priority to a value = 1]

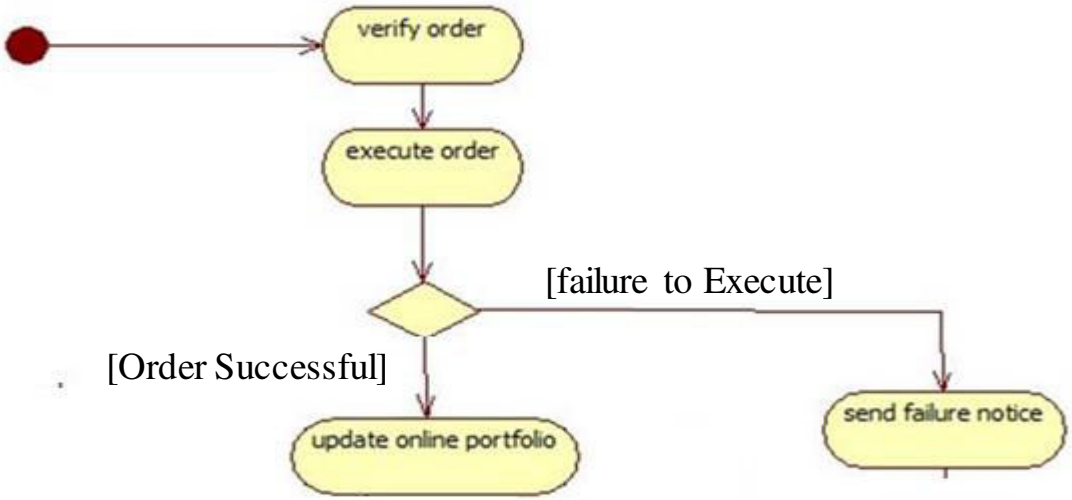
Behavioral Models for Solving a Problem

Constituents or Notions of an Activity Diagrams

Branches

- If there is more than one successor to an action, each arrow (activity edge) may be labelled with a condition in square bracket.
- It specifies alternate paths taken based on some Boolean expression
- This branch is represented as a diamond. This could have one incoming path and more than one outgoing paths





Behavioral Models for Solving a Problem

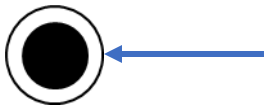
Constituents or Notions of an Activity Diagrams

Initiation and Termination

- A solid circle with an outgoing arrow shows the starting point for the action sequence within an activity diagram. It starts with the solid circle (Start Marker also known as initial state) and proceeds via the outgoing arrow towards the first activities

There can *be only one* initial state on an activity diagram and *only one* transition line connecting the initial state to an action.

- A bulls-eye (a solid circle surrounded by a hollow circle shows the termination point. At this stage overall activity is complete (Stop Marker)



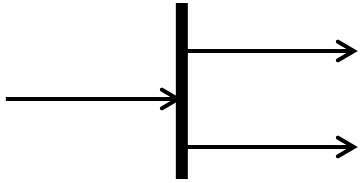
Behavioral Models for Solving a Problem

Constituents or Notions of an Activity Diagrams

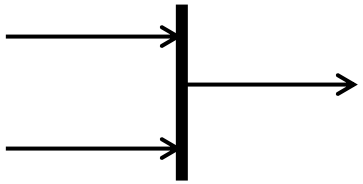
Concurrency

Systems and Organizations can perform more than one actions at a time. These actions can happen at different speeds. These are represented using the fork and solid line to synchronize the beginning. After all the actions are completed, this is marked of completion of the activity using Merge

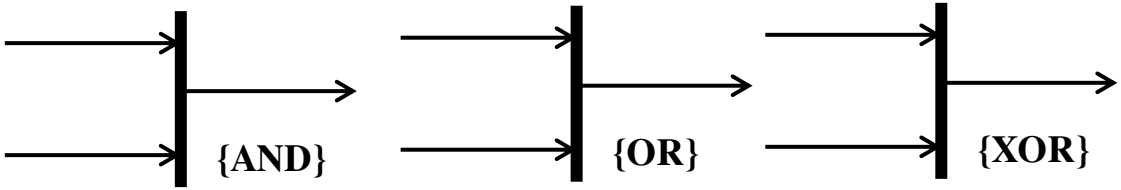
- Fork
  - The activities of each of outgoing transitions are concurrent
  - Order of initiation is not specified or guaranteed
- Merge/Join
  - Actions may end in different order
  - Variants

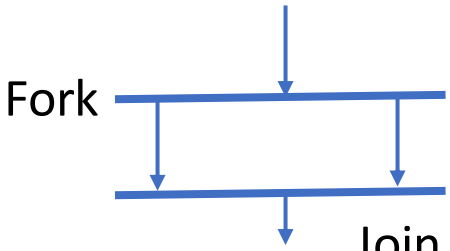


Splitting Bar (Fork)



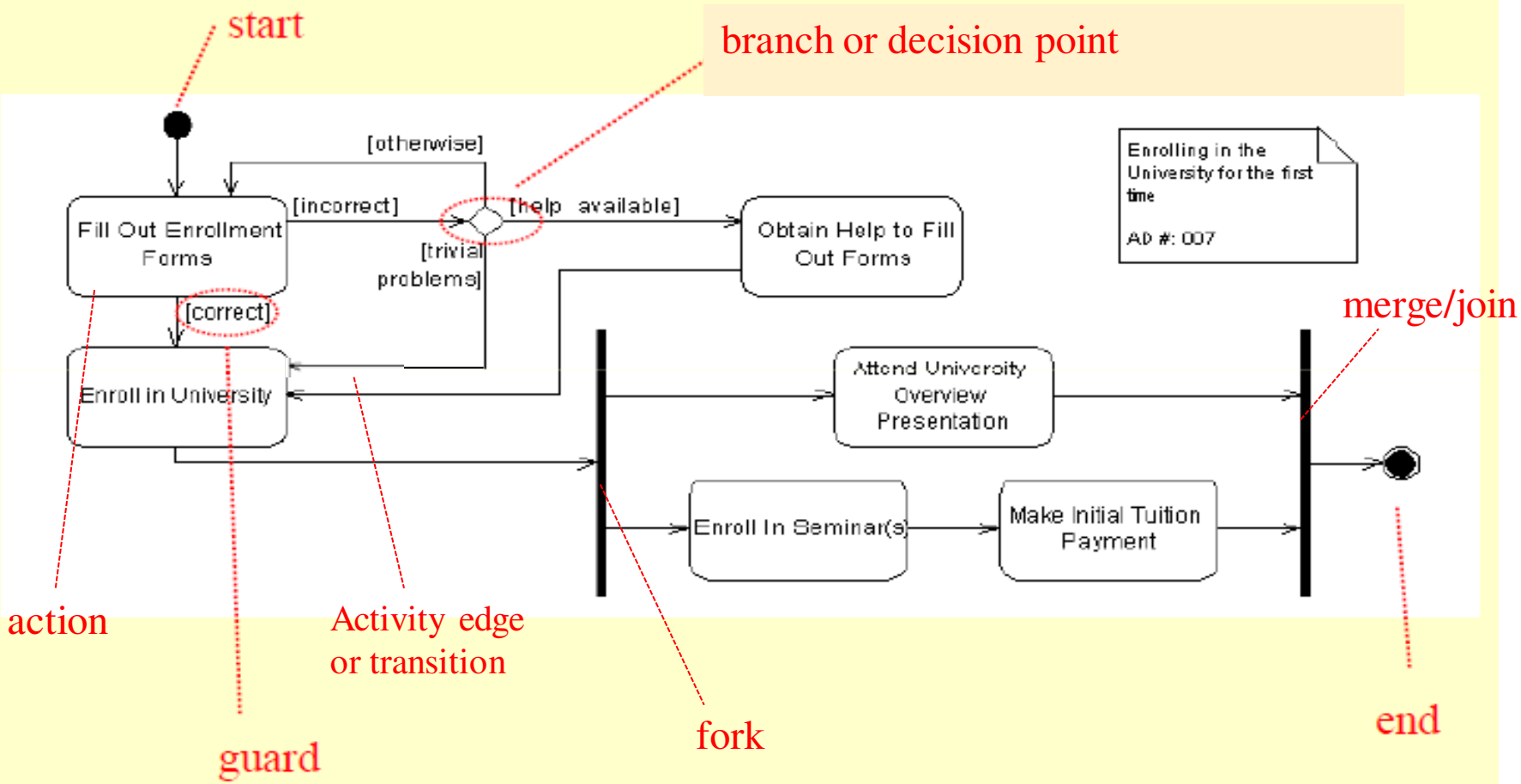
Synch. Bar (Join/Merge)





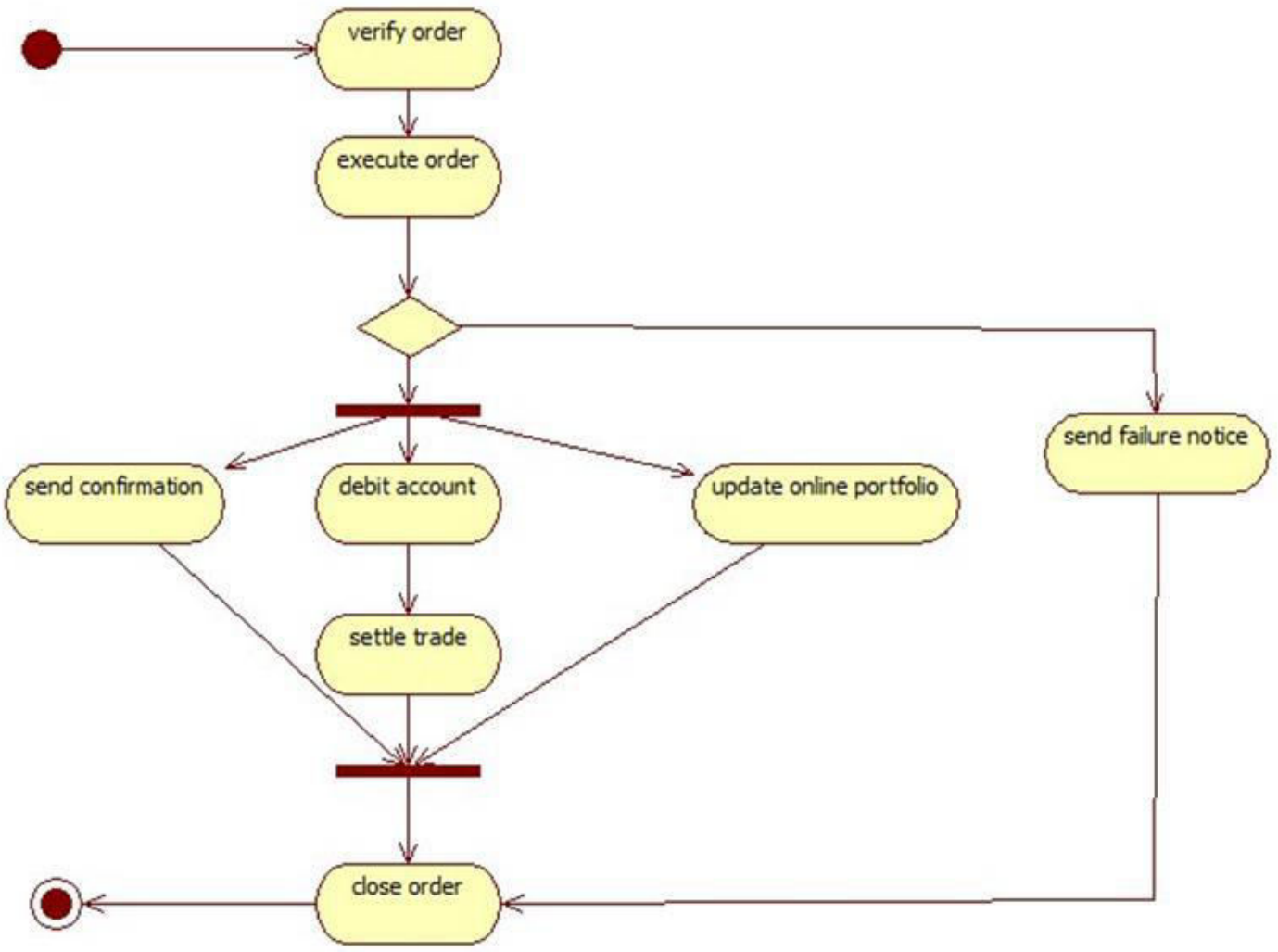
Behavioral Models for Solving a Problem

Activity Diagrams



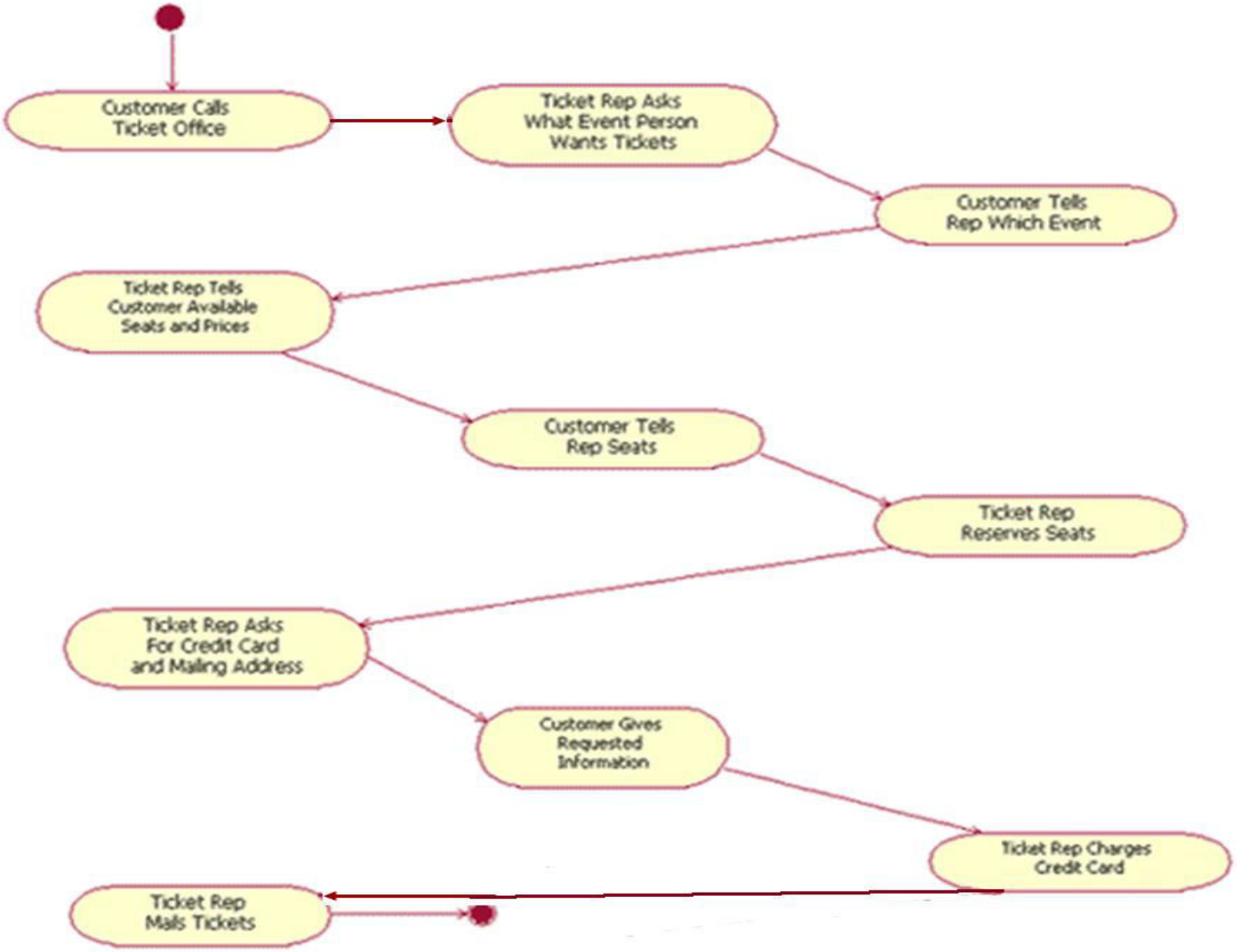
Behavioral Models for Solving a Problem

Activity diagram (Stock Trade Processing)



Behavioral Models for Solving a Problem

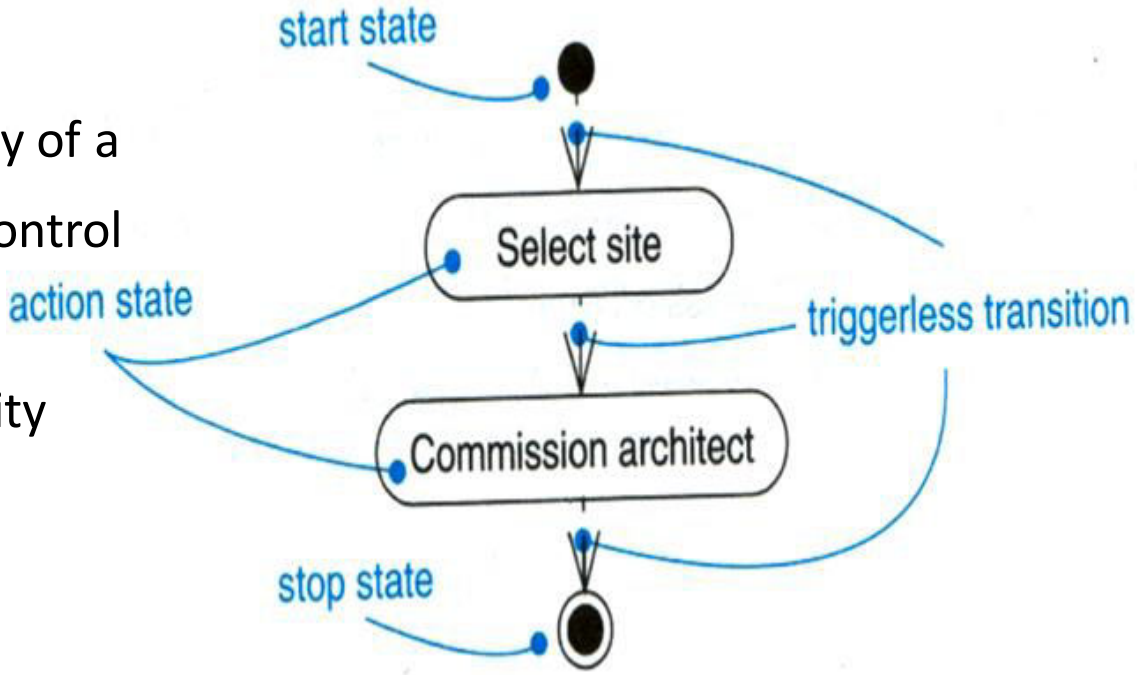
Activity Diagrams



Behavioral Models for Solving a Problem

Activity diagram

- Activity states and action states
  - Action states are atomic and cannot be decomposed
  - Activity states can be further decomposed
    - Their activity being represented by other activity diagrams
    - They may be interrupted
- Transitions
  - When the action or activity of a state completes, flow of control passes immediately to the next action or activity state

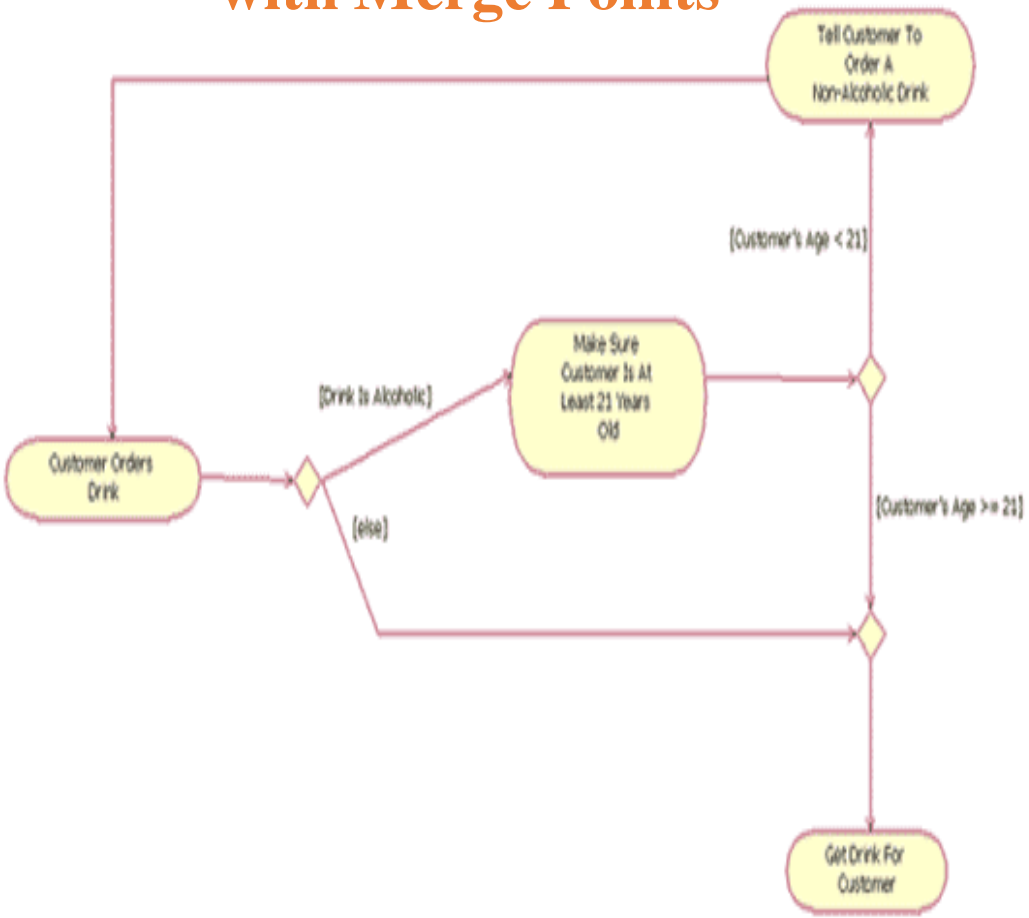


Behavioral Models for Solving a Problem

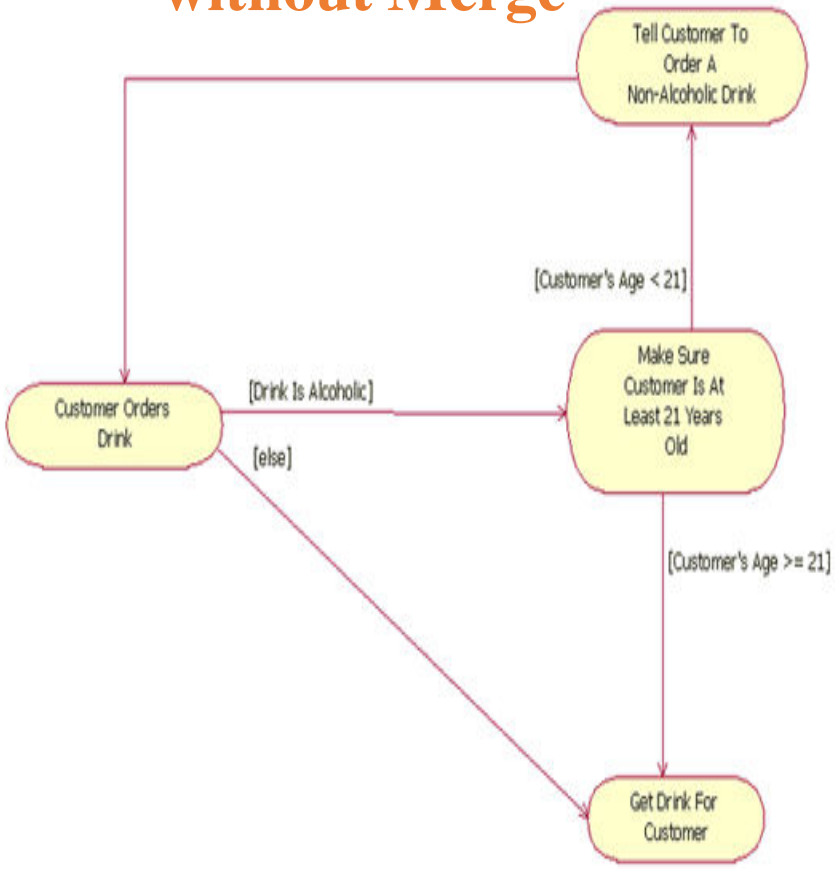
Partial Activity diagram`

A partial activity diagram, showing two decision points ("Drink is alcoholic" and "Customer's age < 21") and one merge ("else" and "Customer's age >= 21")

with Merge Points



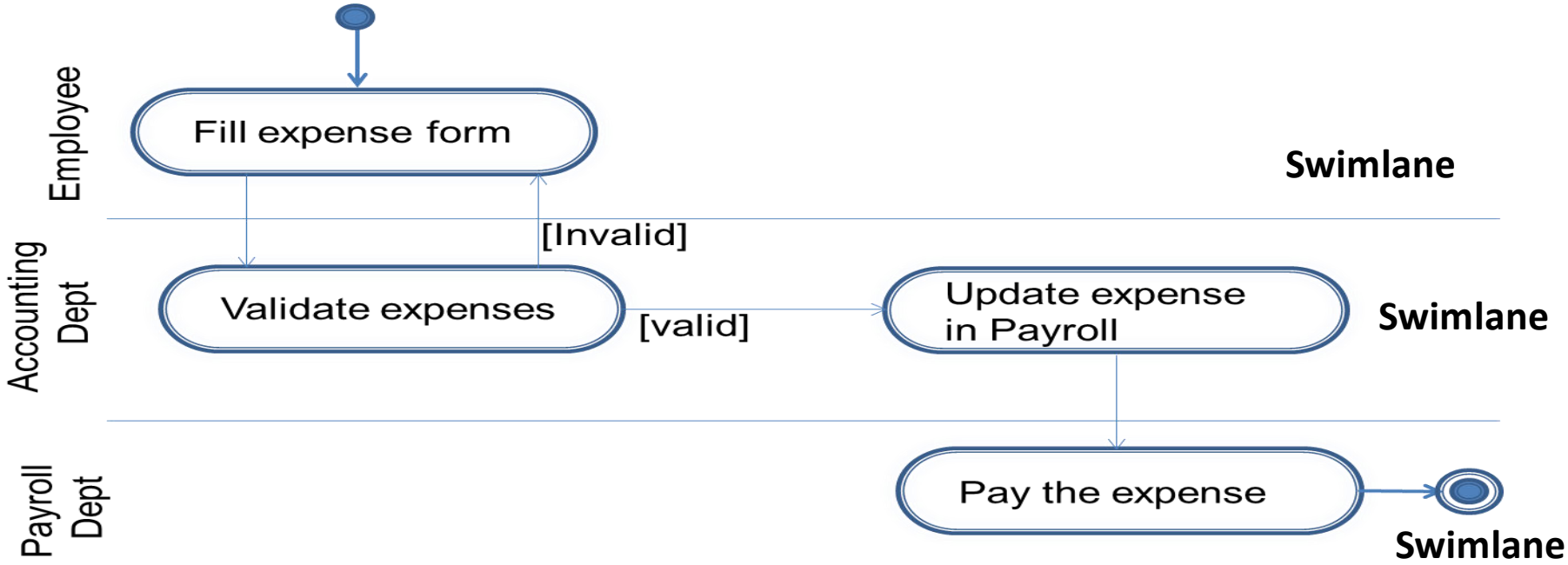
without Merge



Behavioral Models for Solving a Problem

Swim Lanes

- ***Swimlane*** **or** ***an activity partition*** is a high-level grouping of a set of related actions with straight line boundaries as placing responsibilities with competitors at a swim meet.
- This helps in partitioning actions on an activity diagrams into groups where each group could represent an Business Organization responsible for the activities
- In UML each of the partitions (typically are the activities of a single actor or a group of activities happening in a single thread) or swimlanes are visually depicted as a solid horizontal line (or a vertical line)



Swim Lanes

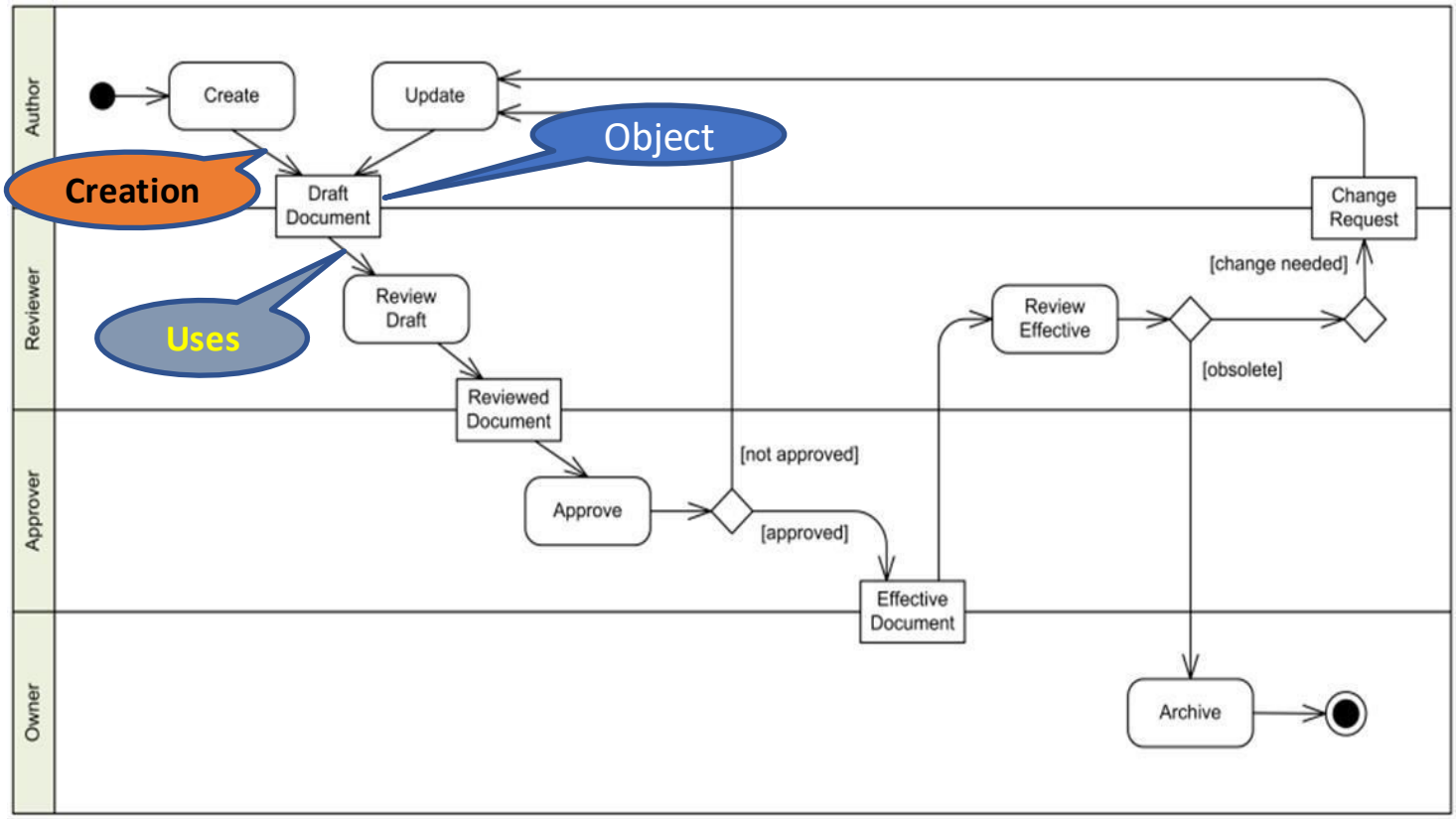
SwimLanes : (Cont.)

- Help understand “division” of work and activity flow across involved parties
  - Interface points (swimlane boundaries) are clarified

Behavioral Models for Solving a Problem

Swim Lanes/Object Flow

- Object flow refers to the creation and modification of objects by activities
- An object flow arrow from an action to an object means that the action creates or influences the object.
- An object flow arrow from an object to an action indicates that the action state uses the object.
- In activities it would be important to have the input and output of activity
- Key objects can be showed as part of the activity flow



Behavioral Models for Solving a Problem

Guidelines for Activity Diagram

- **Don’t misuse activity diagram** :- Elaborate use case & sequence model to study algorithm & workflow.
- **Level diagram :-** Activities on a diagram should be consistent level of detail. place additional details for activity in separate diagram.
- **Be careful with branches & conditions**:- At least one condition should be satisfied.
- **Be careful with concurrent activities** :- Before a merge can happen all inputs must first complete.

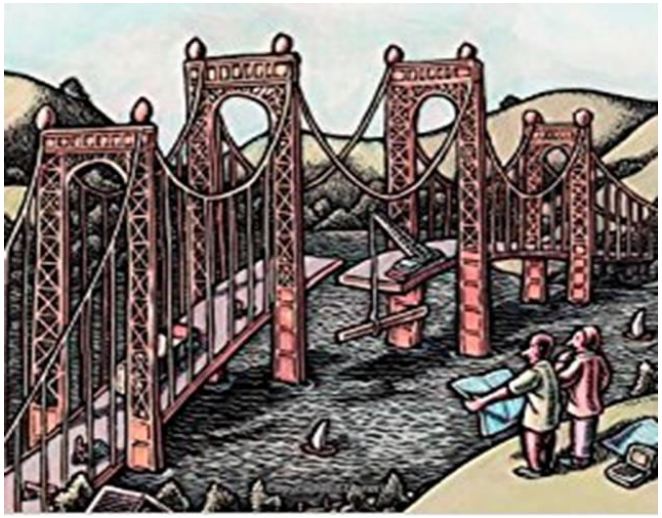
THANK YOU

Dr. H. L. Phalachandra
Department of Computer Science and Engineering
phalachandra@pes.edu

OOAD and SE

Behavioral Models

(Sequence Model)



Dr. H.L. Phalachandra
Department of Computer Science and Engineering
leveraging some information from slides of
Prof. Vinay Joshi

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Behavioral Models for Solving a Problem

Sequence Diagram (Event Diagram)

- We have looked at the Activity model for bringing in the dynamic behavior or interactions of the classes. A sequence diagram depicts interaction between objects in a sequential order i.e. the order in which these interactions take place to support a use case.
- Sequence diagrams are derived from Use Cases
- Sequence diagrams are good for showing behavior sequence as seen by others

Behavioral Models for Solving a Problem

Scenarios

- A scenario is a sequence of events that occurs during one particular execution of a system or a use case realization.
- The scope of a scenario can vary;
  - it may include all events in the system,
  - or it may include only those events generated by certain objects.
- A scenario can be displayed as a list of text statements.
- A scenario contains messages between objects as well as activities performed by objects.
- Each message transmits information from one object to another. Initially the statements can be described at a high level and may require multiple messages, which can get drilled down during later phases of design
- Steps for creating a scenario would be
  - Identify the objects exchanging messages
  - Determine the sender and receiver of each messages and the sequence of messages

Behavioral Models for Solving a Problem

Scenario for session with an online stock broker

- John Doe logs in.
- System establishes secure communications.
- Systems displays portfolio information.
- John Doe enters a buy order for 100 shares of GE at the market price.
- System verifies sufficient funds for purchase.
- System displays confirmation screen with estimated cost.
- John Doe confirms purchase.
- System places order on securities exchange.
- System displays transaction tracking number.
- Jon Doe logs out.
- System establishes insecure communication.
- System displays good-bye screen.
- Securities exchange reports results of trade.

Behavioral Models for Solving a Problem

Key Parts of a Sequence Diagram for a scenario

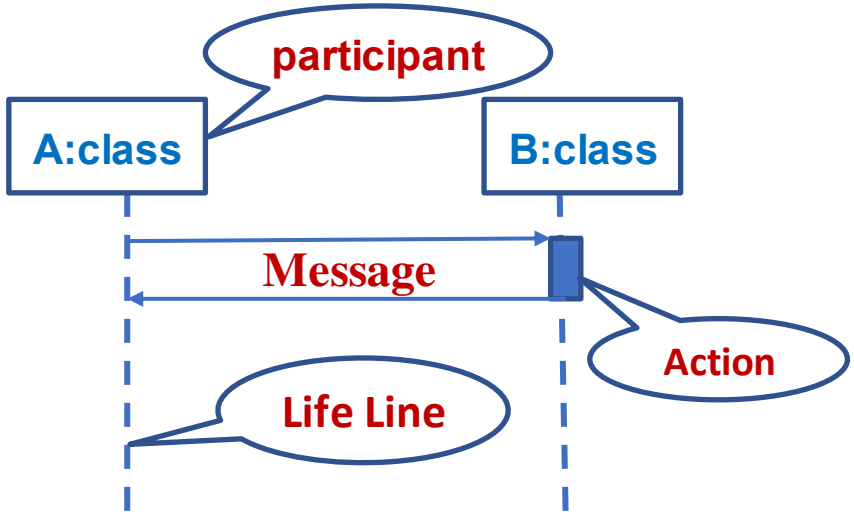
participant: object or entity from a class that acts in the diagram

message: communication between participant objects.

Action: A fragile rectangle can express it on a lifeline. These are also called the method-invocation boxes, and indicate that an object is part of an action. It starts when the message is received and ends when the object is done handling the message.

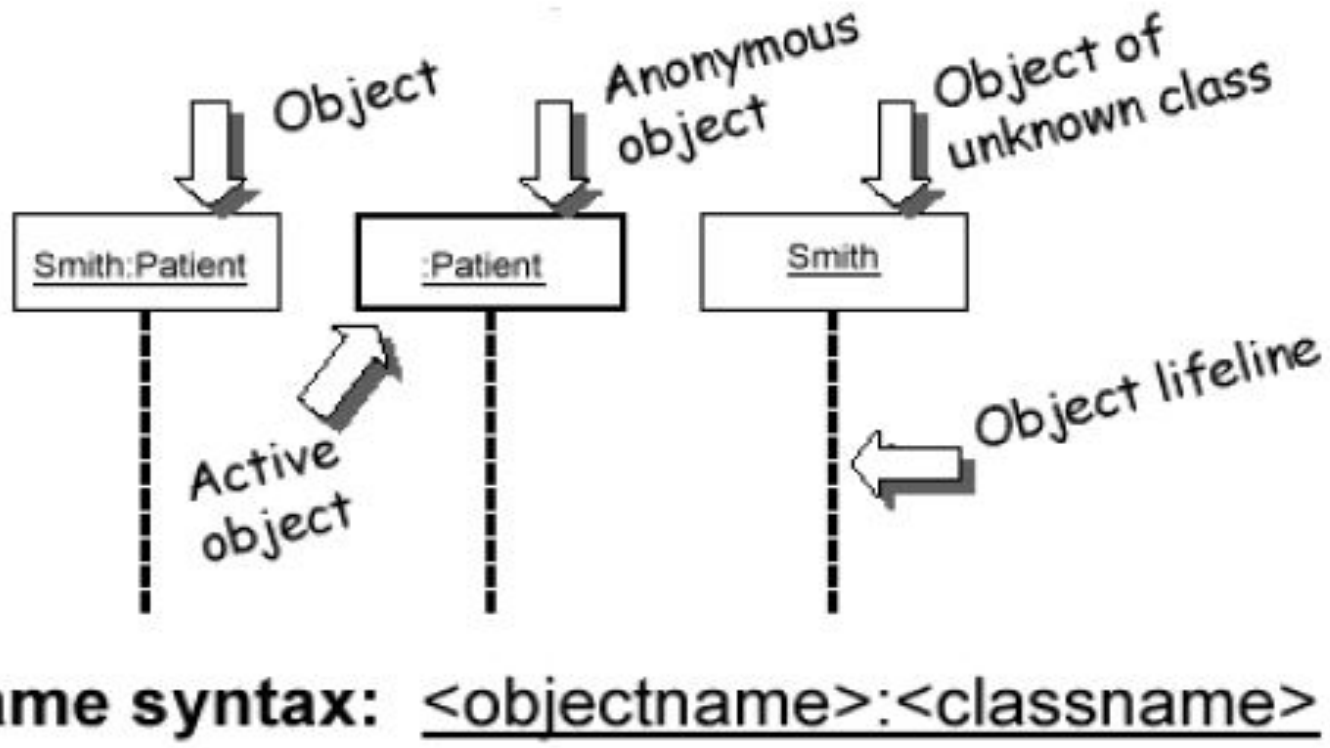
Axis in a sequence diagram:

- **Vertical:** time (moving down signifies moving forward in time). This is also called the life line
- **Horizontal:** which object/participant is acting (and sending messages)



Behavioral Models for Solving a Problem

Representing Objects



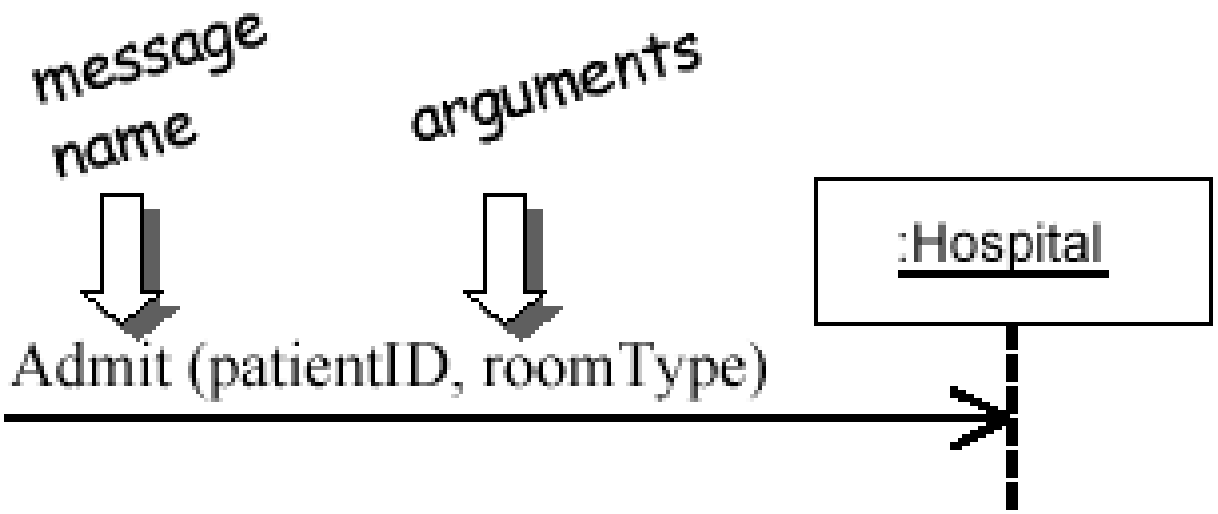
Name syntax: <objectname>:<classname>

Behavioral Models for Solving a Problem

Representing Objects

message (method call) indicated by horizontal arrow to other object

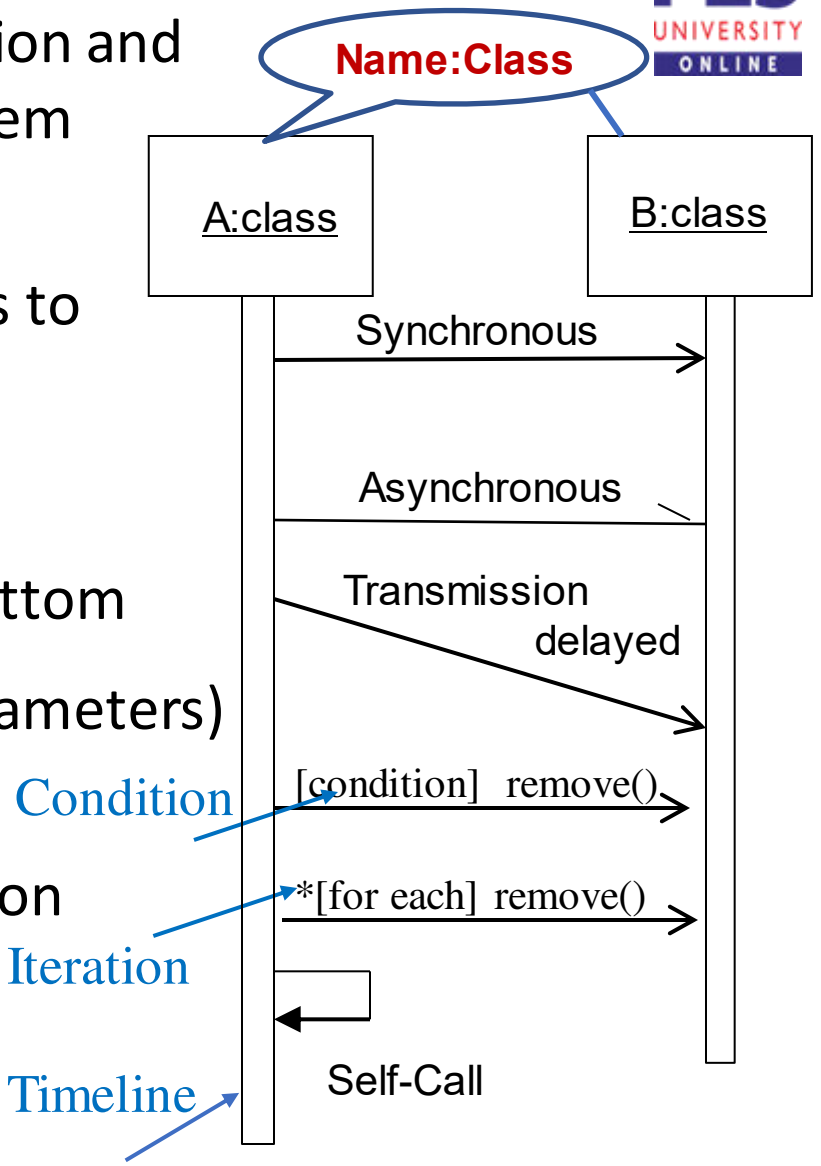
- write message name and arguments above arrow



Behavioral Models for Solving a Problem

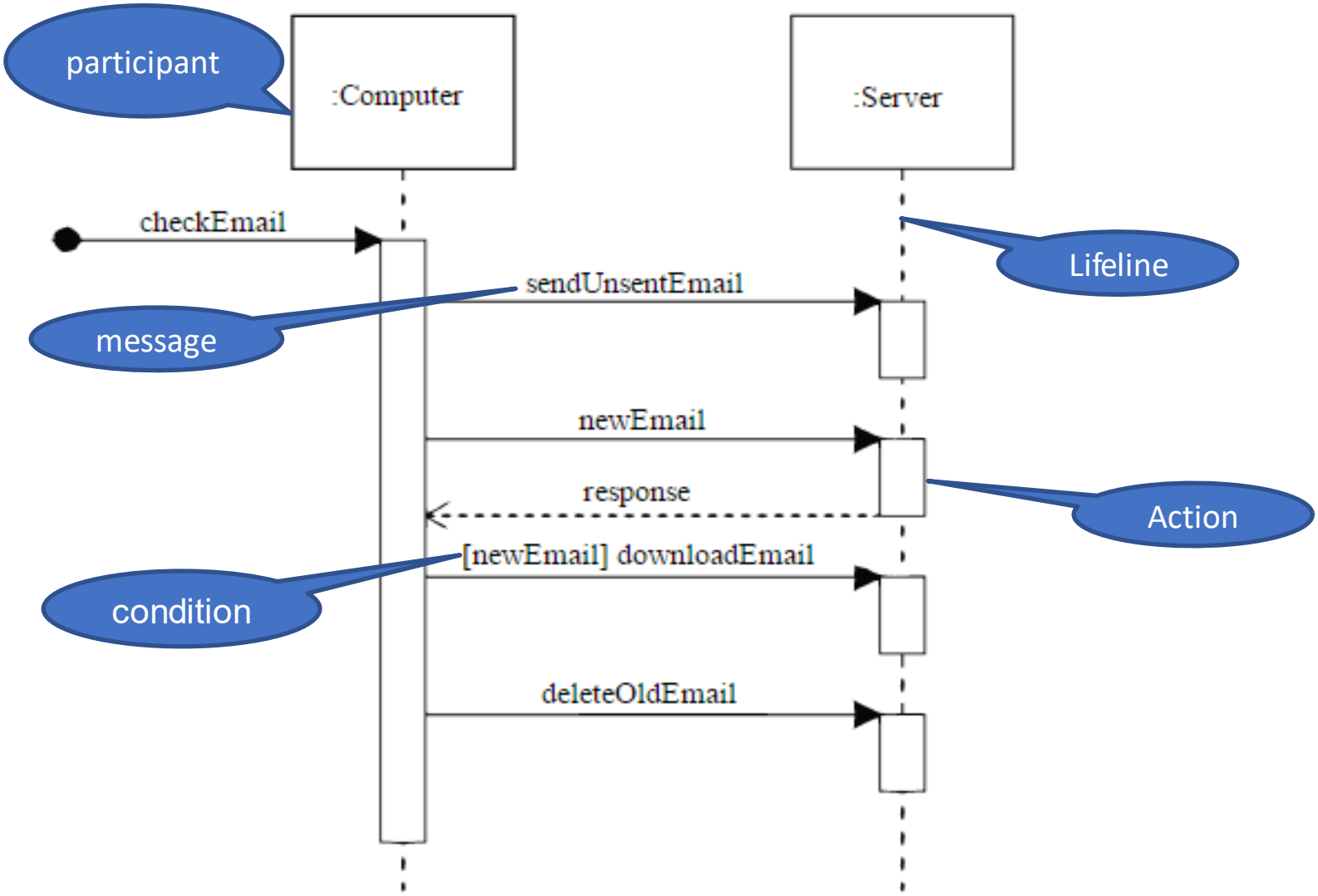
Sequence Diagram

- Shows the participants in an interaction and the sequence of messages among them
- Depicts in a graphical manner the interaction of a system with its actors to perform all or part of a use case
- Each object represented by a lifeline
- Time moves vertically from top to bottom
- Messages between objects (with parameters) shown by arrows
- Can be at different levels of abstraction
  - Object, Module, Subsystem, System



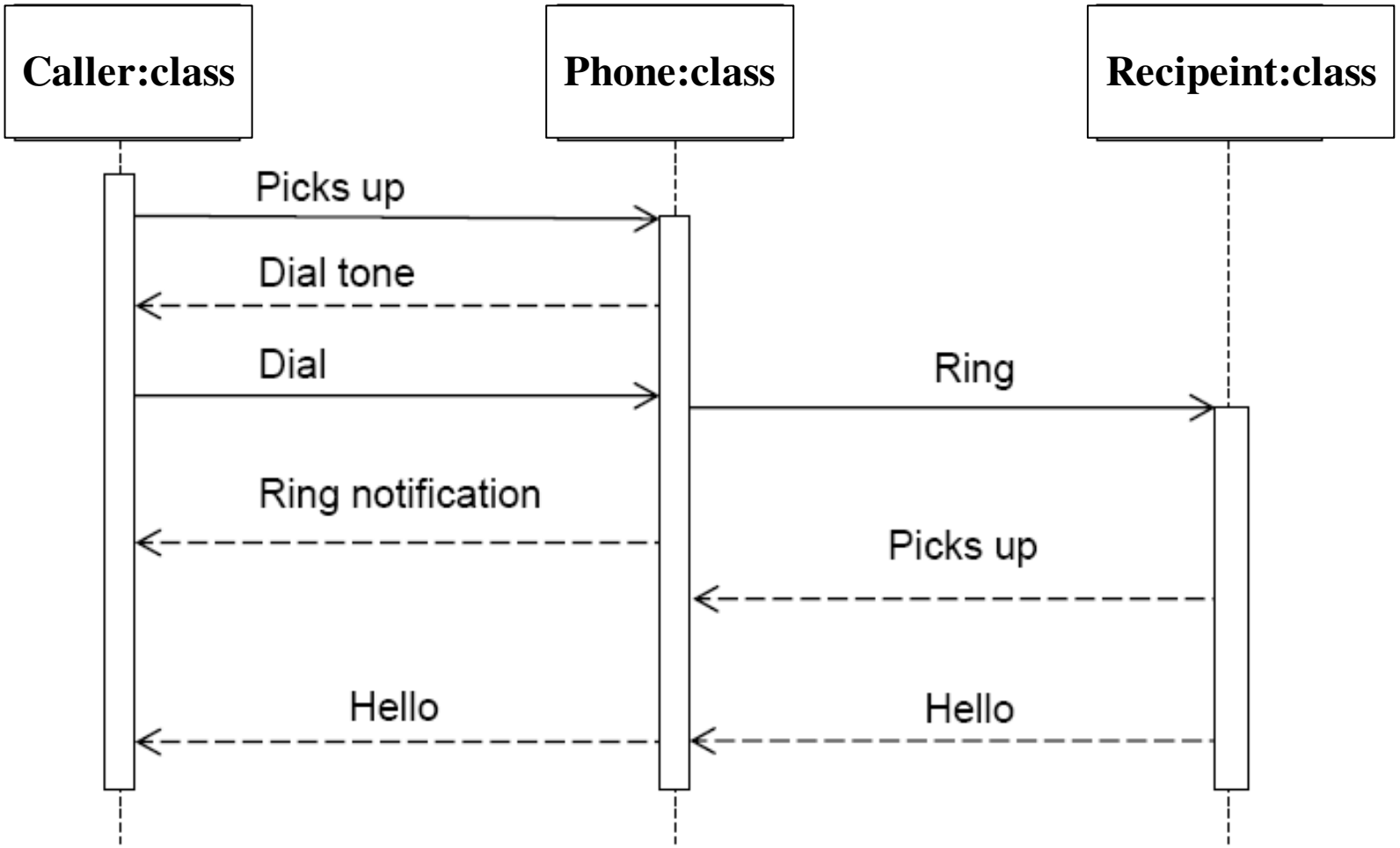
Behavioral Models for Solving a Problem

Sequence Diagram (Email Application)



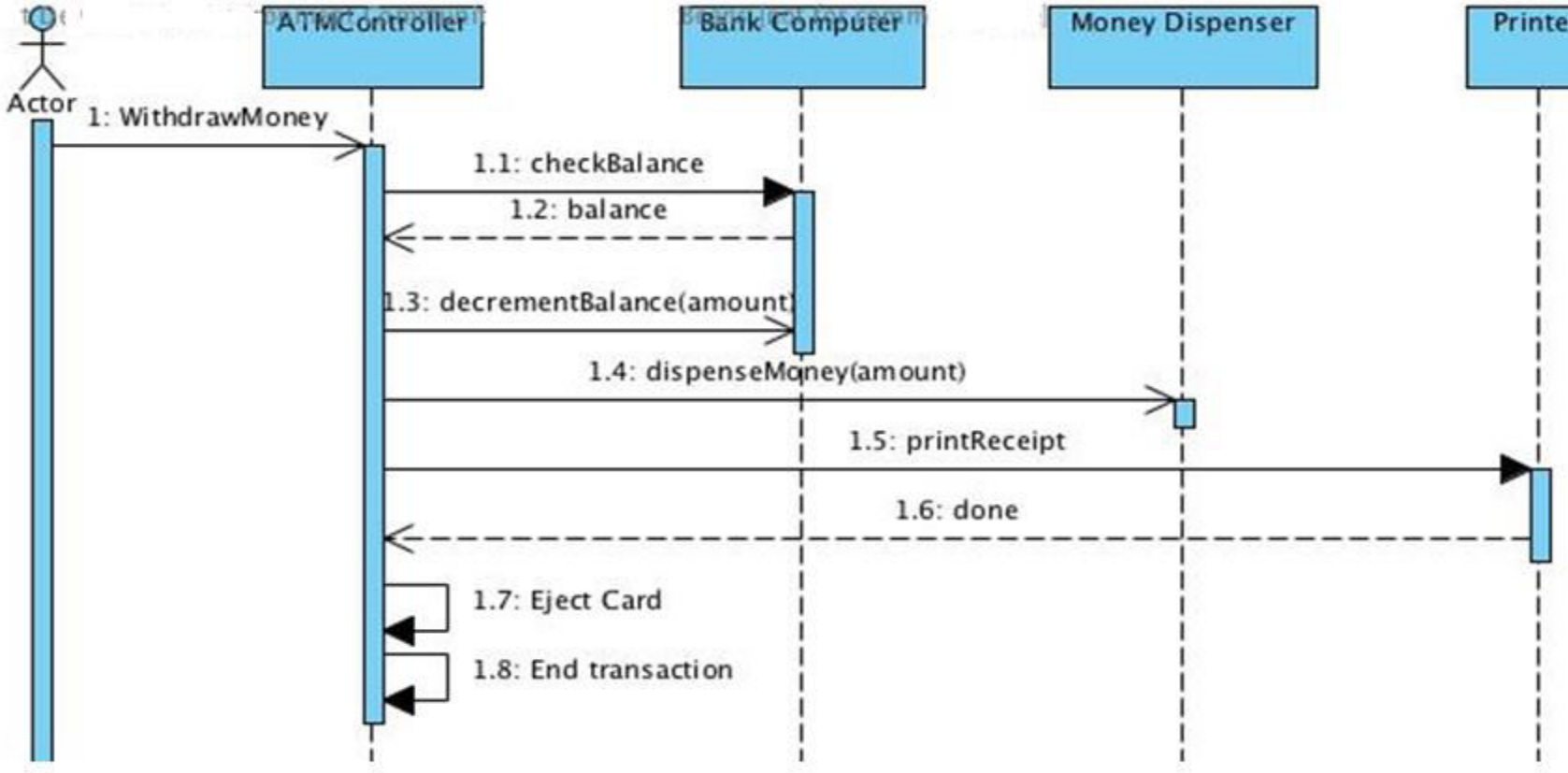
Behavioral Models for Solving a Problem

Sequence Diagram (Phone Calling Application)

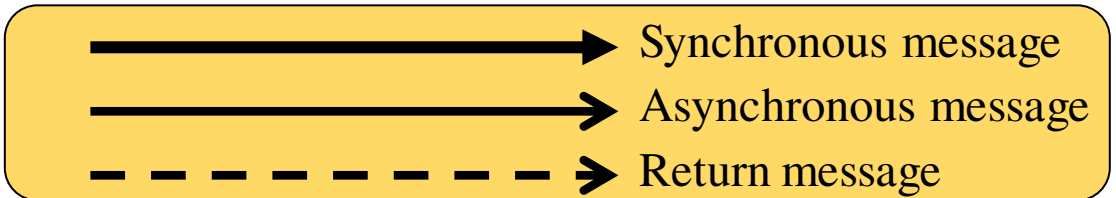


Behavioral Models for Solving a Problem

Asyn Message Example



There are problems here… what are they?

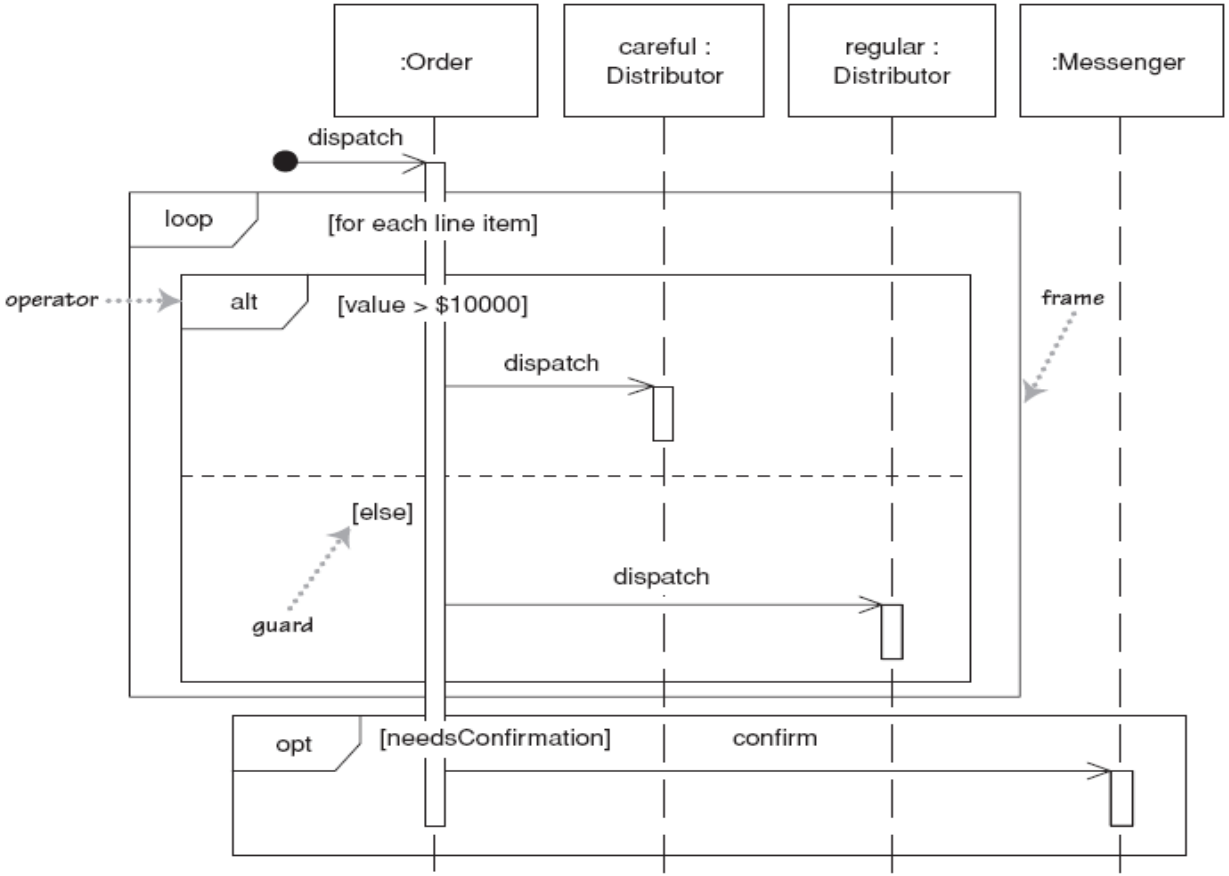


Behavioral Models for Solving a Problem

Indicating selection and loops

frame: box around part of a sequence diagram to indicate selection or loop

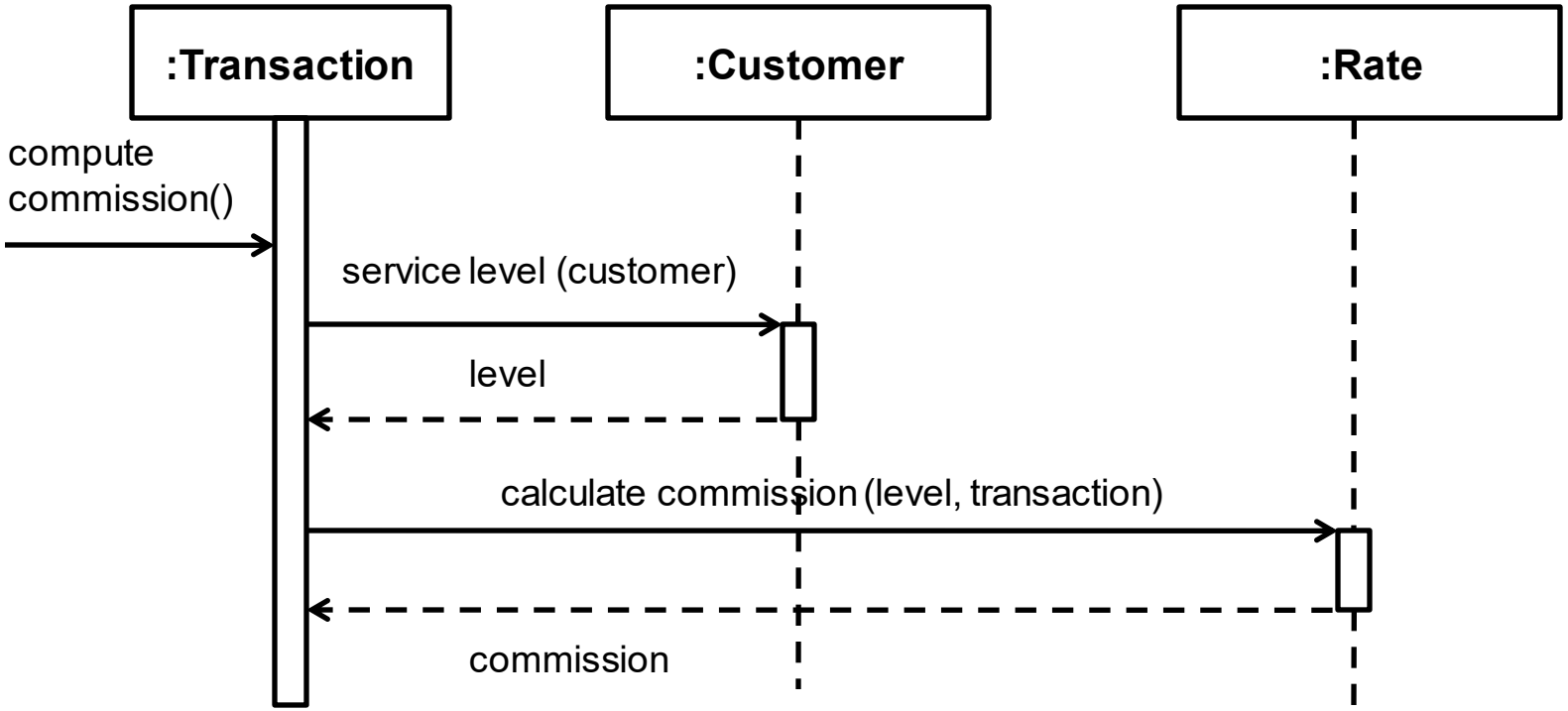
- `if` -> (opt) [condition]
- `if/else` -> (alt) [condition], separated by horizontal dashed line
- loop -> (loop) [condition or items to loop over]



Behavioral Models for Solving a Problem

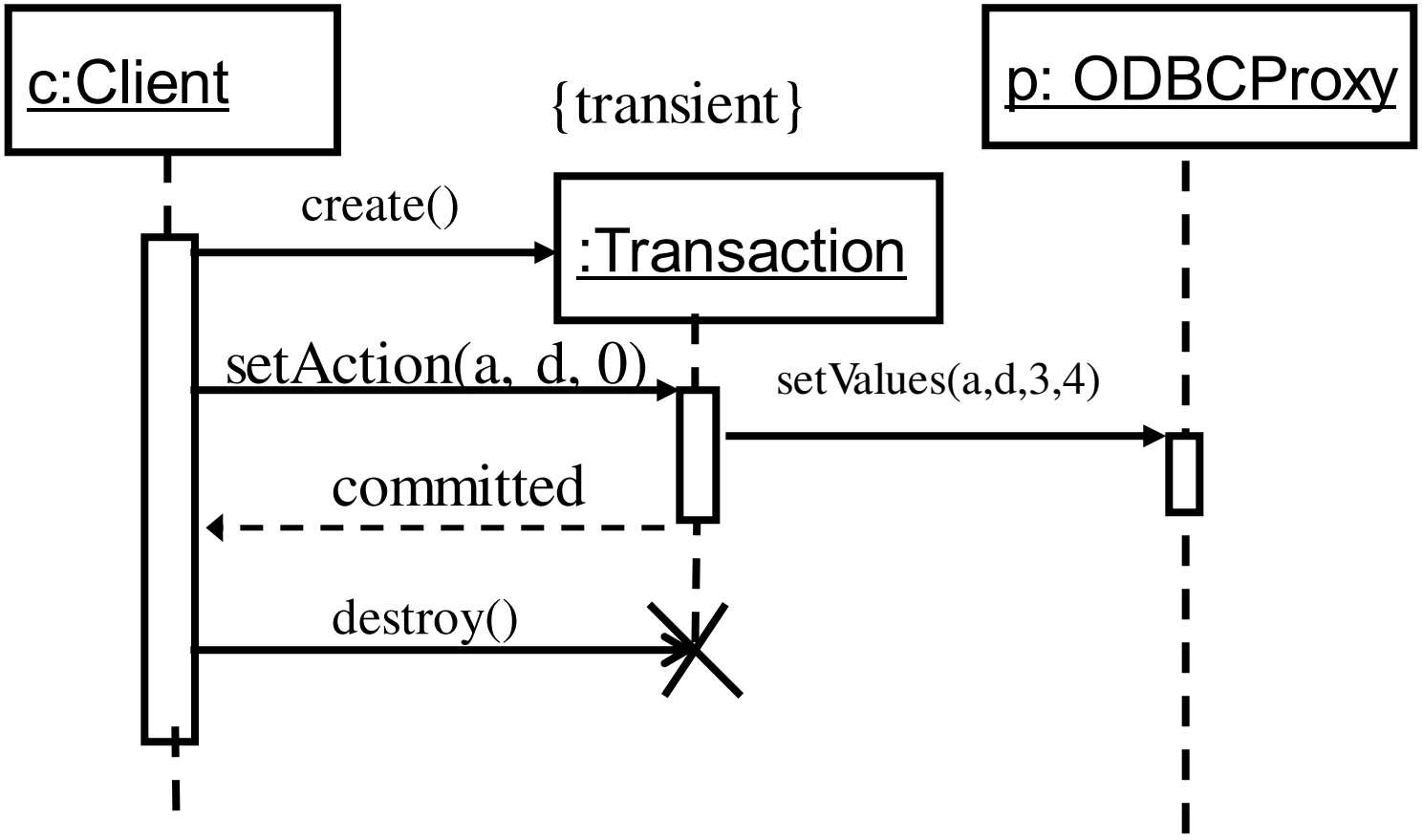
Passive Objects

- Not constantly active
- Activated only when called or cannot begin an operation
- Passive objects can also be active parallelly
- On return of control, becomes inactive
- Period of time for objects execution shown as a thin rectangle



Behavioral Models for Solving a Problem

Transient Objects



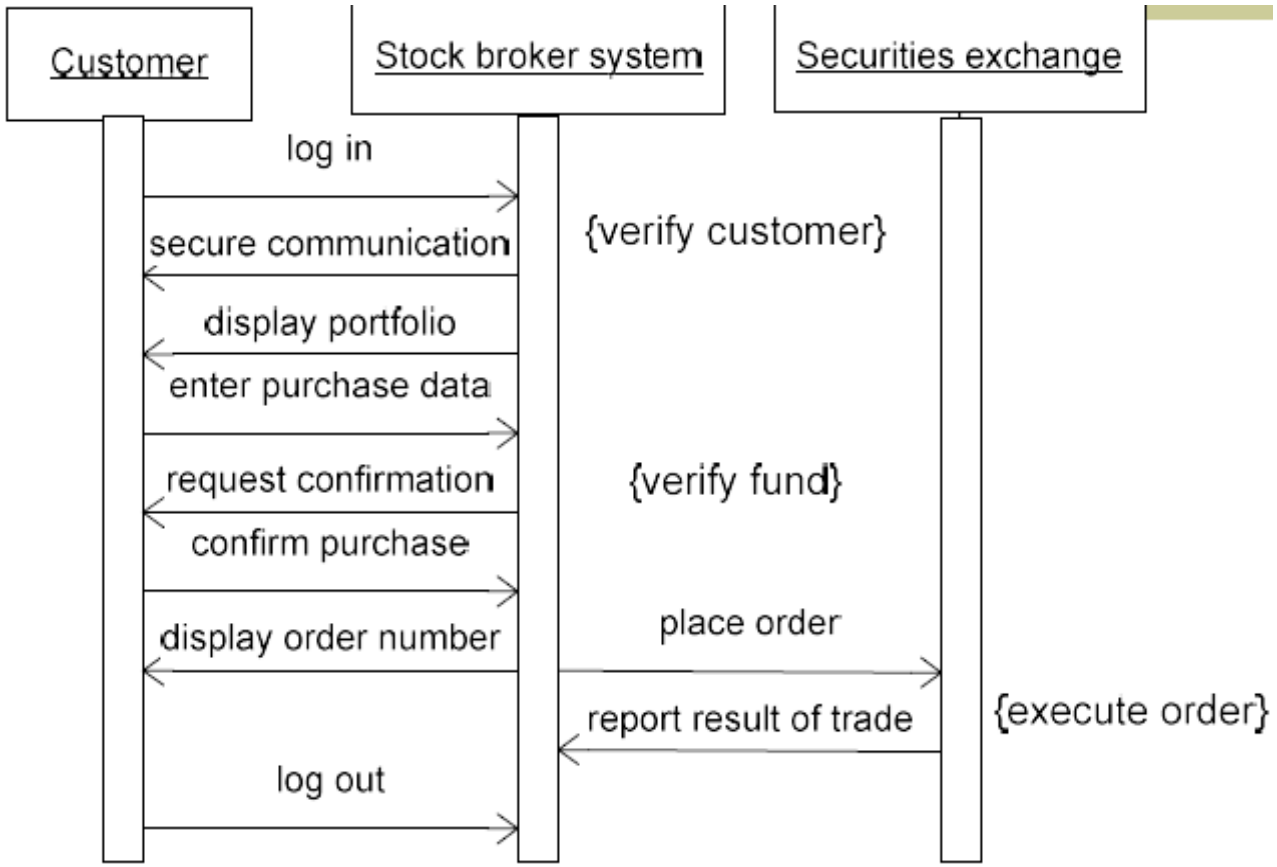
Rules of thumb

- Rarely use options,loops,alt/else
  - These constructs complicate a diagram and make them hard to read/interpret.
  - Frequently it is better to create multiple simple diagrams
- Create sequence diagrams for use cases when it helps clarify and visualize a complex flow
- Remember: the goal of UML is communication and understanding

Behavioral Models for Solving a Problem

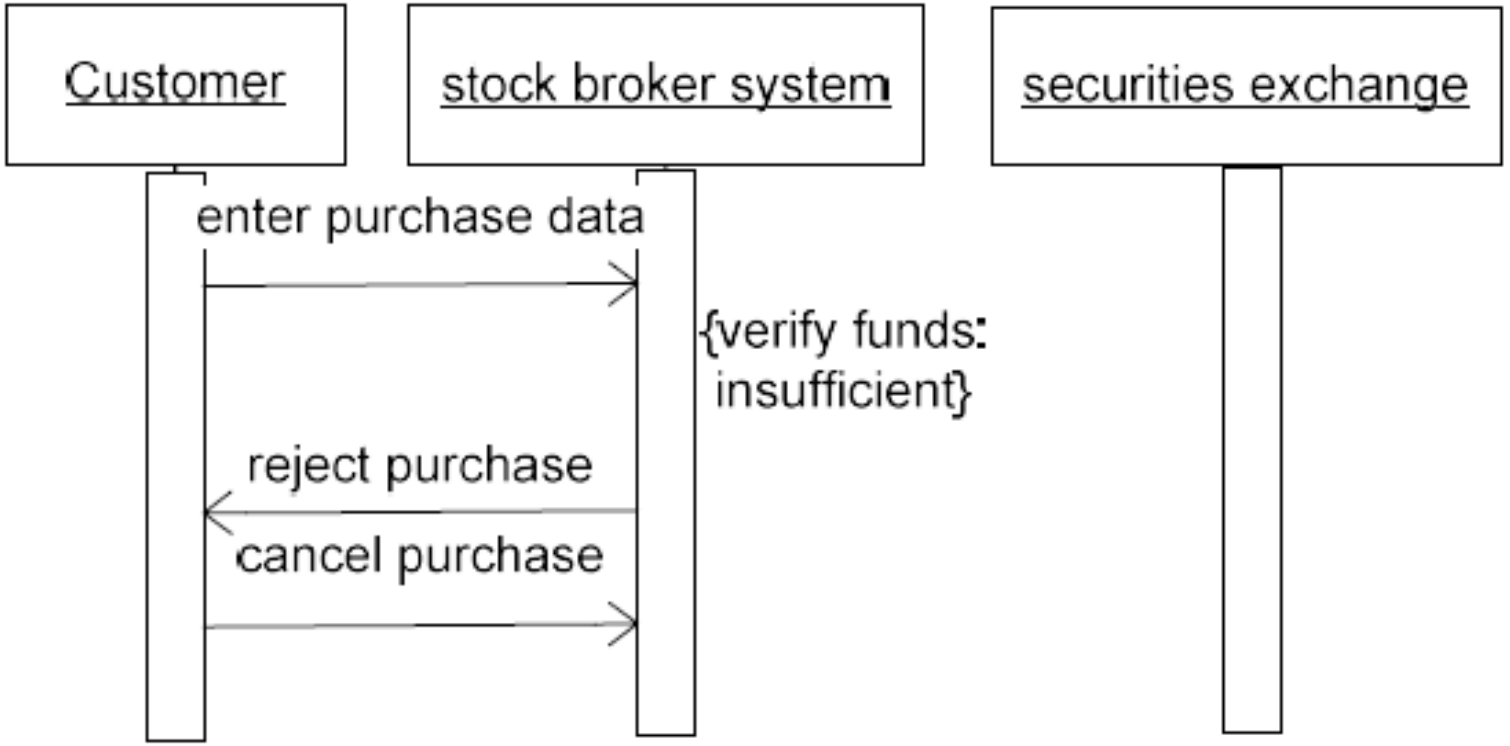
Example

Sequence diagram for a session with an online stock broker



Behavioral Models for Solving a Problem

Sequence diagram for a stock purchase that fails



Behavioral Models for Solving a Problem

Exercise

- Prepare sequence diagram for
  - Withdraw funds (retail banking)
  - Deposit funds (retail banking)
  - Transfer funds (internet banking)
    - For enhanced security, the payee must be added to a payee list
    - Procedure for adding a payee
      - User enters name, account number, and bank code (if different bank) of payee
      - System generates an authorization code and sends it to user's email and cell phone
      - User enter authorization code to add payee

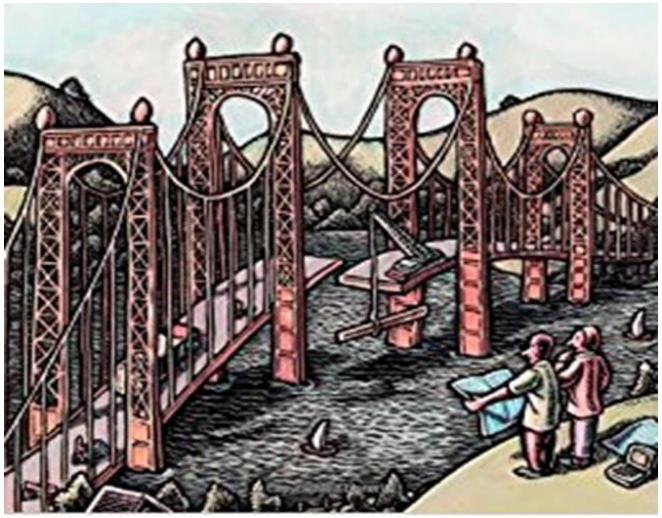
THANK YOU

Dr. H. L. Phalachandra
Department of Computer Science and Engineering
phalachandra@pes.edu

OOAD and SE

Behavioral Models

(State Diagram)



Dr. H.L. Phalachandra
Department of Computer Science and Engineering
leveraging some information from slides of
Prof. Vinay Joshi

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet and supplemented by my experience. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

Behavioral Models for Solving a Problem

Recap

- We discussed on defining the context of system using both static and dynamic components.
  - The static components we looked the blocks of functionality which we wanted to implement, identified the objects derived the classes at the grass-root level, understood how to represent/abstract them into objects/classes factoring in their attributes, responsibilities and operations which it would need to support and the relationships between these classes as part of a class diagram.
  - We then modelled these into a hierarchy of components (like modules and submodules) using the Component diagram, we discussed the individual components and how they could be modelled (represented along with their interfaces).
  - We then discussed on how these components can be related physically by associating each of the components with physical nodes where these components deployable as packages with respective artifacts, would be hosted for execution as part of Deployment diagrams.

Behavioral Models for Solving a Problem

Recap

- We then discussed on we could bring in the interactions between the classes/objects and components into a set of dynamic or behavioral models.
- There were two basic models which we discussed for bringing in the dynamic behavior or interactions of the classes to produce useful results.
  - Activity Model
    - We modelled and represented the workflow of the as activities made up of atomic actions sequentially and concurrently factoring in essential dependencies. We used concepts of swimlanes to partition to group them together.
  - Sequence Models
    - We modelled and represented sequence of events that would occur during a particular a usecase's execution. This would include all the communication in terms of messages which are exchanged between the objects participating in the execution of an action (which is part of an activity)

Behavioral Models for Solving a Problem

Generics of State Diagram

- We now discuss the dynamic behavior of objects over time by modelling the lifecycles of objects of each class over the objects life span represented as the **state diagram**
- In the state diagram we consider
  - Each object treated as an isolated entity that communicates with the rest of the world by detecting events and responding to them.
  - Events representing the different changes that objects can detect... anything that can affect an object can be characterized as an event
- The state of an object is a condition or a situation during the life of an object during which it satisfies some condition, performs some activity or waits for an event
- When an event occurs some activity will take place based on the current state of the object
- Activity which is an ongoing non atomic execution within the state machine, results in some action which could be made of some atomic computation that results in the change in the state of the model or a return of a value

Behavioral Models for Solving a Problem

State Diagram

State diagram can be visualized as representation of potential states of the objects and the transitions among the those states

Few points which are inherently true for these would be

- An object must be in some specific state at any given time during its lifecycle.
- An object transitions from one state to another as a result of some event that affects it.
- There can be only one start state in a state diagram, but there may be many intermediate and final states

We take the approach of looking at the different terms and concepts which are part of this and look at state diagrams

Behavioral Models for Solving a Problem

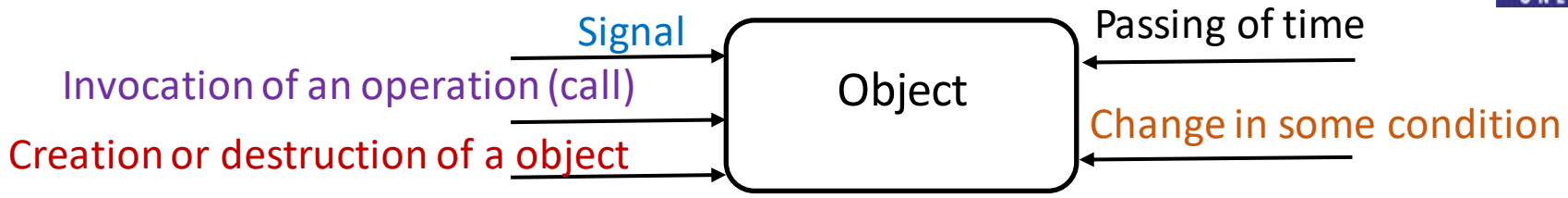
State Diagram UML Symbols

Term and Definition	Symbol
A state Is shown as a rectangle with rounded corners Has a name that represents the state of an object	
An initial state Is shown as a small filled-in circle Represents the point at which an object begins to exist	
A final state Is shown as a circle surrounding a small solid filled-in circle (bull's-eye) Represents the completion of activity	
An event Is a noteworthy occurrence that triggers a change in state Can be a designated condition becoming true, the receipt of an explicit signal from one object to another, or the passage of a designated period of time Is used to label a transition	Event name
A transition Indicates that an object in the first state will enter the second state Is triggered by the occurrence of the event labeling the transition Is shown as a solid arrow from one state to another, labeled by the event name	

Behavioral Models for Solving a Problem

State Diagram Terms and Concepts : Events

- Object instance could be exposed to different events like



- An Event is an occurrence at a given point in time. It often corresponds to a verb in past sense or on the onset of a condition other E.g. User depresses a button, customer reports a problem, shipment arrives, phone taken off hook, paper try becomes empty, temperature falls below zero
- A event represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another
- Located in time and space, considered as Instantaneous occurrence, Duration unimportant – the fact that it has occurred is important
- These events could be
  - Normal as well as error conditions
  - External or internal (e.g. page loaded)
- Concurrent and Related Events
  - Related events if one event logically or causally follows another
  - If the events are causally unrelated they are concurrent events

Behavioral Models for Solving a Problem

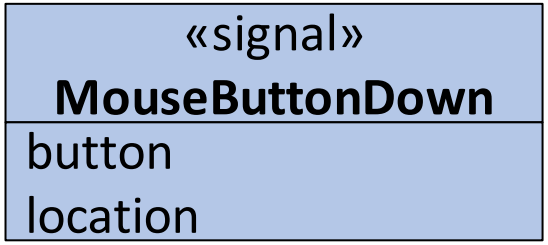
State Diagram Terms and Concepts : Events types – Signal Event

- Signal – One way transmission of information/message from one object
  to another
- An object sending a signal to another object may expect a reply, but the reply is a separate signal under the control of the second object, which may or may not choose to send it.
- ***Signal Event*** is the event of sending or receiving a signal
- Contrasting signal and signal event, signal is a message between objects, whereas the signal event is an occurrence in time

Behavioral Models for Solving a Problem

State Diagram Terms and Concepts : Events types – Signal

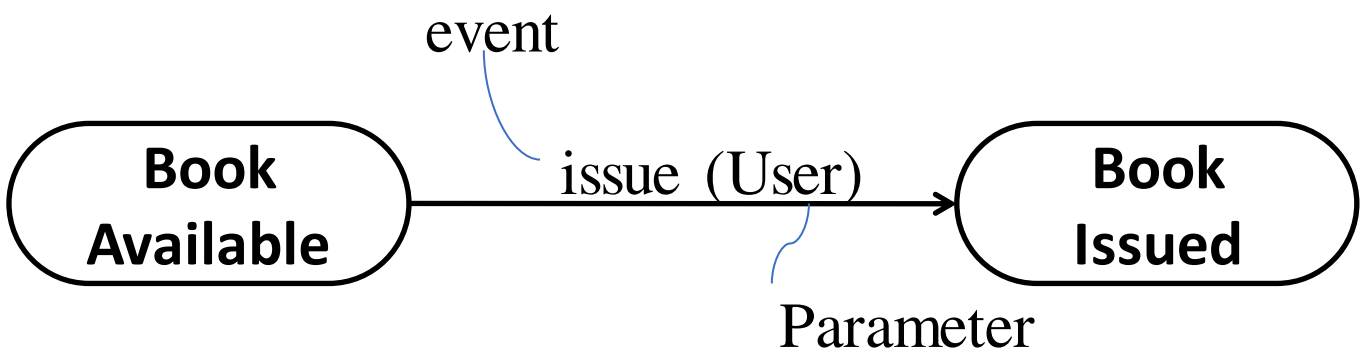
- Signal occurrences are unique, but are grouped to ***signal classes*** with a class name to indicate common structure and behavior. They also have attributes indicating values they convey
- They are represented in UML as <<signal>> on top of class name



Behavioral Models for Solving a Problem

State Diagram Terms and Concepts : Events types – Call Event

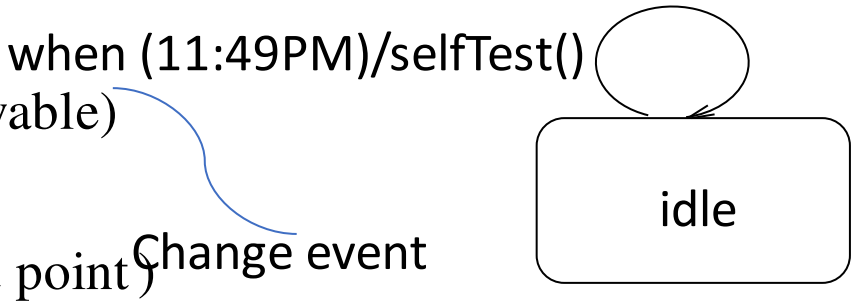
- Results in a dispatch of an operation (contrast to signal which represents the occurrence of a signal)
- Receipt by an object of a request for an operation
  - Operation may change state of receiver
  - Call event handled by operation (method)
- Generally synchronous
  - Control transfer to receiver, Caller blocked till operation completed



Behavioral Models for Solving a Problem

State Diagram Terms and Concepts : Events types – Change Event

- Caused by satisfaction of Boolean expression
- Modeled using
  - when (expression)
- Expression below is continuously (practically) tested & whenever the expression evaluates to true the event happens
- Examples
  - when (inventory < reorder point)
  - when (storage used > maximum allowable)
  - when (altitude < 10000)
  - when (room temperature < heating set point)
  - when (room temperature > cooling set point)
  - when (battery power < lower limit)
  - when (tire pressure < minimum pressure)
- Generally detected by polling or other application logic

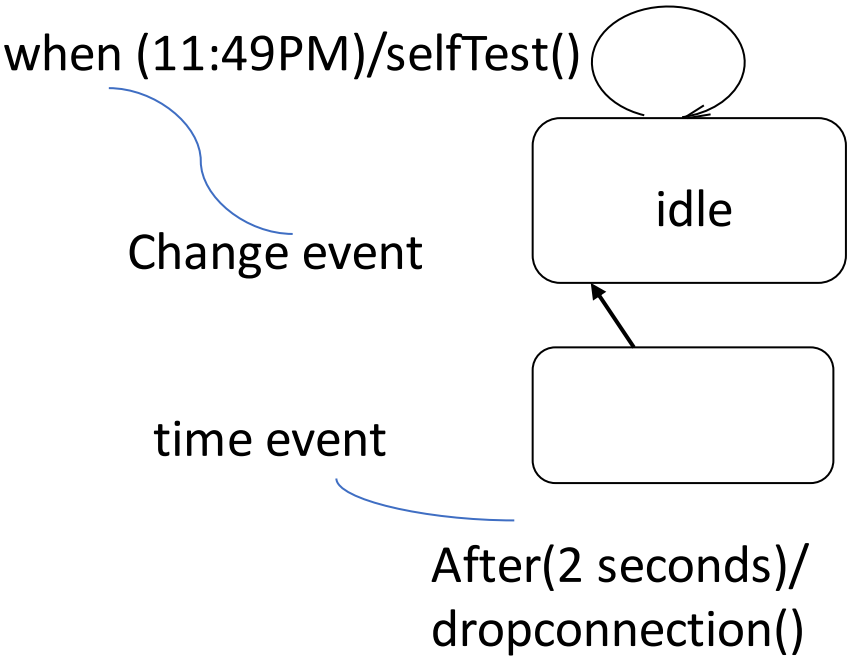


Behavioral Models for Solving a Problem

State Diagram Terms and Concepts : Events types – Time Event

- Represents elapse of time or absolute time
- Modeled using
  - when (expression)
  - after (expression)
- UML notation for an absolute time is the keyword when followed by a parenthesized expression involving time.

Eg:
-- when (date = November 1, 2011)
-- after (10 seconds)



Behavioral Models for Solving a Problem

State Diagram Terms and Concepts : State

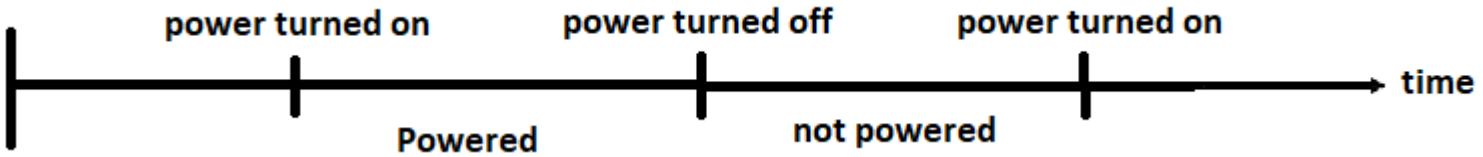
- State is an abstraction of values and links of an object (some internal condition). Sets of values and links are grouped together into a state according to the gross behaviour of objects.

State is represented as

- States correspond to verbs with a suffix of “ing” (Waiting, Dialing) or the duration of some condition (Powered, BelowFreezing)

Idle

- Objects of a class have a finite number of possible states. Each object can be in one state at a time.
- State determines response of the object to input events. All inputs other than ones which are defined for are ignored
- State of the object depends on the past events which in most cases are eventually hidden by subsequent events
- Events corresponds to a specific points in time and states represent intervals of time between two events received by the object



Behavioral Models for Solving a Problem

State Diagram Terms and Concepts : Characterizing States

State Alarm ringing on a watch

- **State** : *Alarm Ringing*
- **Description** : alarm on watch is ringing to indicate target time
- **Event sequence that produces the state** :

  setAlarm (targetTime)

  any sequence not including *clearAlarm*

  when (*currentTime = targetTime*)
- **Condition that characterrizes the state:**

  alarm = on, alarm set to *targetTime*,

  targetTime <= currentTime <=targetTime+20 sec , and no button has been pushed since *targetTime*
- **Events accepted in the state:**

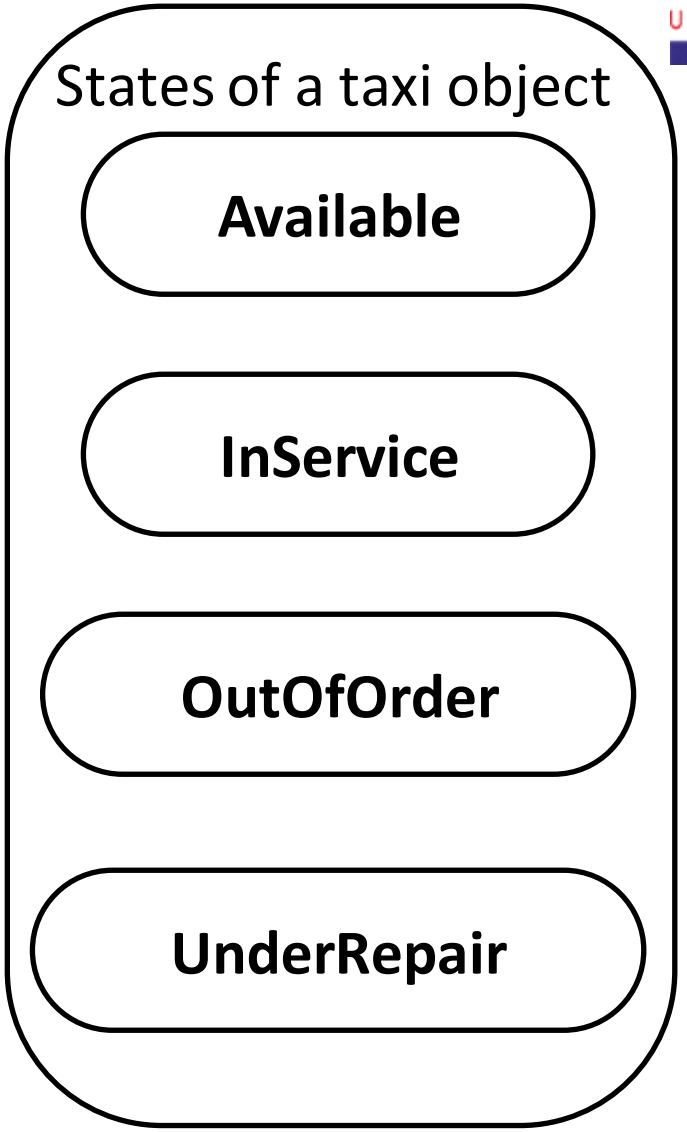
event	response	next state
when (*currentTime =targetTime+20*)	*resetAlarm*	*normal*
buttonPushed(any button)	*resetAlarm*	*normal*

Behavioral Models for Solving a Problem

State Diagram Terms and Concepts : Characterizing States

States of a Taxi Object for Example

- Name
- Description
- Entry/Exit effects
  - Actions executed on entering and exiting the state
- Events accepted, not accepted, deferred for a state
  - Apply it to the example states here
- Sub-states
  - Apply it to the example states here



Behavioral Models for Solving a Problem

State Diagram Terms and Concepts : Transition and Conditions

Transition: an instantaneous change in state from one state to another

- triggered by an event
- Transition is said to fire upon the change from source to target state
- Original and target state of a transition depends on the original state and could be different or the same.

e.g. when a phone line is answered, the phone line transitions from the ***Ringing*** state to the ***Connected*** state.

Guard Condition:

- boolean expression that must be true for transition to occur
- checked only once, at the time event occurs; transition fires if true

e.g. when you go out in the morning ***(event)***, if the temperature is below freezing ***(condition)***, then put on your gloves ***(next state)***.

Guard Condition Example – Turnstile with coin

Behavioral Models for Solving a Problem

State Diagram : Guard Condition vs Change Event

Guard condition	change event
a guard condition is checked only once	a change event is checked continuously
UML notation for a transition is a line	may include event label in italics
followed by guard condition in square brackets	from the origin state to the target state an arrowhead points to the target state.

Behavioral Models for Solving a Problem

State Diagram :

- State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions;
- A state diagram is a graph whose nodes are states and whose directed arcs are transitions between states.
- It represents
  - State sequences caused by event sequences
  - State names must be unique within the scope of a state diagram
  - Common behavior of all instances of the class

Behavioral Models for Solving a Problem

State Model :

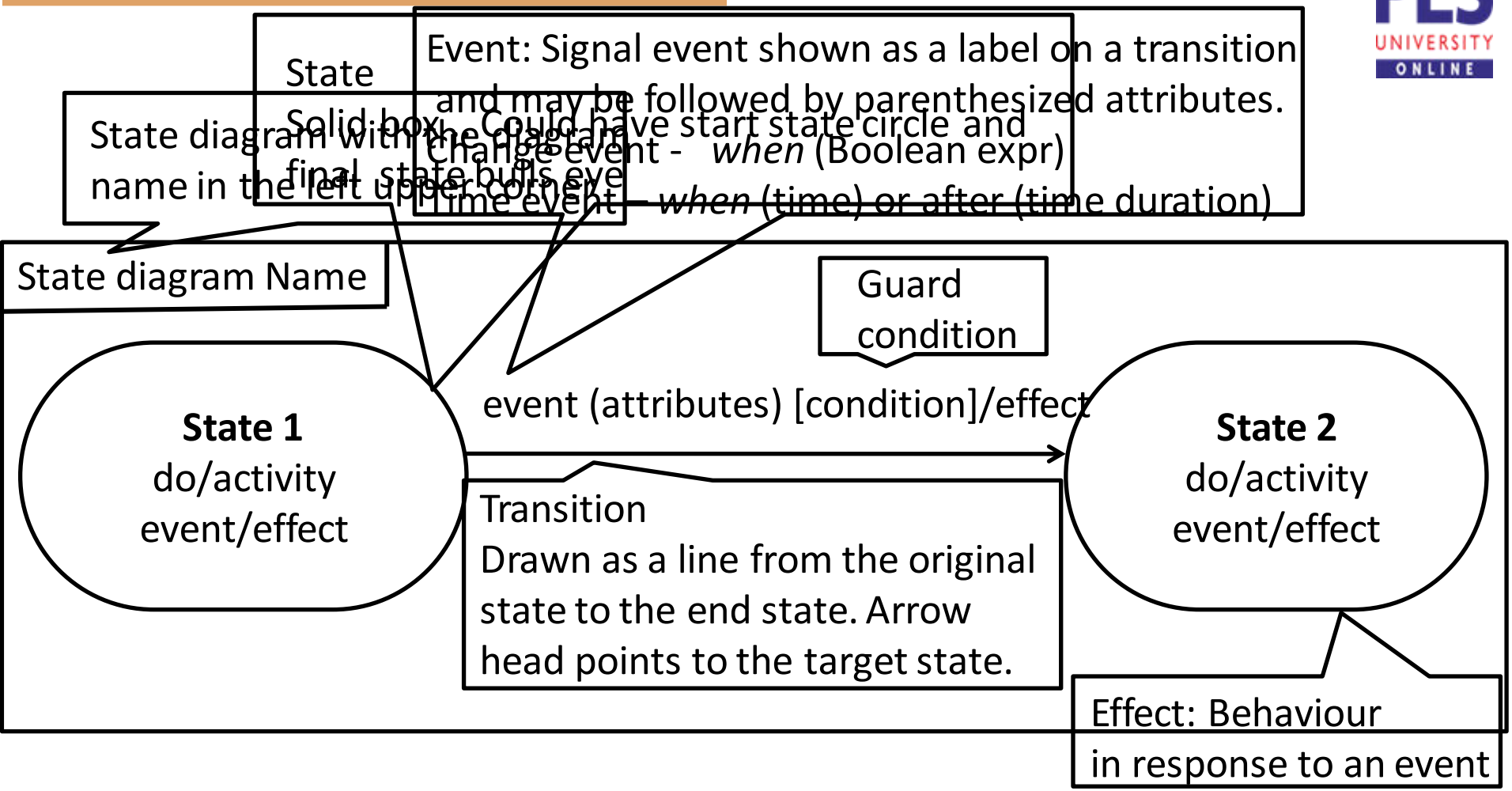
- A state model consists of multiple state diagrams one state diagram for each class with important temporal behavior. The individual state diagrams interact by passing events and through the side effects of guard conditions.

- UML notation for a state diagram is a rectangle with its name in small pentagonal tag in the upper left corner. The constituent states and transitions lie within the rectangle.

- States do not totally define all values of an object.

Behavioral Models for Solving a Problem

State Diagram :

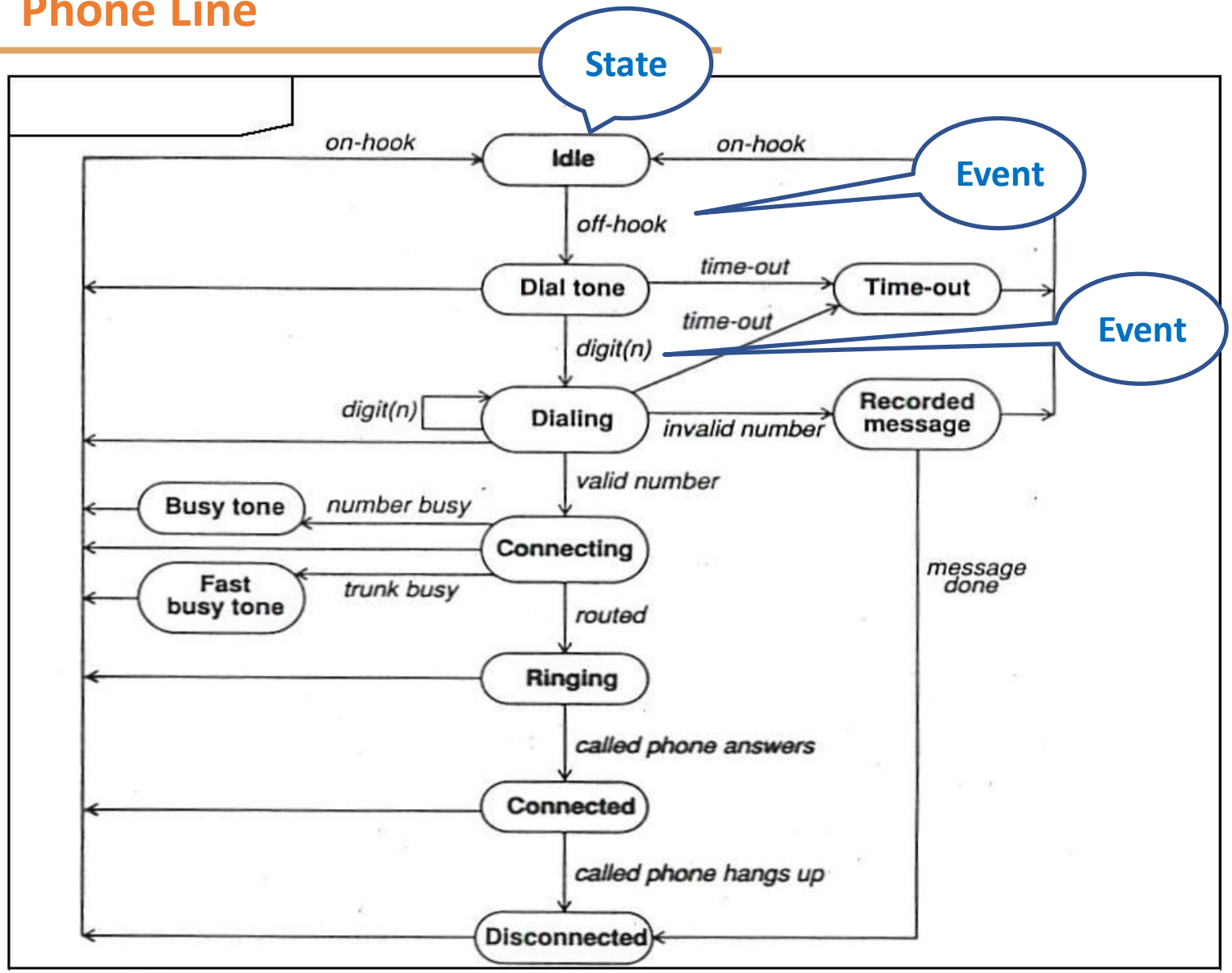


Event can be entry, exit, etc.

Do activity continues as long as the object is in that state

Behavioral Models for Solving a Problem

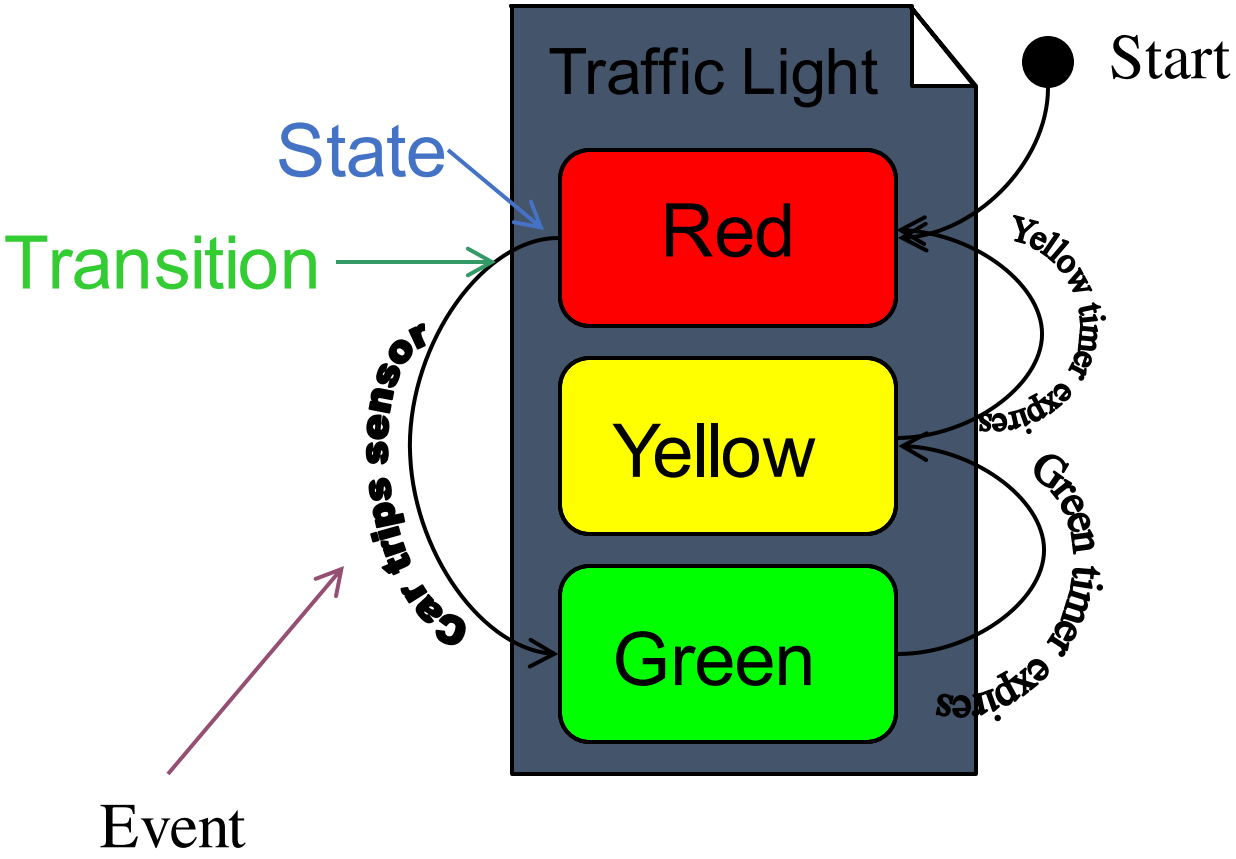
Phone Line



State diagram for phone line

Behavioral Models for Solving a Problem

State Diagrams (Traffic light example)



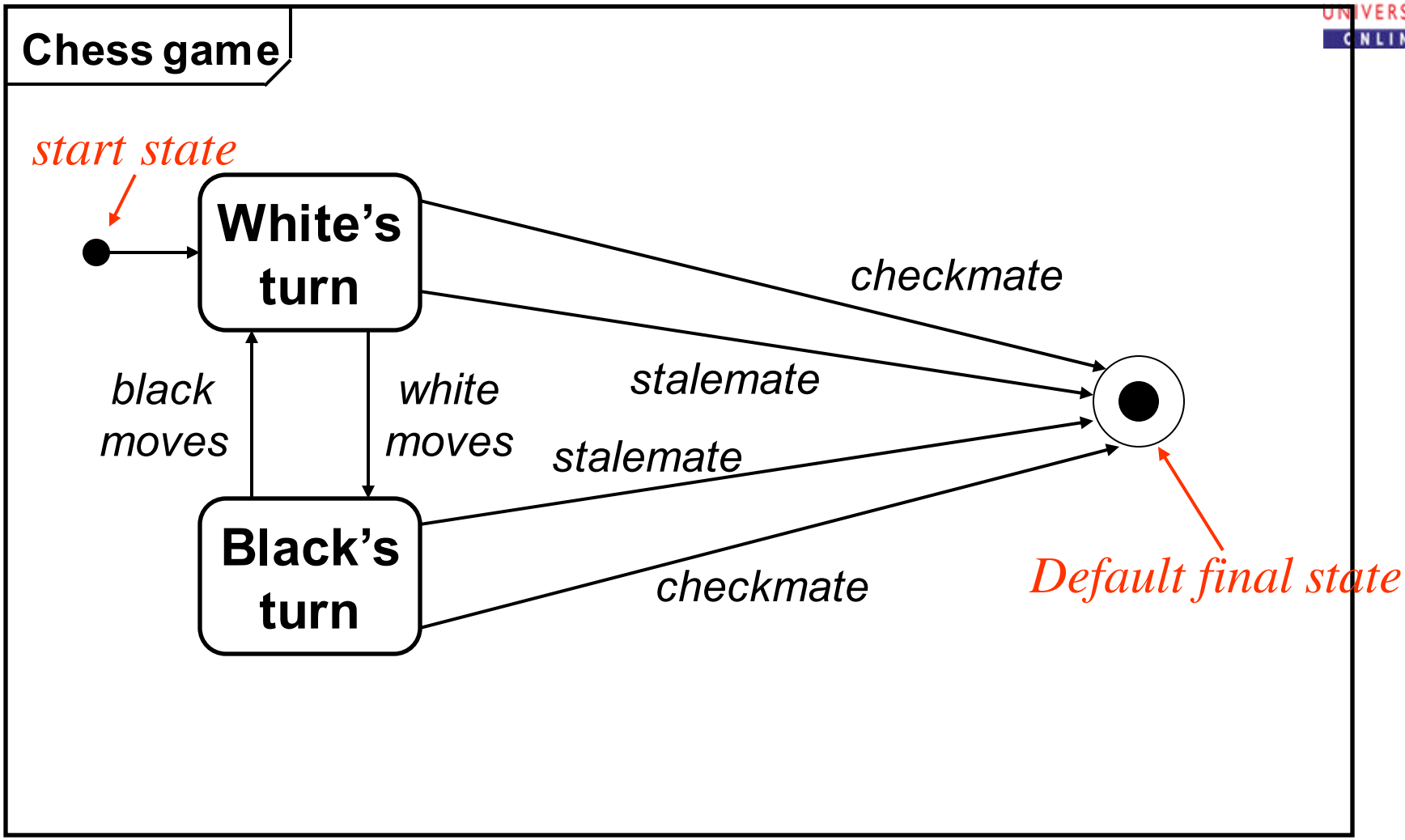
Behavioral Models for Solving a Problem

ONE SHOT STATE DIAGRAMS

- State diagrams can represent continuous loops or one-shot life cycles. Eg: Diagram for the phone line is a continuous loop
- One – shot state diagrams represent objects with finite levels and have initial and final states. The initial state is entered on creation of an object ;entry of the final state implies destruction of the object.
- Object created in a state and change of the state deletes the object
- As an alternate notation, you can indicate initial and final states via **entry** and **exit** points. Entry points (hollow circles) and exit points (circles enclosing an "x") appear on the state diagram's perimeter and may be named.

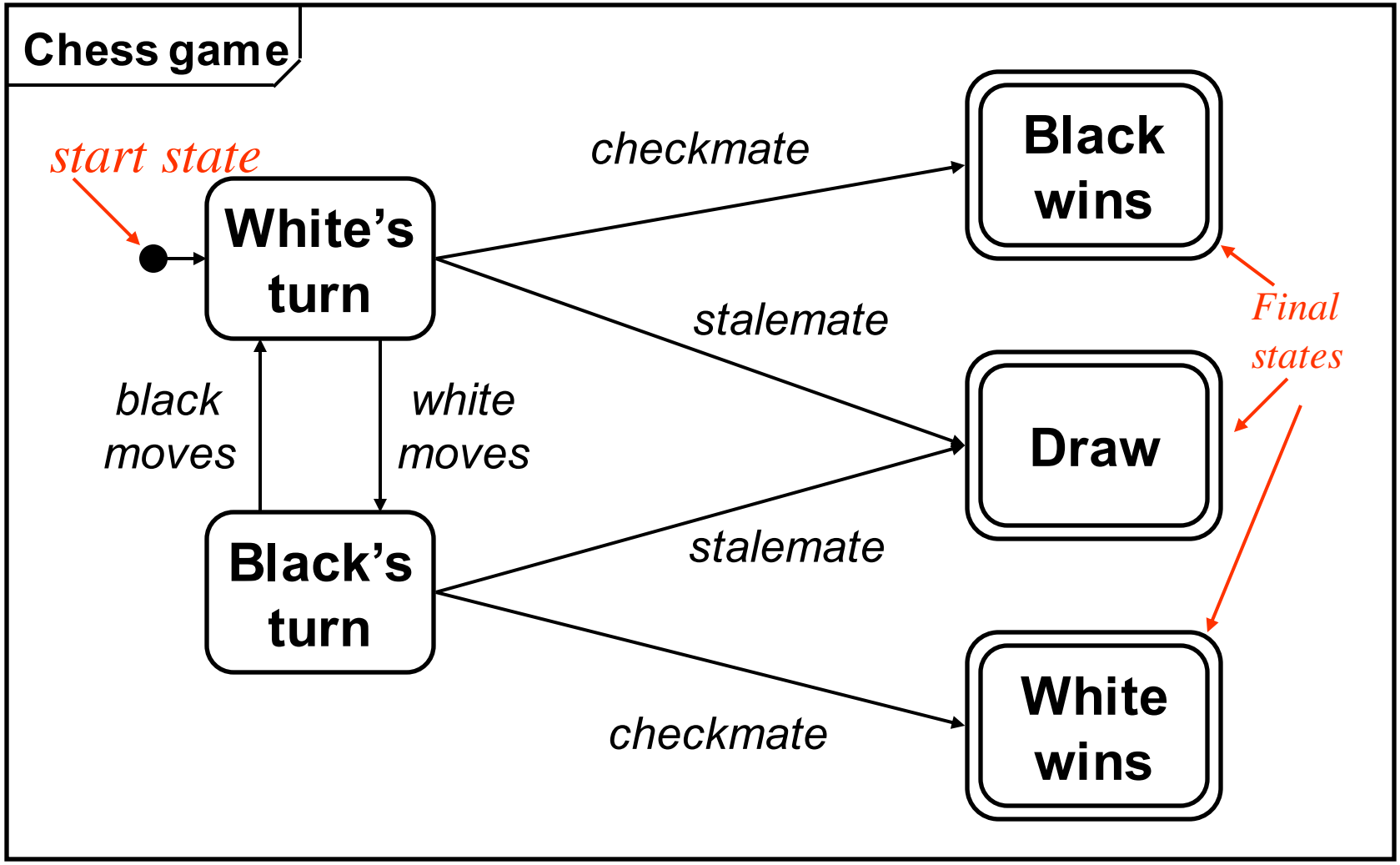
Behavioral Models for Solving a Problem

Example



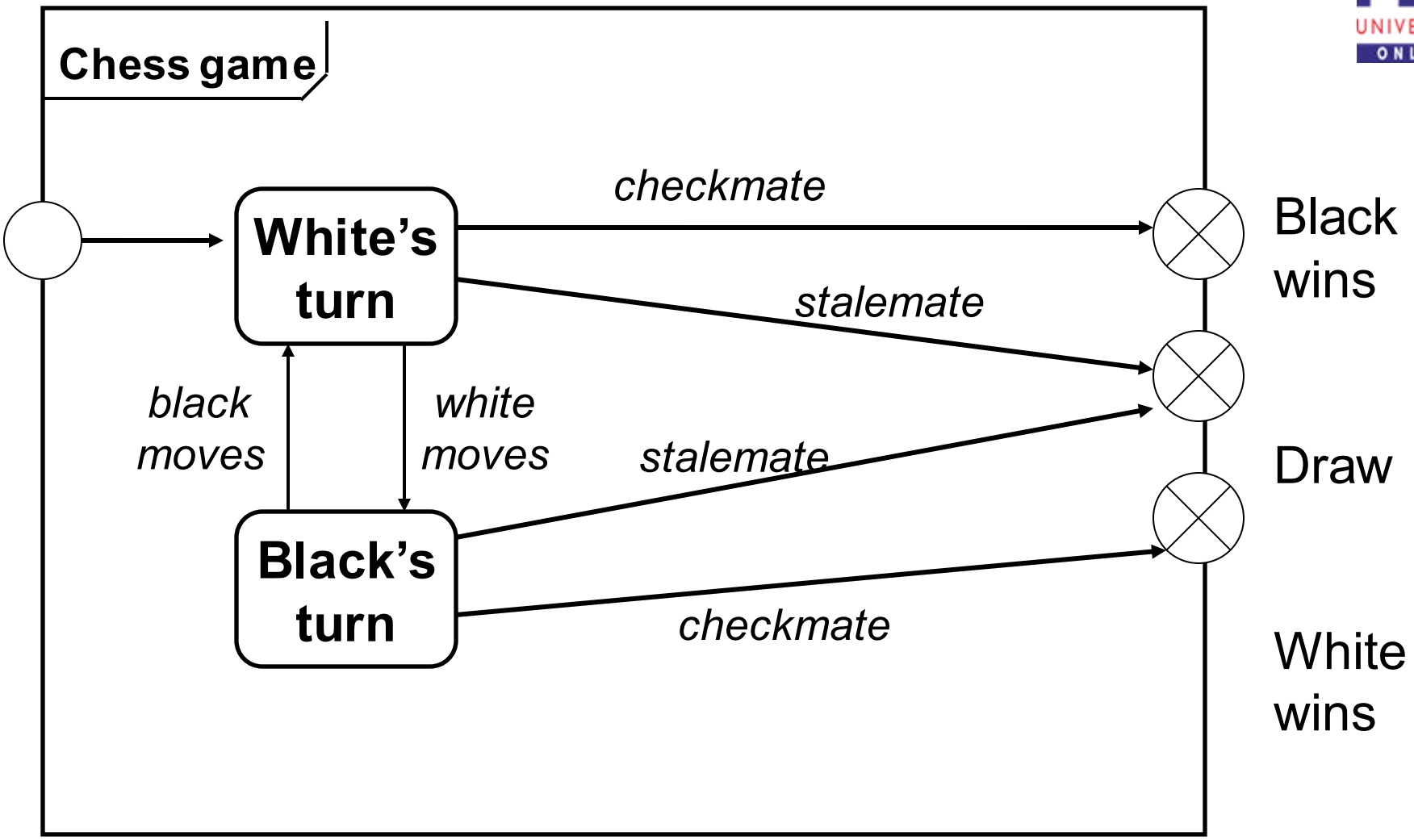
Behavioral Models for Solving a Problem

Example



Behavioral Models for Solving a Problem

Example - entry and exit points



Behavioral Models for Solving a Problem

State Diagram : Activity Effects - representing Behavior of object

- This references to the behavior of the object with state change or the activity (an action of an activity) that is invoked in response to an event. An activity is the actual behavior that can be invoked by any number of effects. Eg: disconnectPhoneLine might be an activity that is executed in response to an onHook event.
- An activity may be performed upon
  - a transition
  - the entry to or exit from a state, or
  - some other event within a state.

Behavioral Models for Solving a Problem

Activity Effects

- *Activity* = behavior that can be invoked by any number of effects
- May be performed upon:
  - a transition
  - entry to or exit from a state
  - some event within a state
- Notation:
  - *event* / resulting-activity

Behavioral Models for Solving a Problem

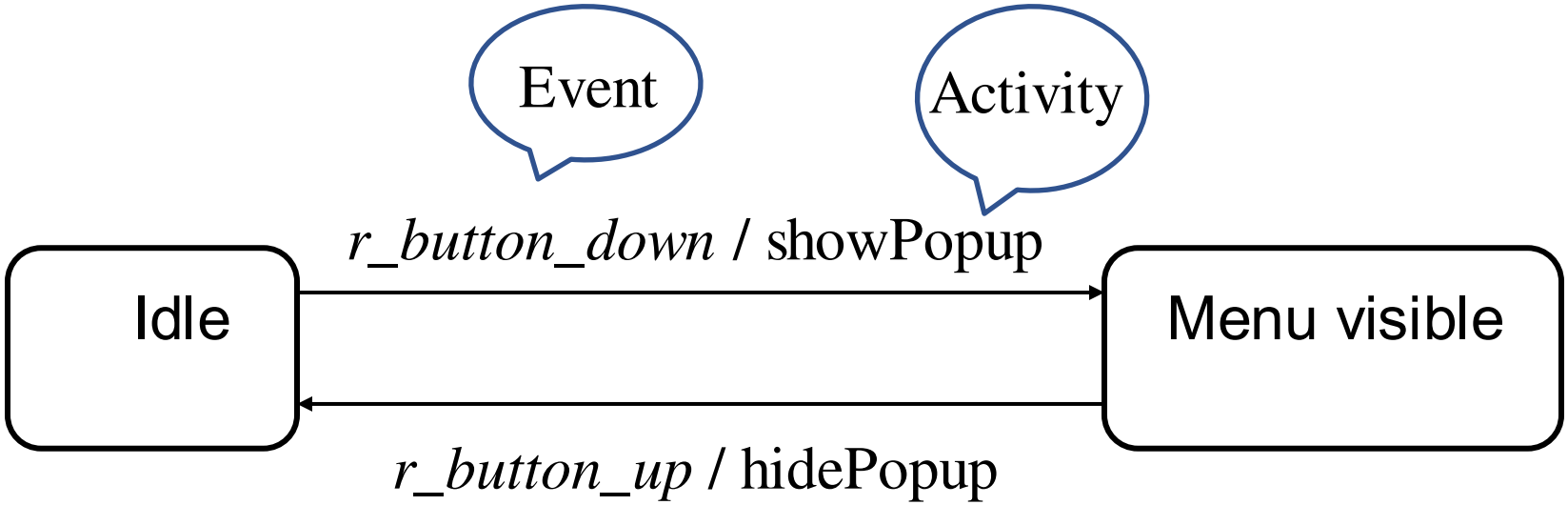
Activities

Often useful to specify an *activity* that is performed within a given state

- E.g., while in **PaperJam** state, the warning light should be flashing
- E.g., on entry into the **Opening** state, the motor should be switched on
- E.g., upon exit of the **Opening** state, the motor should be switched off

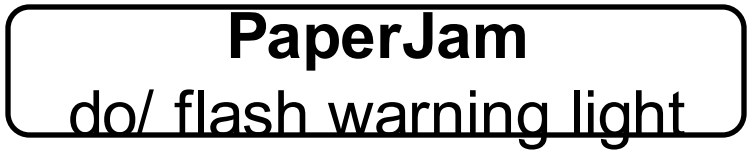
Behavioral Models for Solving a Problem

Activity effects



Behavioral Models for Solving a Problem
Do-Activities

- continue for an extended time
- can occur only within a state
- can not be attached to a transition
- include
    - continuous operations, such as displaying a picture on a television screen
    - Sequential operations that terminate by themselves after an interval of time
- may be performed for all or part of time that an object is in a state
- may be interrupted by event received during execution; event may or may not cause state transition



Behavioral Models for Solving a Problem

Entry and Exit Activities

- can bind activities to entry to/ exit from a state

- All transitions into a state perform the same activity, in which case it is more concise to attach the activity to the state

Opening
entry / motor up
exit / motor off

Behavioral Models for Solving a Problem

Order of activities

1. activities on incoming transition
2. entry activities
3. do-activities
4. exit activities
5. activities on outgoing transition

Events that cause transitions out of the state can interrupt do-activities. If a do-activity is interrupted, the exit activity is still performed

Behavioral Models for Solving a Problem

EXAMPLE

The control of a garage door opener

- The user generates depress events with a pushbutton to open and close the door.
- Each event reverses the direction of the door.
- The control generates motor up and motor down activities for the motor.

Behavioral Models for Solving a Problem

FIG: THE CONTROL OF A GARAGE DOOR OPENER

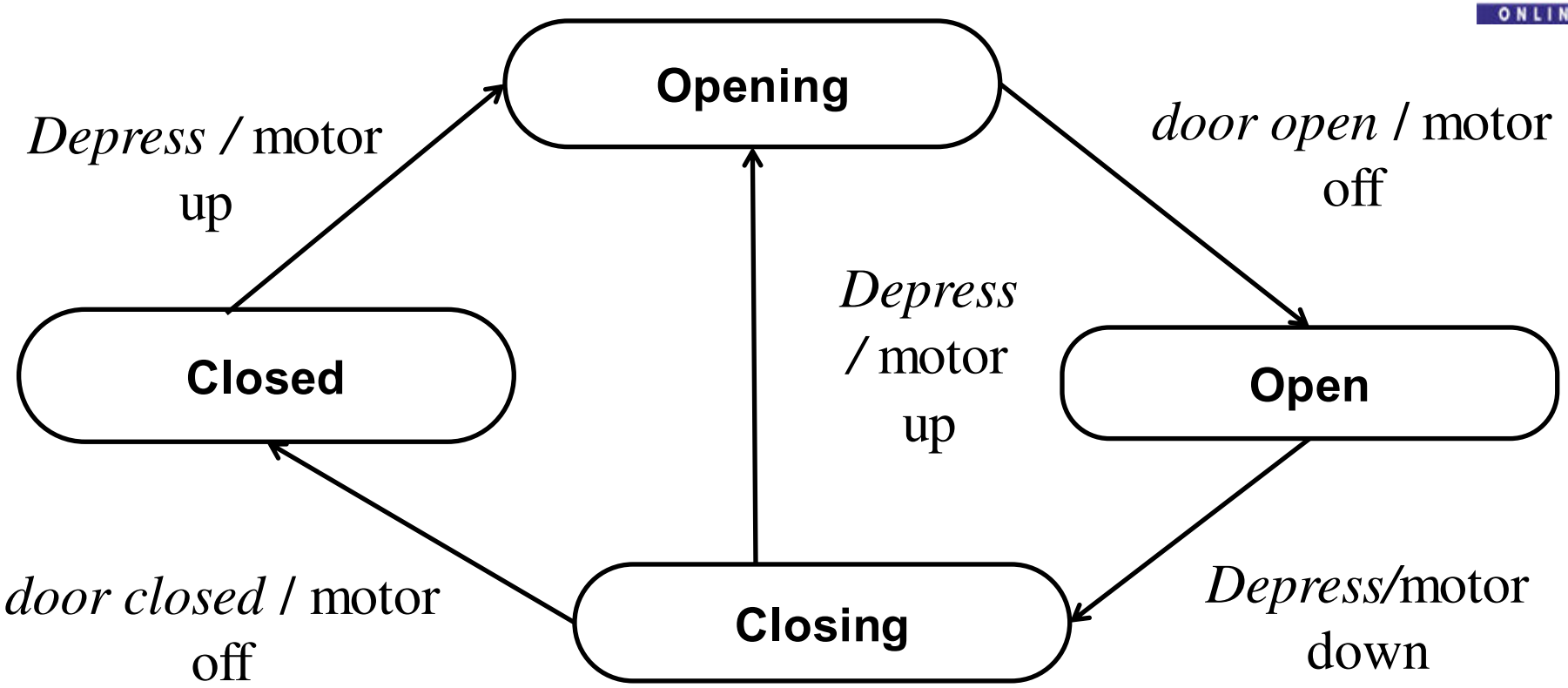


Figure: Activities on transitions. An activity may be bound to an event that causes a transition

Behavioral Models for Solving a Problem

FIG: THE CONTROL OF A GARAGE DOOR OPENER

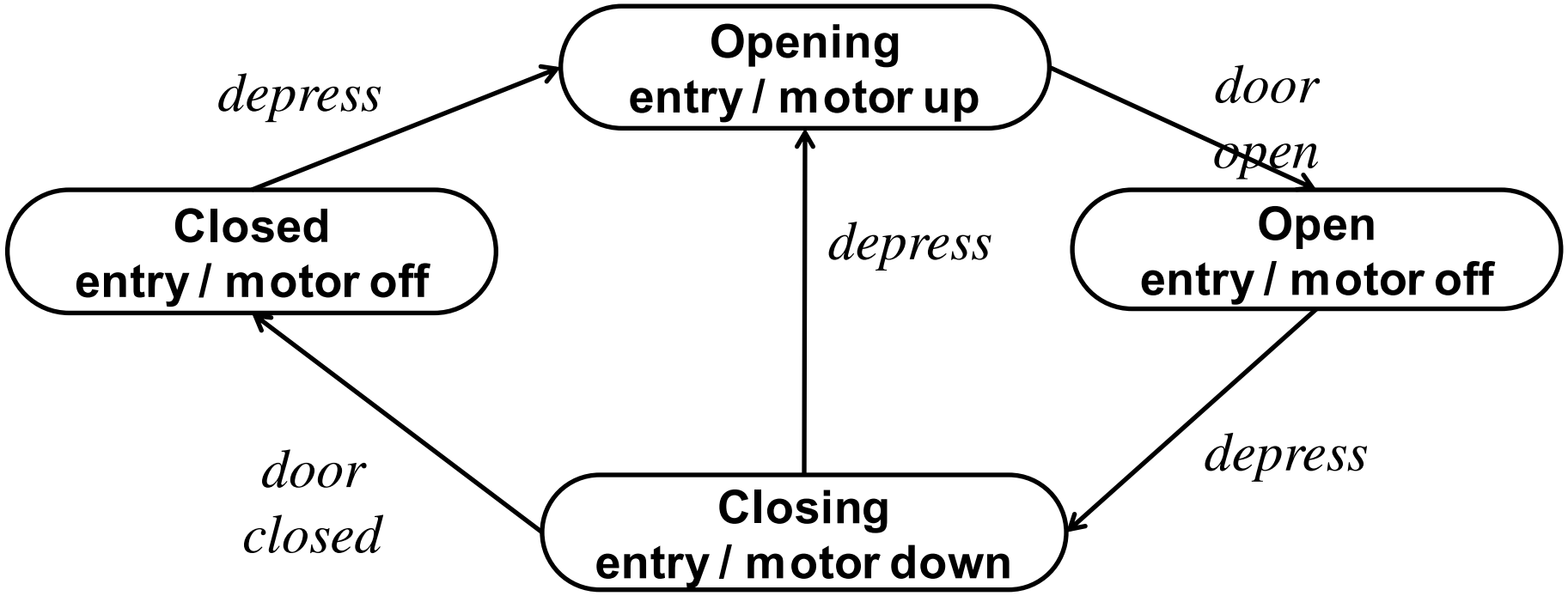


Figure: Activities on transitions. An activity may also be bound to an event that occurs within a state

THANK YOU

Dr. H. L. Phalachandra
Department of Computer Science and Engineering
phalachandra@pes.edu