Course Overview

IT 3106– Object Oriented Analysis and Design

Level II - Semester 3

Course Aim and Intended Learning Outcome

Course Aim:

To provide necessary skills and competencies to analyze and design a system using object-oriented approach

Intended Learning Outcomes:

After following this course, students should be able to

- describe Object Oriented Analysis and Design concepts and apply them to solve problems
- define UML (Unified Modeling Language) and its various types of diagrams
- prepare Object Oriented Analysis and Design documents for a given problem using Unified Modeling Language
- explain the transition from analysis to design
- describe the key activities in the component-based software engineering (CBSE) process

Required Tools

ArgoUML or

StarUML or

Visual Paradigm-Community Edition or

any tool that supports UML 1.4 and higher

References/ Reading Materials:

Ref 1: Alan Dennis, Barbara Haley, David Tegarden, Systems analysis design, An Object-Oriented Approach with UML : an object oriented approach, 5th edition, John Wiley & Sons, 2015, ISBN 978-1-118-80467-4

Ref 2: Ian Somerville, Software Engineering, 10th edition, Pearson , Pearson Education, 2016, ISBN: 978-0133943030

Ref 3: https://www.geeksforgeeks.org/computer-aided-software-engineering-case/

Ref 4.: https://www.uml-diagrams.org/

1 : System Concepts for Object Modelling

IT 3106– Object Oriented Analysis and Design

Level II - Semester 3

Overview

In this section students

- will learn the following Object Oriented concepts
  - Classes and Objects
  - Methods and Messages
  - Encapsulation and Information Hiding
  - Inheritance
  - Polymorphism and Dynamic Binding
- will be introduced to Unified Modeling Language (UML) and its 14 diagrams mentioned below.

  Class, Object, Use Case, Activity, Sequence, Communication, Package, State, Component, Deployment, Interaction Overview, Composite Structure, Timing , Profile

Intended Learning Outcomes

At the end of this lesson, you will be able to;

- Identify and describe the Object-Oriented concepts
- Define Unified Modeling Language (UML) and its various types of diagrams

List of Subtopics

1.1 Classes and Objects [Ref 1: Pg. 19-20]

1.2 Methods and Messages [Ref 1: Pg. 20]

1.3 Encapsulation and Information Hiding [Ref 1: Pg. 20]

1.4 Inheritance [Ref 1: Pg. 21]

1.5 Polymorphism and Dynamic Binding [Ref 1: Pg. 22]

1.6 Introduction to Unified Modeling Language (UML) [Ref 1: Pg. 34-35, Ref 4]

Ref 1: Alan Dennis, Barbara Haley, David Tegarden, Systems analysis design, An Object Oriented Approach with UML : an object oriented approach, 5th edition, John Wiley & Sons, 2015, ISBN 978-1-118-80467-4

Ref 4.: https://www.uml-diagrams.org/

1.1 Classes and Objects

- A problem can be divided into subjects.
- Subjects are used as logical divisions for development of large systems
- Each subject is generally considered as a problem domain

Subjects eg. ***Airport Management System***

1.1 Classes and Objects

- We understand the real world through ideas that are organized into recognizable ***patterns*** and ***concepts***.
- *Concepts* can be divided into following types :

Tangible - *book*
Roles - *doctor*
Relational - *marriage*
Intangible - *time, quality*
Judgmental - *good pay*
Event - *Purchase, loan*

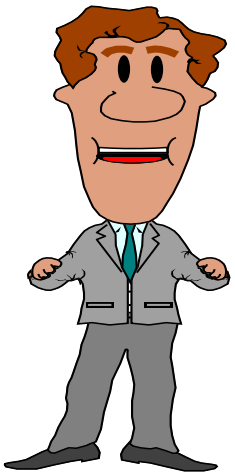
1.1 Classes and Objects

System concepts for object modeling

- In the real-world interaction of these **concepts** allow complex operations to be performed.

Library System

Core Concepts - *Borrowing, Returning of books, Book registration, Member registration (Use Cases)*



Instance of a Member Concept

Concepts - *Member, Copy, Book (Classes)*

*** He knows certain facts -** eg. Name address etc.

*** Also perform certain activities**

1.1 Classes and Objects

- **Classes**
  - Represent the concepts that are to be modeled.
  - Initial Step in OOA is the identification of Actors, Use cases, *classes (Problem Domain)*.
  - Library System: identify *Use Cases-Borrowing books, returning of books , registration of Books* in a library system
  - Classify type of expected users (Actors) of the system (Librarian, Member etc.)
  - *Classes* in a library system - *Book, Copy, Member etc. (problem Domain Classes)*

1.1 Classes and Objects

- **Objects (Instances of Classes)**
  - Something that is or is capable of being seen, touched, or otherwise sensed and about which users store data and associate behavior
  - Types of objects
    - Person: e.g. employee, customer, instructor, student
    - Place: e.g. warehouse, building, room, office
    - Thing: e.g. product, vehicle, computer, videotape
    - Event: e.g. an order, payment, invoice, application
    - Sensual: e.g. phone call, meeting

1.1 Classes and Objects

- **Attributes and Services**

  by examining the descriptions/documents given by the users, event type concepts (Scenarios) can be identified.

  - **Attributes** : The data that represents characteristics of interest about an object.
  - **Services** : The set of things that an object can do and that correspond to functions that act on the object's data or attributes.

1.1 Classes and Objects

- **Attributes and Services**

eg. Library System

Main Actions → *Borrowing of Books*

Main Actions → *Return of Books*

Main Actions → *Reservation of Books*

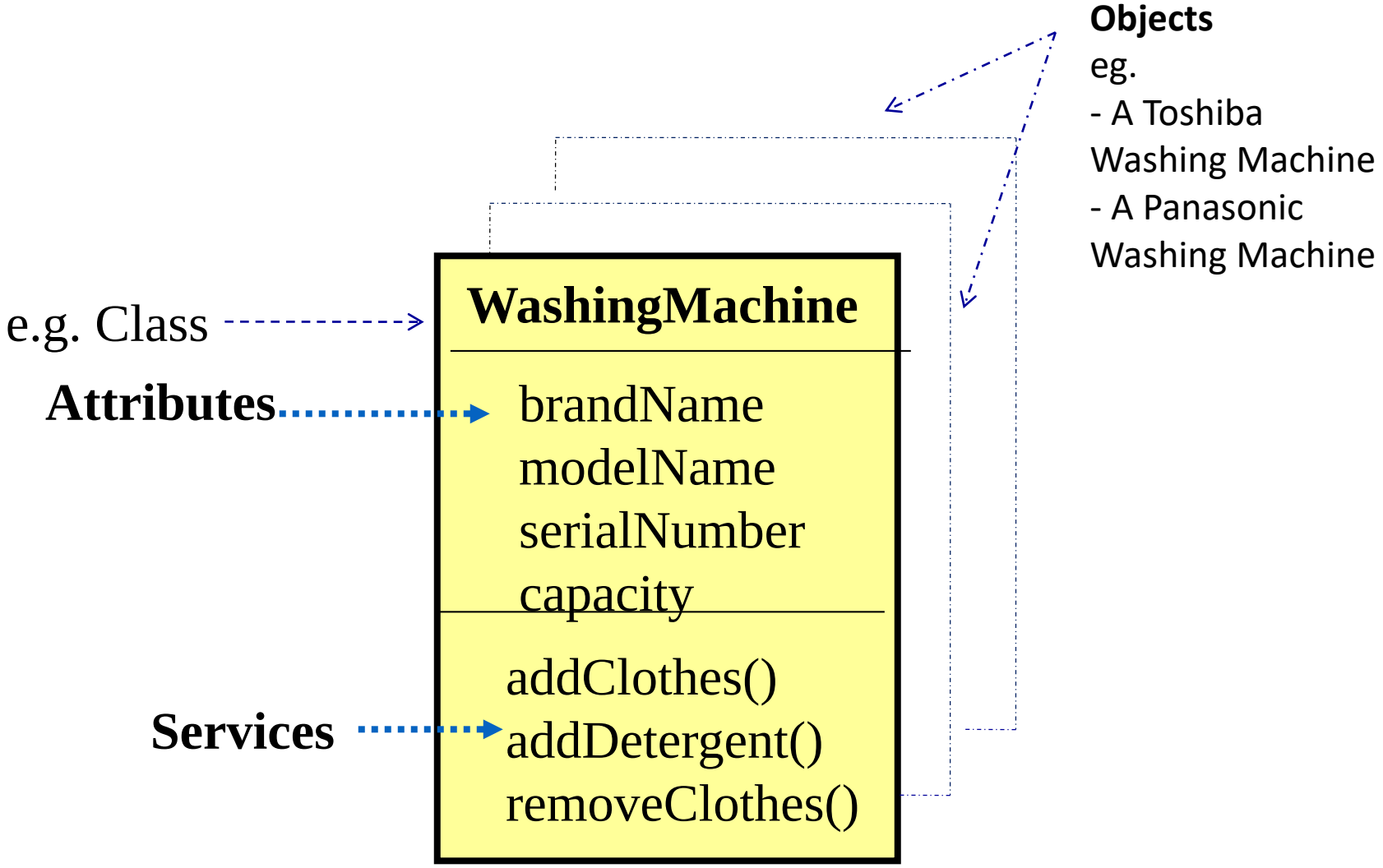
Each Action → Involves with Pool of Objects (Instance of a Class)

Each Class offers

* *Various **Services***

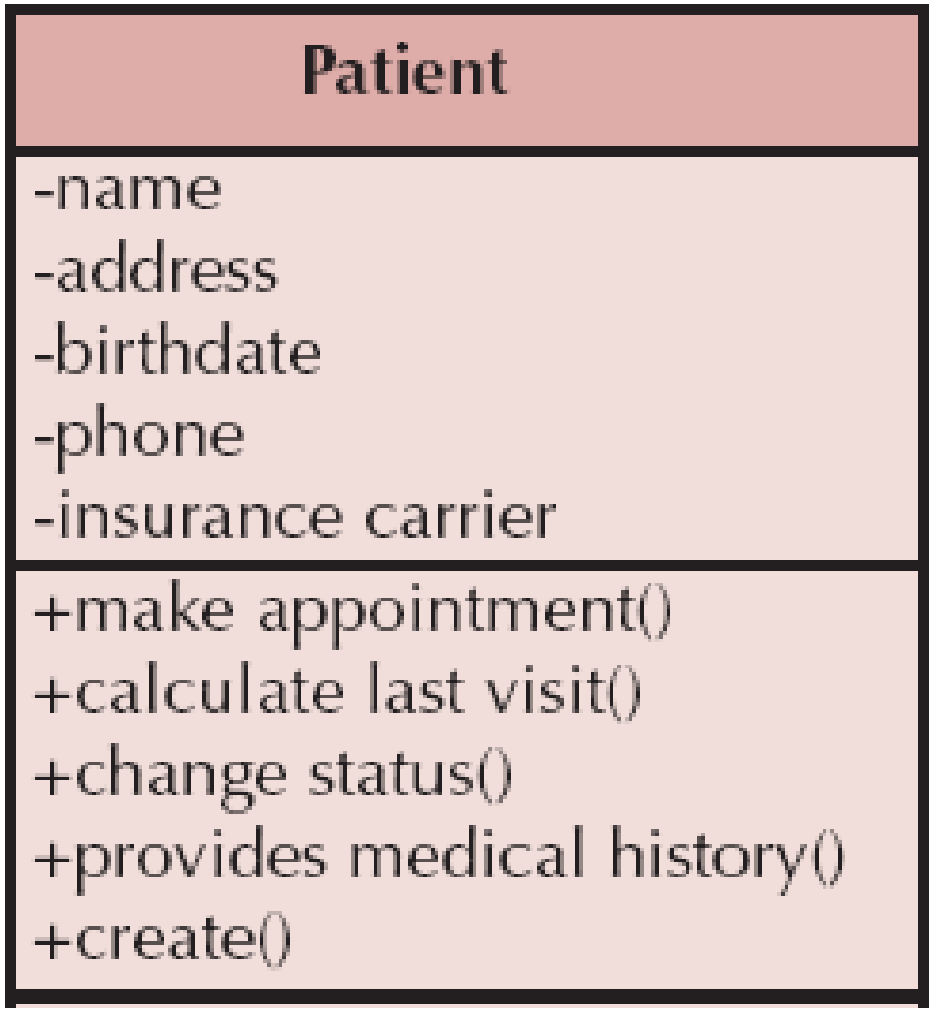
* *Contains information reflected as **Attributes***

1.1 Classes and Objects



1.1 Classes and Objects

Another Example



1.1 Classes and Objects

- A principle used to derive ***attributes*** and ***services***

  * What the specific concept knows ?
    - ***attributes***
  * What are the responsibilities of a class ?
    - ***services***

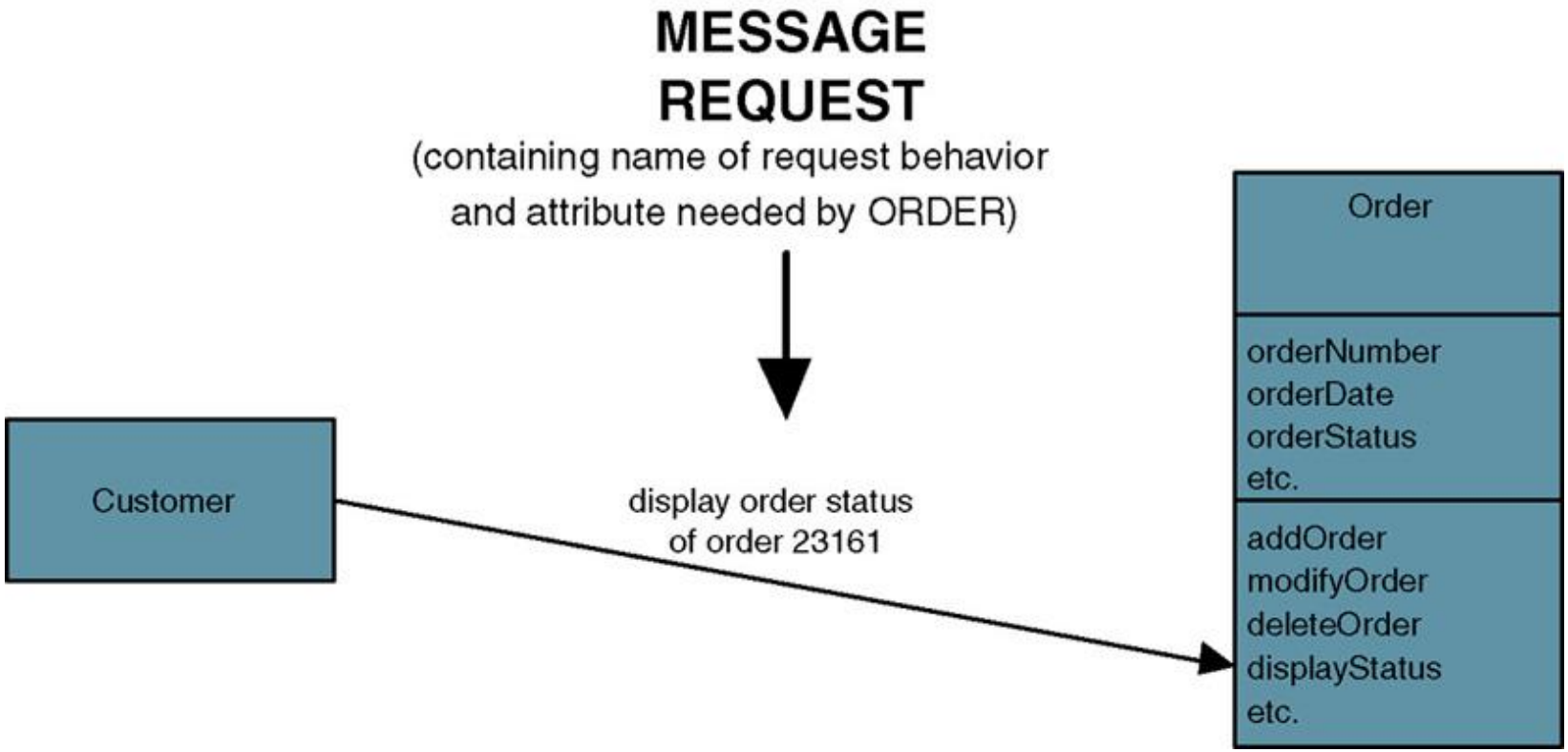
  Services operate when they are involved through message connections.

Student knows how to draw a rectangle ***but*** he will not draw unless somebody ask him to draw

Another example : TV and Remote

1.2 Methods and Messages

Message – Communication that occurs when one object invokes another object's method (behavior) to request information or some action

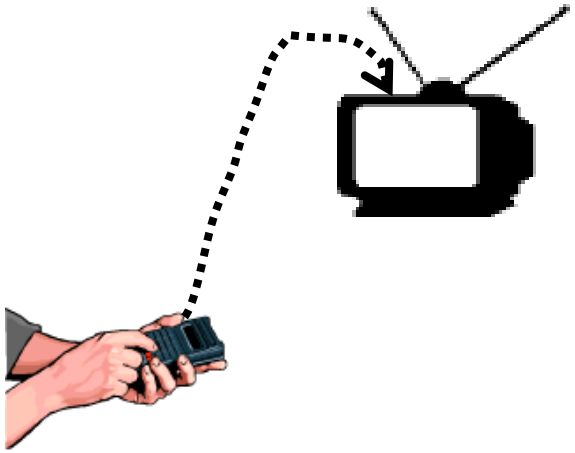


1.2 Methods and Messages

- Message Passing
  - In a system, objects work together.
  - They do this by sending messages to one another.
  - One object sends another a message to perform an operation.
  - The receiving object performs that operation.

    eg. A TV and a Remote

1.2 Methods and Messages



- Message Passing
  - When you want to watch a TV show,
    - You hunt round for a remote,
    - Settle into your favorite chair and
    - Push the *On* Button.
  - What happens?
    - The remote object sends a message (literally!) to the TV object to turn itself on.
    - The TV object receives the message.
    - It knows how to perform the turn-on operation and turns itself on.

1.2 Methods and Messages

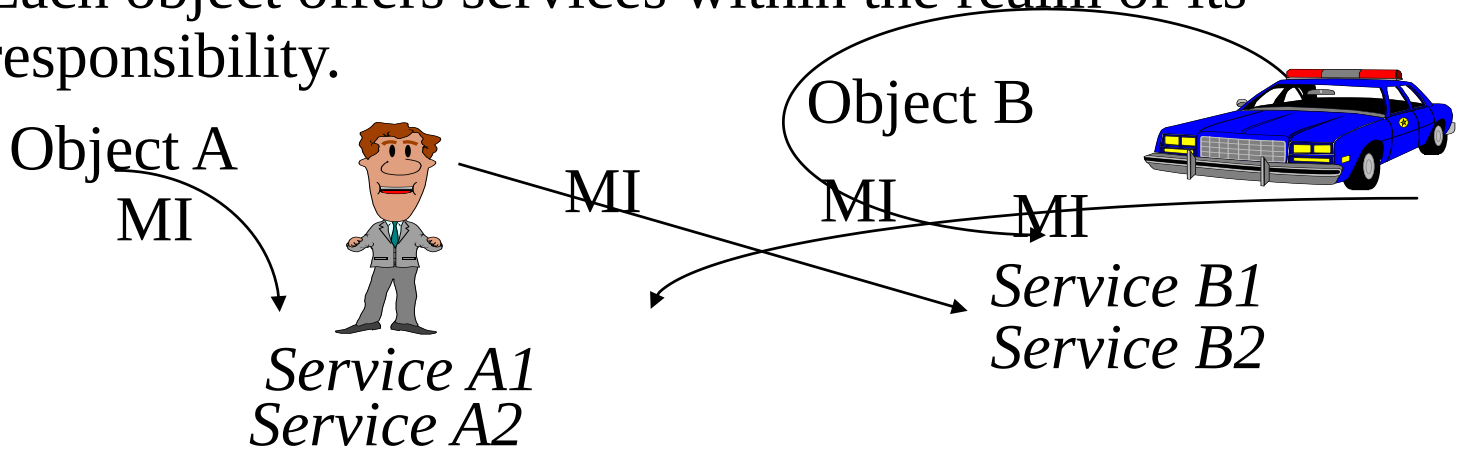
- Message Passing
  - When you want to watch a different channel,
    - You click the appropriate button on the remote,
    - Remote object sends a different message *change channel* to the TV object.
  - The remote can also communicate with the TV via other messages for *changing the volume, muting the volume etc.*.

1.2 Methods and Messages

- Message Passing
  - Let's go back to interfaces for a moment.
  - Most of the things you do from the remote, you can also do by *getting out of the chair*, *going to the TV*, and *clicking buttons on the TV.*
  - The interface the TV presents to you is obviously not the same interface it presents to the remote.

1.2 Methods and Messages

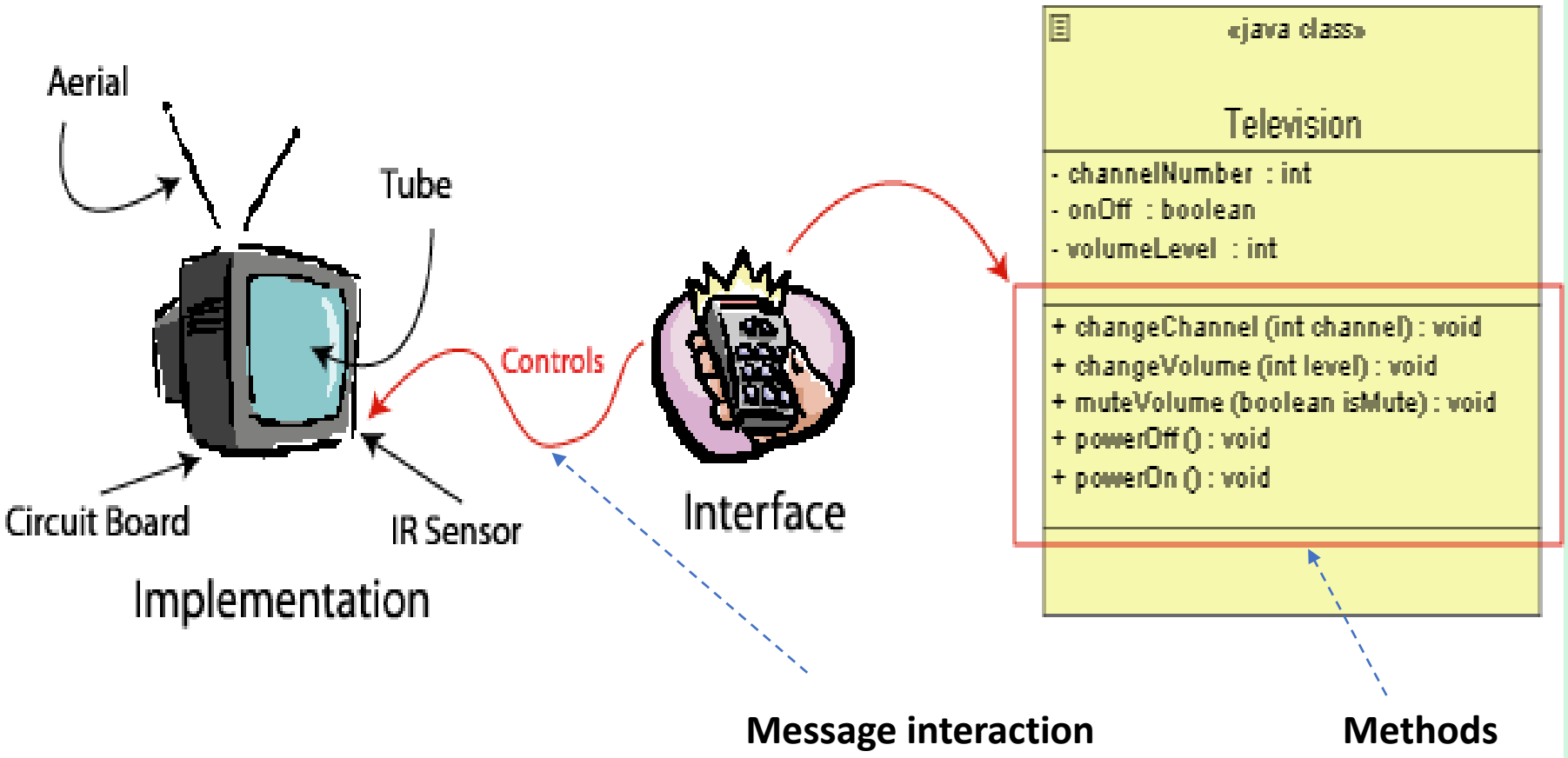
- Each object offers services within the realm of its responsibility.



* Objects use other objects services through *message passing*

* An object may even call its own services

1.2 Methods and Messages



1.3 Encapsulation and Information Hiding

- **Data Abstraction**
  - A mechanism to reduce and filter out details so that one can focus on a few concepts at a time. Focus on Essentials
  - Filter out an object's properties and operations until just the ones you need are left.
  - Ignore the irrelevant. Ignore the unimportant.

    eg. Book – ~~Bookshop you bought~~ -Not important
- Abstraction, information hiding, and encapsulation are very different, but highly-related, concepts.
- One could argue that abstraction is a technique that help us identify which specific information should be visible, and which information should be hidden.
- **Encapsulation** is then the technique for packaging the information in such a way as to hide what should be hidden and make visible what is intended to be visible.

1.3 Encapsulation and Information Hiding

- **Encapsulation**
  - **Packaging of several items together into one unit (both attributes and behavior of the object), Also protects the contents.**
  - **The only way to access or change an object's attribute is through that object's specific behavior.**
  - **Objects *encapsulates* what they do.**
    - That is, they hide the inner workings of their operations
      - **from the outside world**
      - **and from other objects**

Eg. Class BankAcount
{
Private: Balance
……
Withdraw(int)
Deposit(int)
}

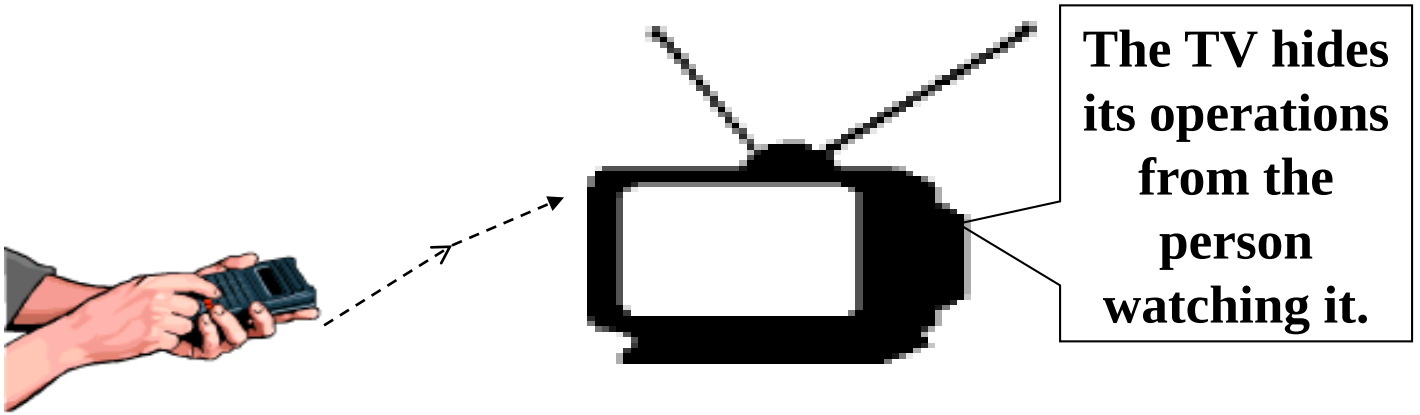
1.3 Encapsulation and Information Hiding

When an object carries out its operations, those operations are hidden.

E.g. When most people watch a television show,

- they usually don't know or care about the complex electronics that sit behind the TV screen

- or the operations that are happening.



1.3 Encapsulation and Information Hiding

- **Information Hiding**
  - The principle of information hiding suggests that only the information required to use a software module be published to the user of the module.
  - Typically, this implies that the information required to be passed to the module and the information returned from the module are published.
  - It is not relevant to know exactly how the module implements the required functionality
  - We really do not care how the object performs its functions, as long as the functions occur.
  - In object-oriented systems, combining encapsulation with the information-hiding principle supports treating objects as black boxes.(a black box is a device or object which can be viewed in terms of its inputs and outputs without any knowledge of its internal workings.)

1.3 Encapsulation and Information Hiding

1.3 Encapsulation and Information Hiding

Why is this important?

- In the software world, encapsulation helps cut down on the potential for bad things to happen.
- In a system that consists of objects, the object depends on each other in various ways.
- If one of them happen to malfunction, software engineers must change it in some way.
- Hiding its operations from other objects means it probably won't be necessary to change those other objects.

1.3 Encapsulation and Information Hiding

- Turning from software to reality, you see the importance of *encapsulation*.

eg. Your computer monitor, hides its operations from your computers CPU.

When something goes wrong with your monitor, you either fix or replace it.

You probably won't have to fix or replace the CPU along with it.

1.3 Encapsulation and Information Hiding

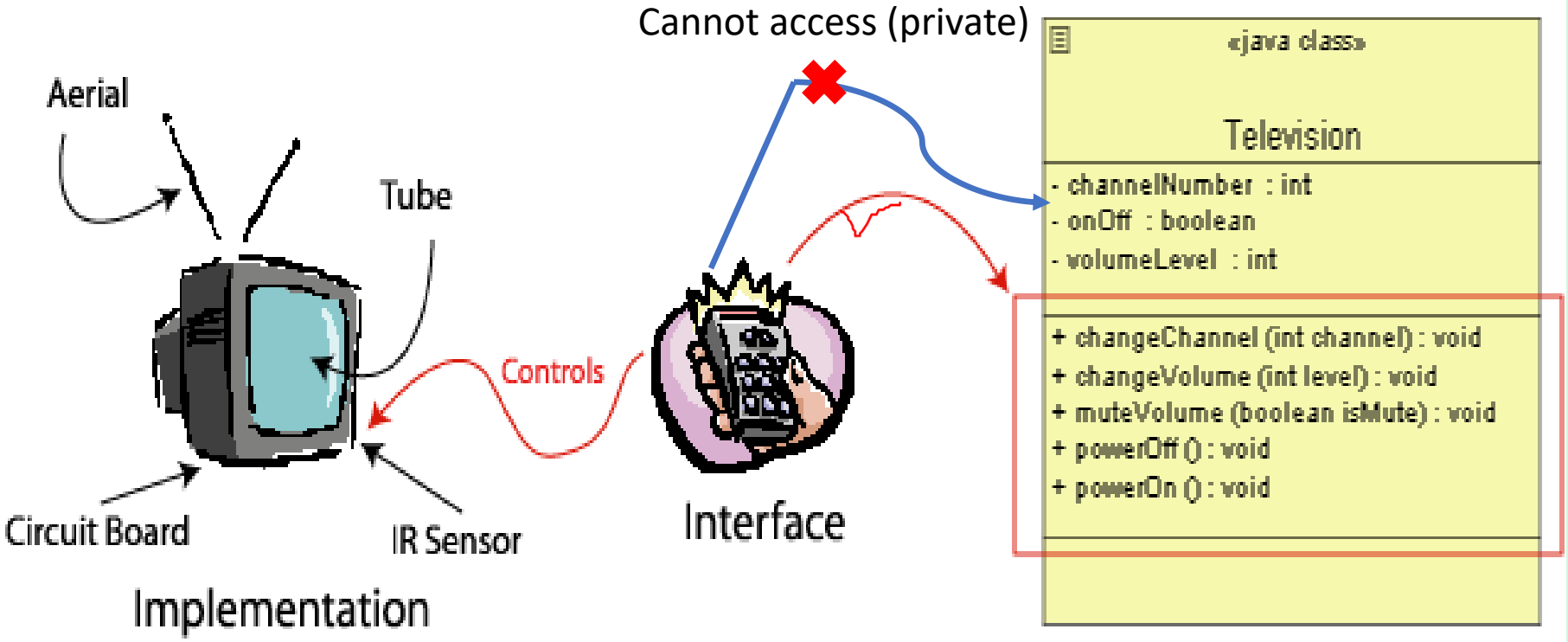
- An object hides what it does from other objects and from outside world
- But an object does have to present a “face“ to the outside world, so you can initiate those operations.

eg. A TV, has a set of buttons either on

the TV or on a remote.

The TV’s buttons are called *interfaces*.

1.3 Encapsulation and Information Hiding



- **Private :** cannot access from outside objects
+ **Public** : can access from outside objects

1.3 Encapsulation and Information Hiding

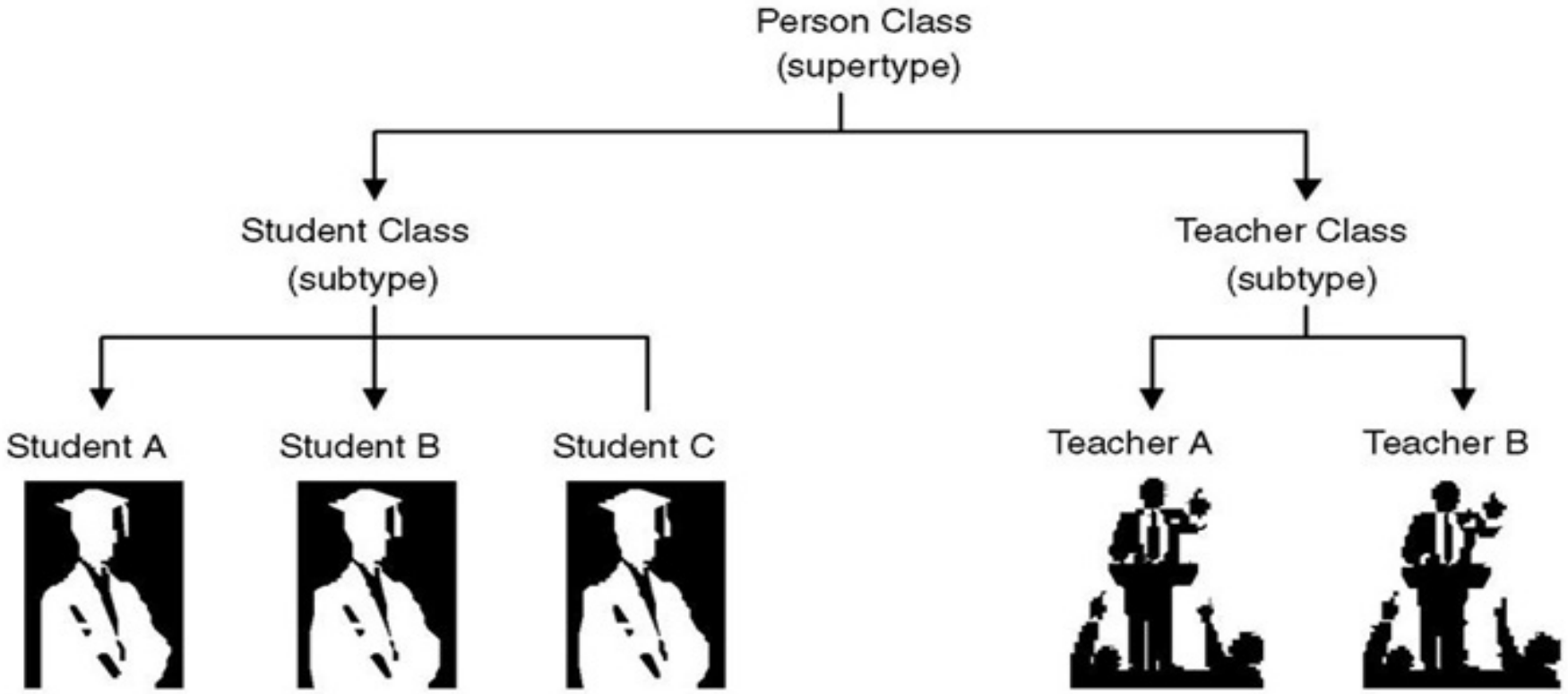
- Encapsulation is done through the definition of *region of access*.
- *Region of access* defines the accessibility of the *services* or *attributes* in a class.
- In C++ (OO programming Language)
  - 3 types of access regions: *private, public, protected*
- *In Java*
  - *Access modifiers: default, public, private, protected,*
  - *Static modifier, final modifier, synchronized modifier, native modifier*

1.4 Inheritance

- The **concept** wherein methods and/or attributes defined in a class can be inherited or reused by another class.
- The data modeling literature suggests using inheritance to identify higher-level, or more general, classes of objects.
- Common sets of attributes and methods can be organized into super-classes.
- Typically, classes are arranged in a hierarchy whereby the super-classes, or general classes, are at the top and the subclasses, or specific (specialized) classes, are at the bottom.

e.g. some individuals in the room might be classified as STUDENTS and TEACHERS. Thus, STUDENT and TEACHER classes are members of the class PERSON

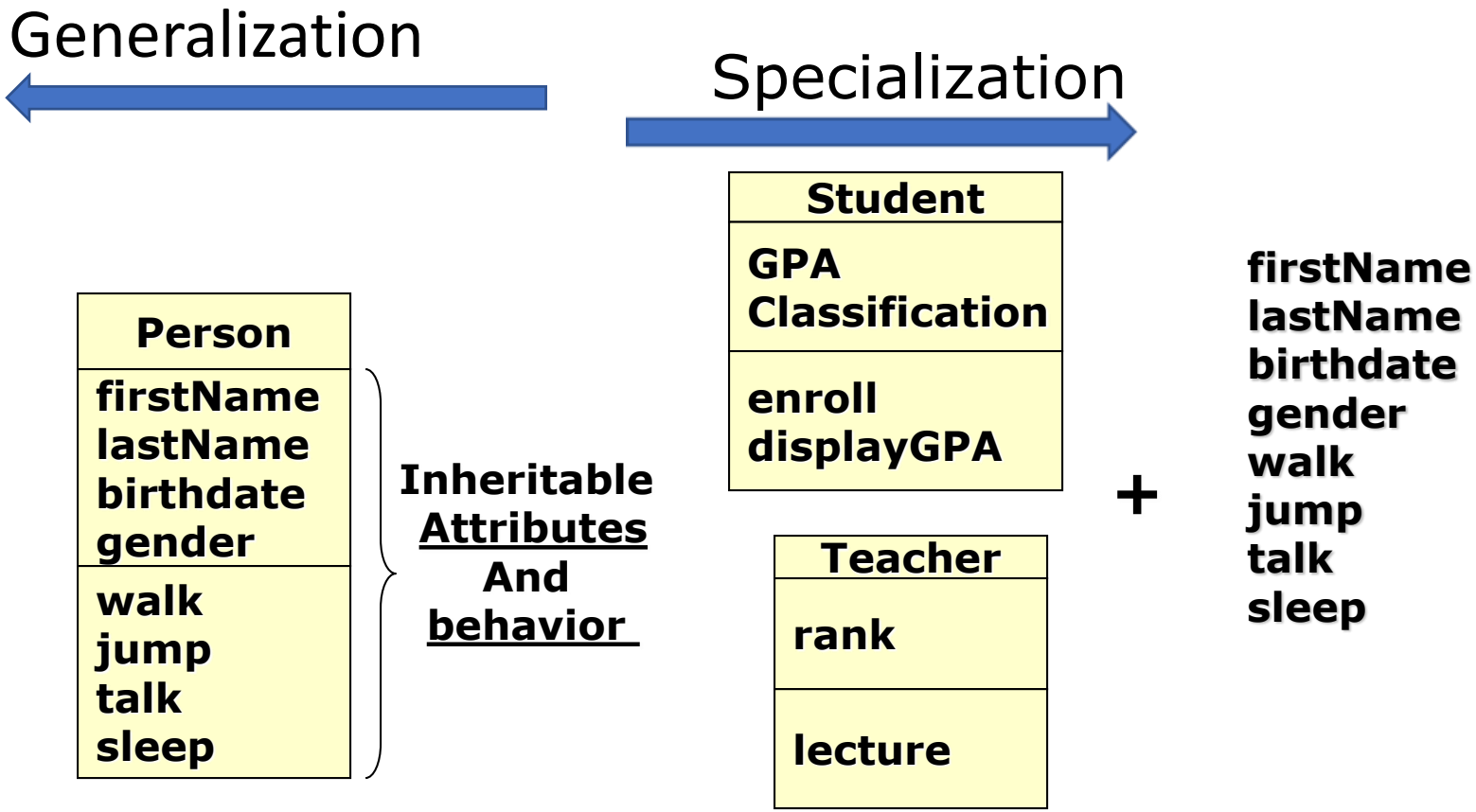
1.4 Inheritance



1.4 Inheritance

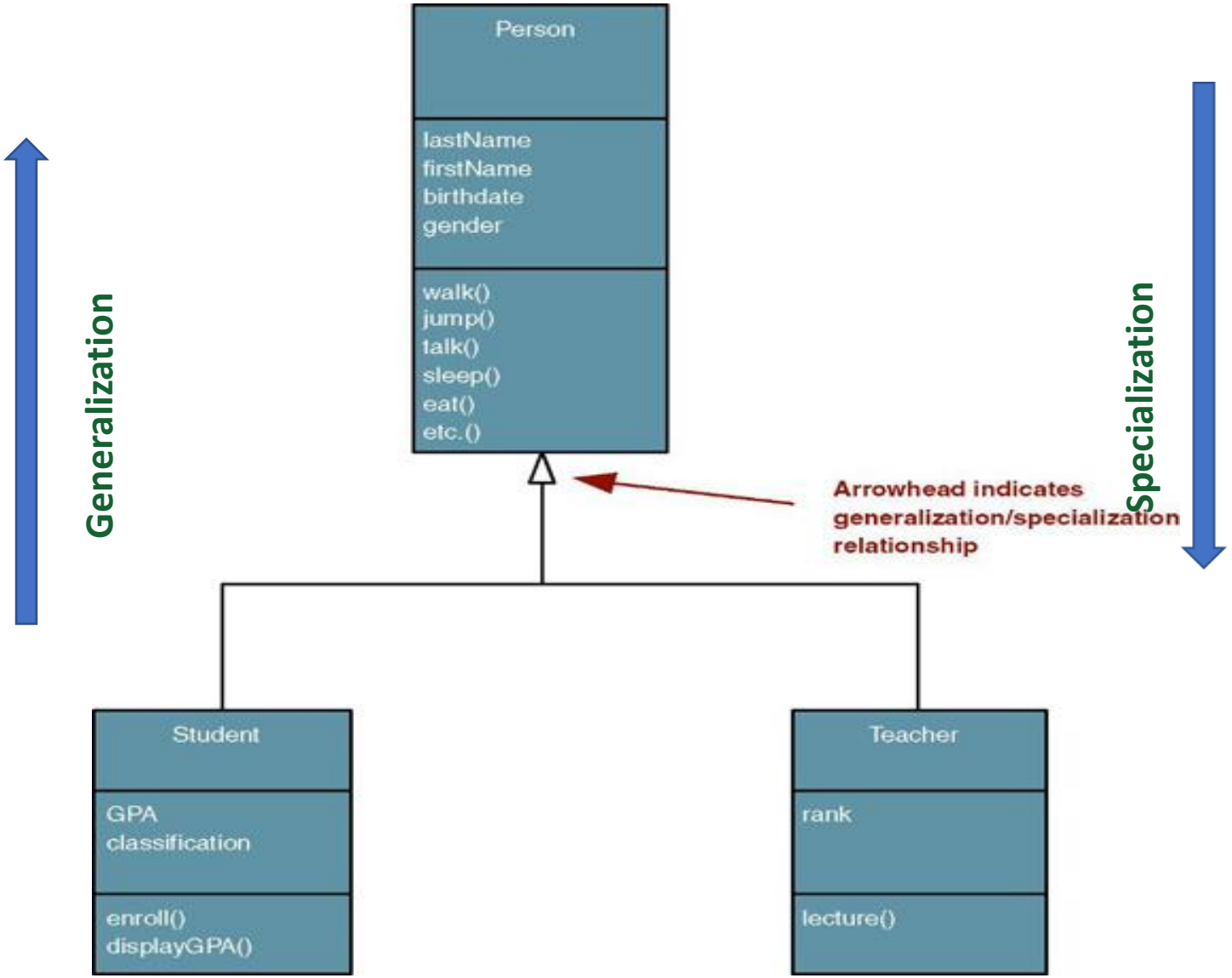
- Generalization / Specialization
  - A **technique** wherein the attributes and behaviors that are common to several types of classes are grouped / abstracted into their own class called a supertype.
  - The attributes and methods of the supertype class are then inherited by those classes (subtype)
  - Sometimes abbreviated as gen/spec.
- Generalization is the term that we use to denote abstraction of common properties into a base class. When we implement Generalization in a programming language, it is called Inheritance.

1.4 Inheritance



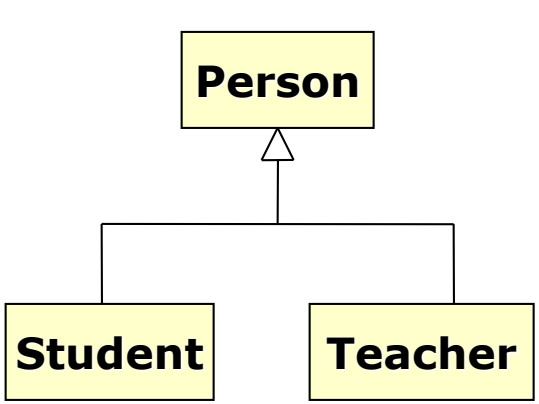
1.4 Inheritance

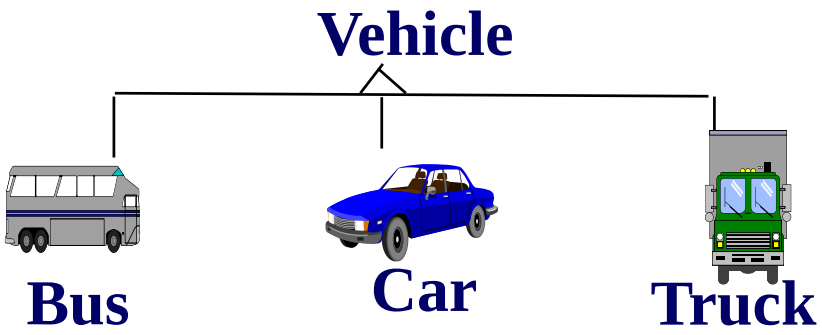
An Example of UML (Unified Modeling Language) Representation of Generalization /Specialization



1.4 Inheritance

- Generalization / Specialization





*** Specialized classes inherit from the parent class**

1.5 Polymorphism and Dynamic Binding

- Polymorphism is the concept that different objects can respond to the same message in different ways.
  - It allows different forms of the same service to be defined.
  - Sometimes an operation has the same name in different classes.

eg. You can open a door, you can open a window, or a bank account. In each case, you are performing a different operation.

1.5 Polymorphism and Dynamic Binding

- We can simply send a message to an object, and that object will be responsible for interpreting the message appropriately.
- For example, if an artist sent the message ‘draw yourself’ to a square object, a circle object, and a triangle object, the results would be very different, even though the message is the same.

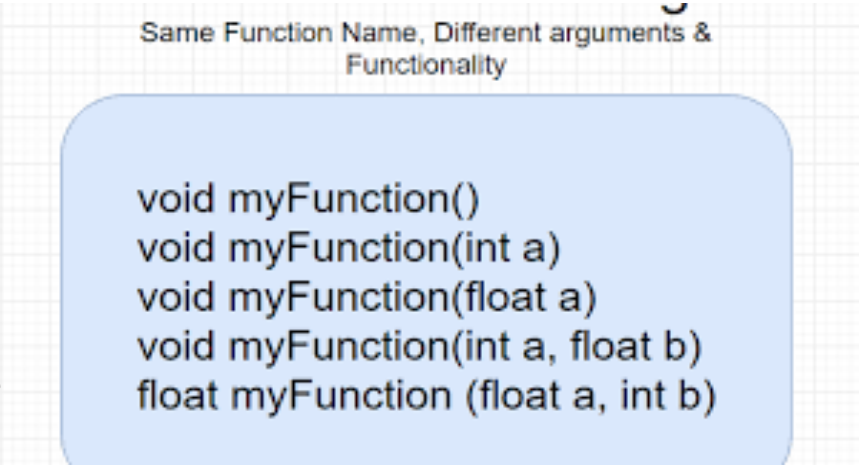
1.5 Polymorphism and Dynamic Binding

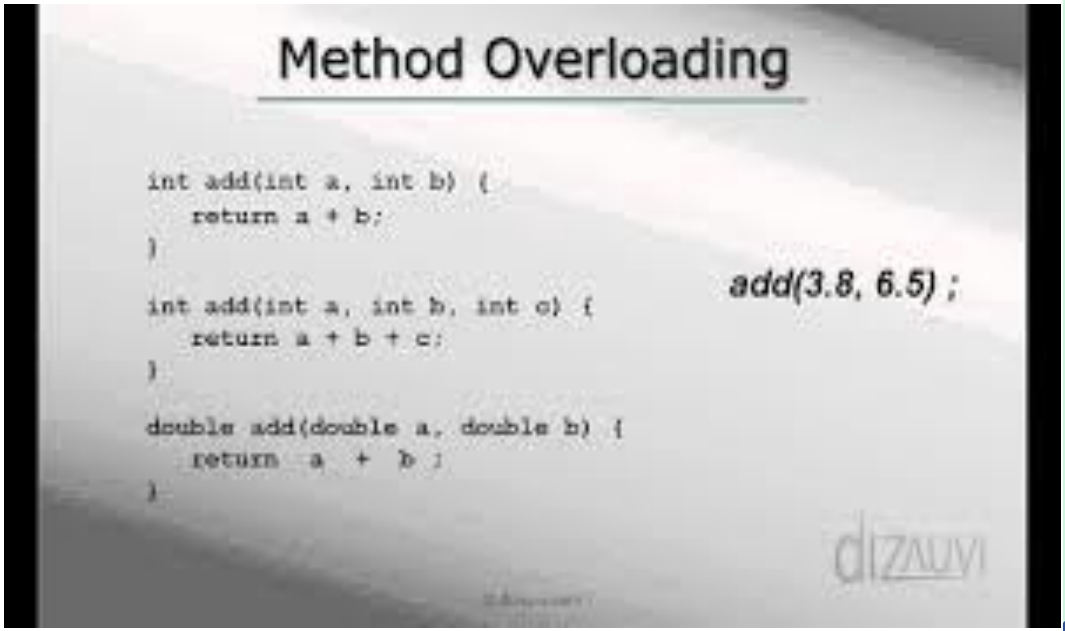
Related terms used in OO programming language

- Function Overloading,
- Operator Overloading,
- Method Overriding

1.5 Polymorphism and Dynamic Binding

In some programming languages, **function overloading** or **method overloading** is the ability to create multiple functions of the same name with different implementations.

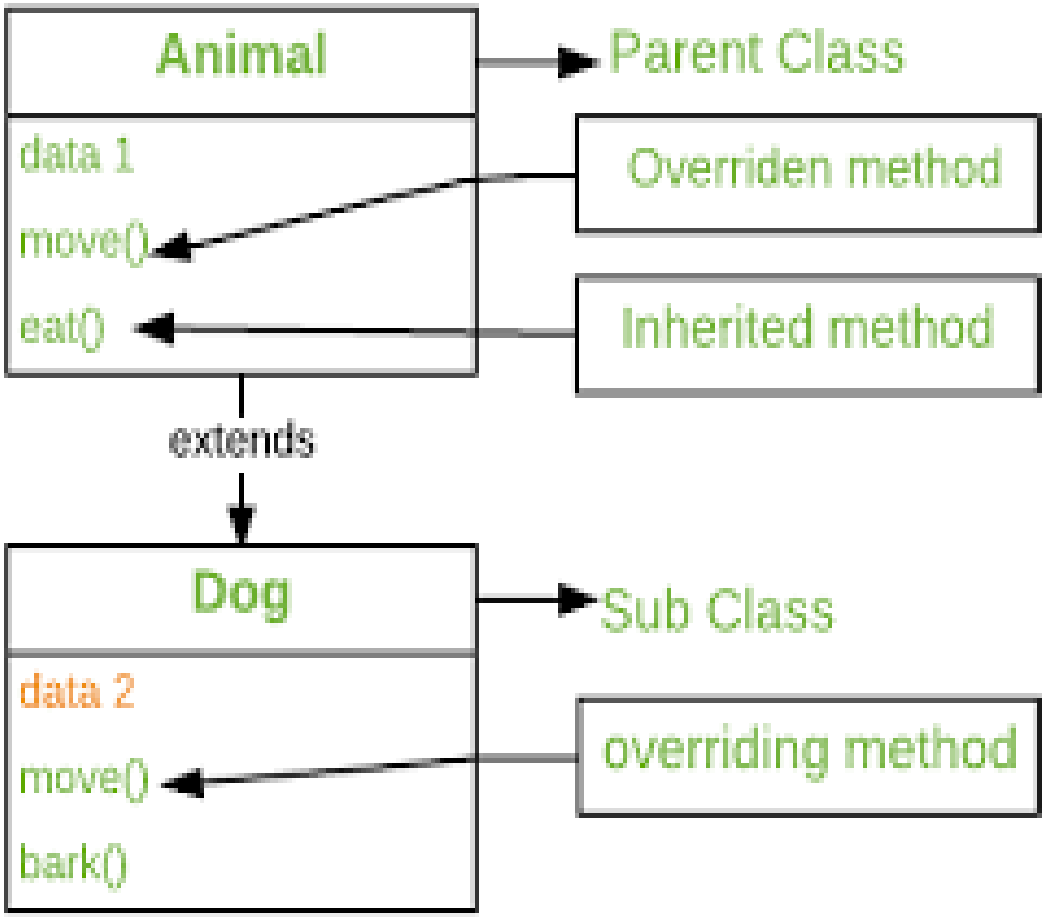




1.5 Polymorphism and Dynamic Binding

Polymorphism

Overriding – a technique whereby a subclass (subtype) uses an attribute or behavior of its own instead of an attribute or behavior inherited from the class (supertype).



1.5 Polymorphism and Dynamic Binding

Polymorphism – Operator Overloading

In languages like C++, we can change the way operators work for user-defined types like objects and structures. This is known as operator overloading.

Suppose we have created three objects **c1**, **c2** and **result** from a class named **Complex** that represents complex numbers.

we can redefine how the + operator works and use it to add the complex numbers of c1 and c2 by writing the following code:

```
result = c1 + c2;
```

instead of something like

```
result = c1.addNumbers(c2);
```

1.5 Polymorphism and Dynamic Binding

Operator Overloading in C++

```
 class complex
{
 float real,imag;
  public : complex operator+(complex&);
                complex() { } //default constructor
                complex(float r, float s) // constructor
                   { real=r; imag=s;}
                void printcomplex( )
                   {
              std::cout<<"Real component="<<real<<'\n';
             std::cout<<"Imaginary component="<<imag;
                   }
};
```

1.5 Polymorphism and Dynamic Binding

complex operator+(complex&);

- Definition of the operator is like a member function, except the operator symbol is prefixed by the keyword operator.

```
complex complex::operator+(complex &a) {
  complex z;
  z.real=real+a.real;
  z.imag=imag+a.imag;
  return z;
  }
```

```
int main()
  {
   complex var1(1.2,3.4), var2(4.5,6.5), outc;
   outc=var1+var2;
   outc.printcomplex();
   }
```

Sample Output :

Real component = 5.7

Imaginary component = 9.9

1.5 Polymorphism and Dynamic Binding

- Polymorphism is made possible through **dynamic binding**. Dynamic, or late, binding is a technique that delays typing the object until run-time. The specific method that is actually called is not chosen by the object-oriented system until the system is running.
- This is in contrast to static binding. In a statically bound system, the type of object is determined at compile-time. Therefore, the developer has to choose which method should be called instead of allowing the system to do it.
- This is why most traditional programming languages have complicated decision logic based on the different types of objects in a system.
- For example, in a traditional programming language, instead of sending the message *Draw yourself* to the different types of graphical objects we would have to write decision logic using a case statement or a set of if statements to determine what kind of graphical object we wanted to draw, and we would have to name each draw function differently

  (e.g., draw square, draw circle, or draw triangle). Th is obviously makes the system much more complicated and difficult to understand.

1.6 Introduction to Unified Modeling Language (UML)

- Until 1995, object concepts were popular but implemented in many different ways by different developers.
- Each developer had his or her own methodology and notation (e.g., Booch, Coad, Moses, OMT, OOSE)
- Then in 1995, Rational Software brought three industry leaders together to create a single approach to object-oriented systems development.
- Grady Booch, Ivar Jacobson, and James Rumbaugh worked with others to create a standard set of diagramming techniques known as the Unified Modeling Language (UML).
- The objective of UML was to provide a common vocabulary of object-oriented terms and diagramming techniques rich enough to model any systems development project from analysis through implementation.

1.6 Introduction to Unified Modeling Language (UML)

- UML , A visual modeling language enables system builders to create blueprints that capture their visions in a standard, easy-to-understand way
- Provides a mechanism to effectively share and communicate these visions with others (Technical or Nontechnical)

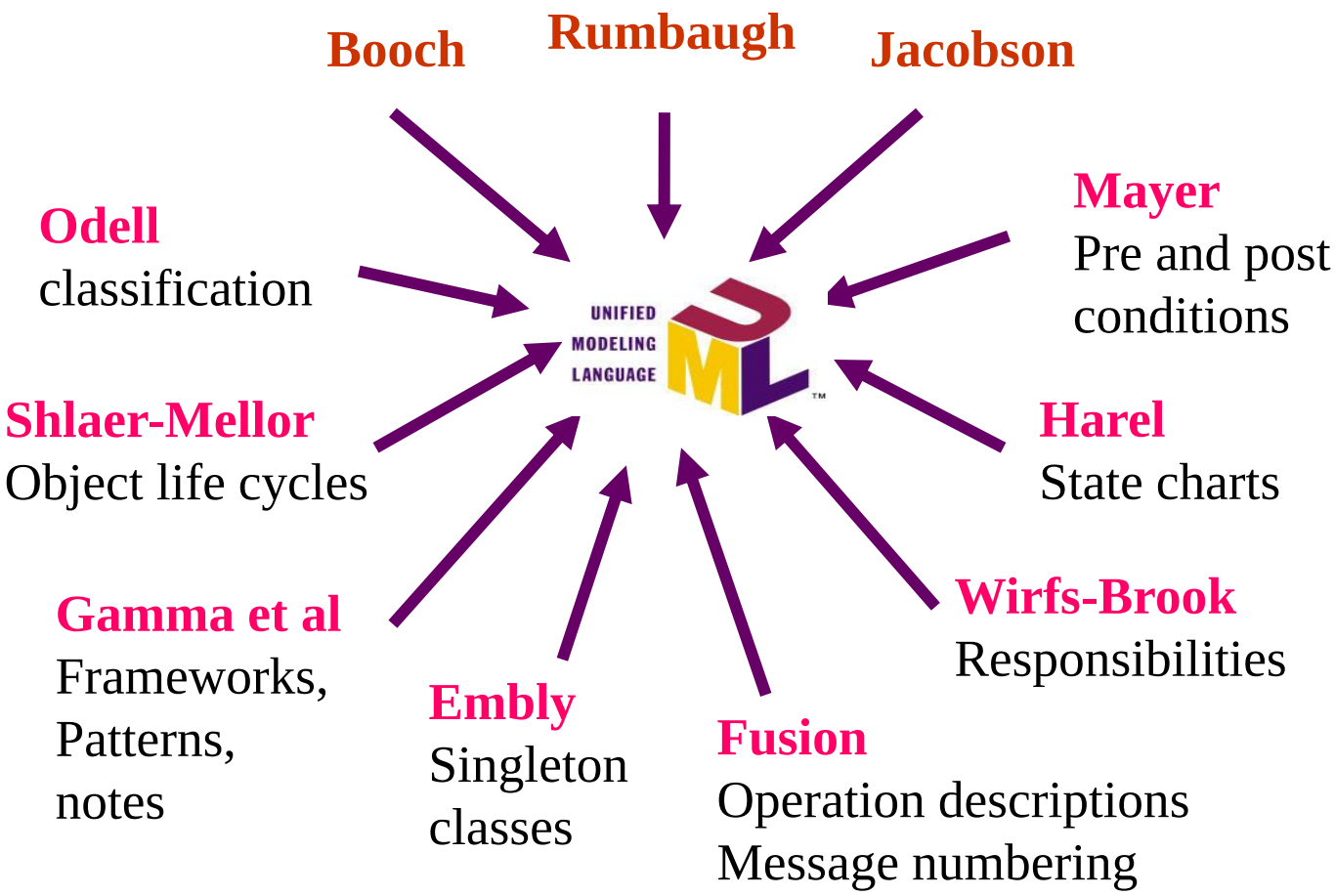
1.6 Introduction to Unified Modeling Language (UML)

- It most directly unifies the methods of
  - Booch,
  - Rumbaugh (OMT) and
  - Jacobson

as well as the best ideas from several other methodologies.

1.6 Introduction to Unified Modeling Language (UML)

UML Inputs



1.6 Introduction to Unified Modeling Language (UML)

- The UML does **not prescribe a method** for developing systems—**only a notation** that is now widely accepted as a standard for object modeling.

UML History

- The first public draft version – (version 0.8) – Oct 1995
- Feedback from public and Ivor Jacobson's inputs included – (ver. 0.9 Jul 1996, ver. 0.91 Oct 1996)
- Ver 1.0/1.1 was presented to OMG group for standardization in July/Sep 1997.
- Nov 1997 – UML adopted as the standard modelling language by OMG
- Ver 1.2 - June 1998
- Ver 1.3 – Dec 1998
- Ver 1.4 – 2000
- Ver 2.0 – 2003
- Ver 2.1 – 2007
- Ver 2.2 - 2009
- Ver 2.3 – 2010,
- Ver 2.4 –Jan 2011
- Ver 2.5 June 2015

1.6 Introduction to Unified Modeling Language (UML)

UML *Components and Capabilities*

- The UML consists of a number of graphical elements that combine to form diagrams.
- Because it is a language, the UML has rules for combining these elements.
- Lets look at the diagrams before looking at the elements and rules.
- The purpose of the diagrams is to present multiple views (model) of a system.
- UML 2.5 include 14 different diagrams.

1.6 Introduction to Unified Modeling Language (UML)

- UML diagrams are divided into two categories.

  - Structure Diagrams :

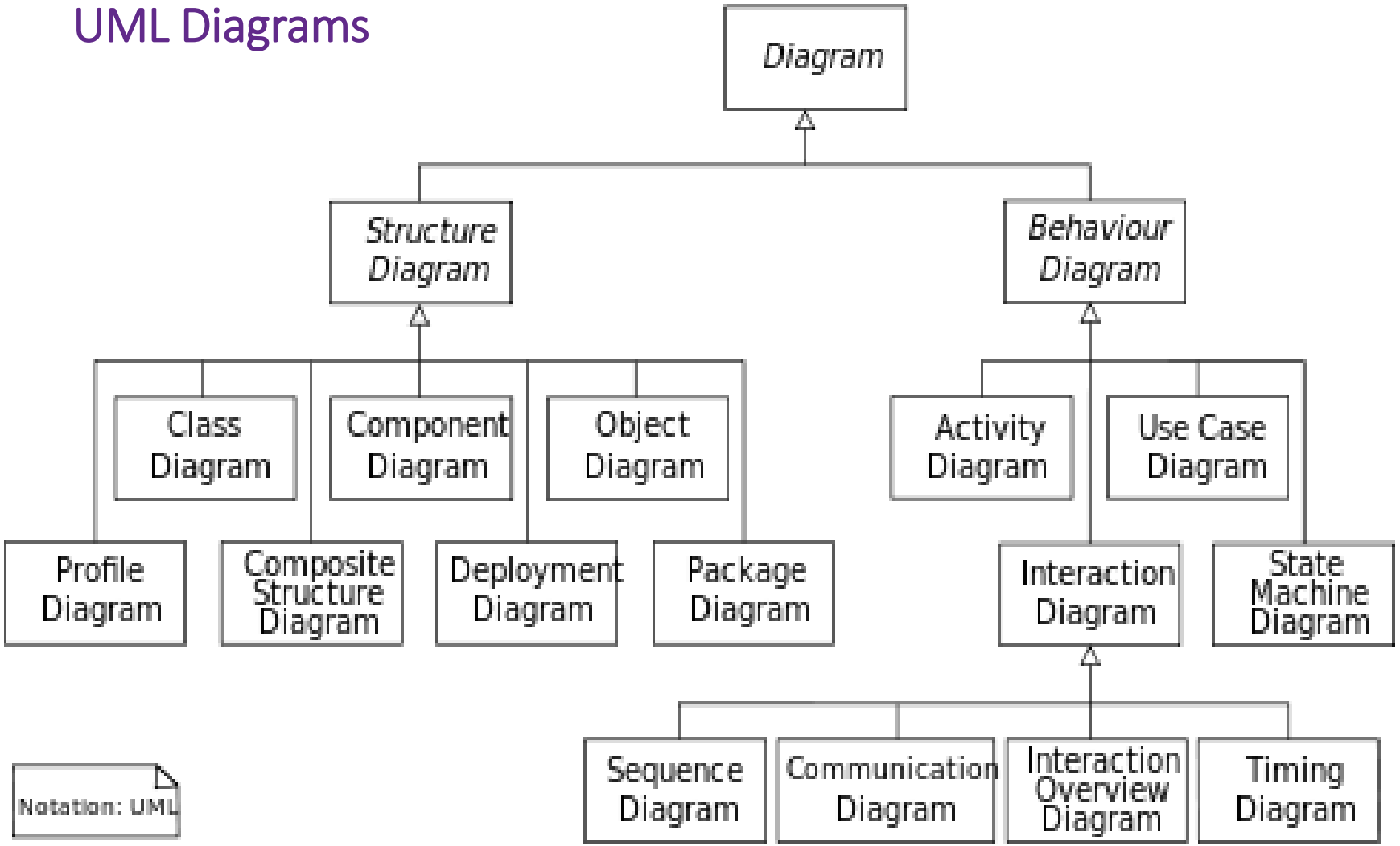
  Seven diagram types represent *structural* information,

  -Behaviour Diagrams :

  other seven represent general types of *behavior*..

1.6 Introduction to Unified Modeling Language (UML)

UML Diagrams



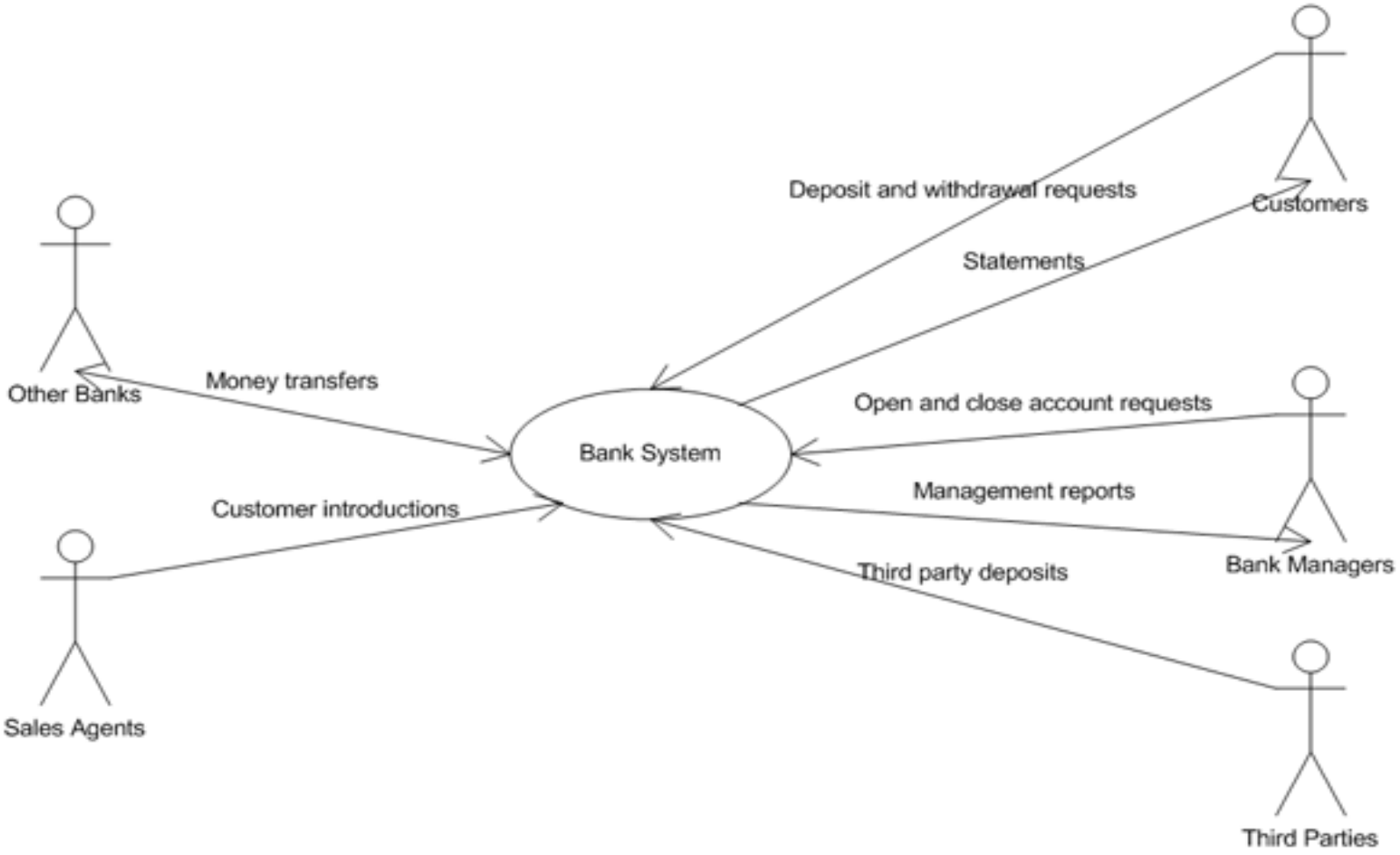
1.6 Introduction to Unified Modeling Language (UML)

Context Models

- Context models are used to illustrate the operational context of a system
- they show what lies outside the system boundaries.
- **Use Case** Diagram can be used to represent a Context model for a system.
- There is no single diagram in UML that would map to the definition

1.6 Introduction to Unified Modeling Language (UML)

Context Models



1.6 Introduction to Unified Modeling Language (UML)

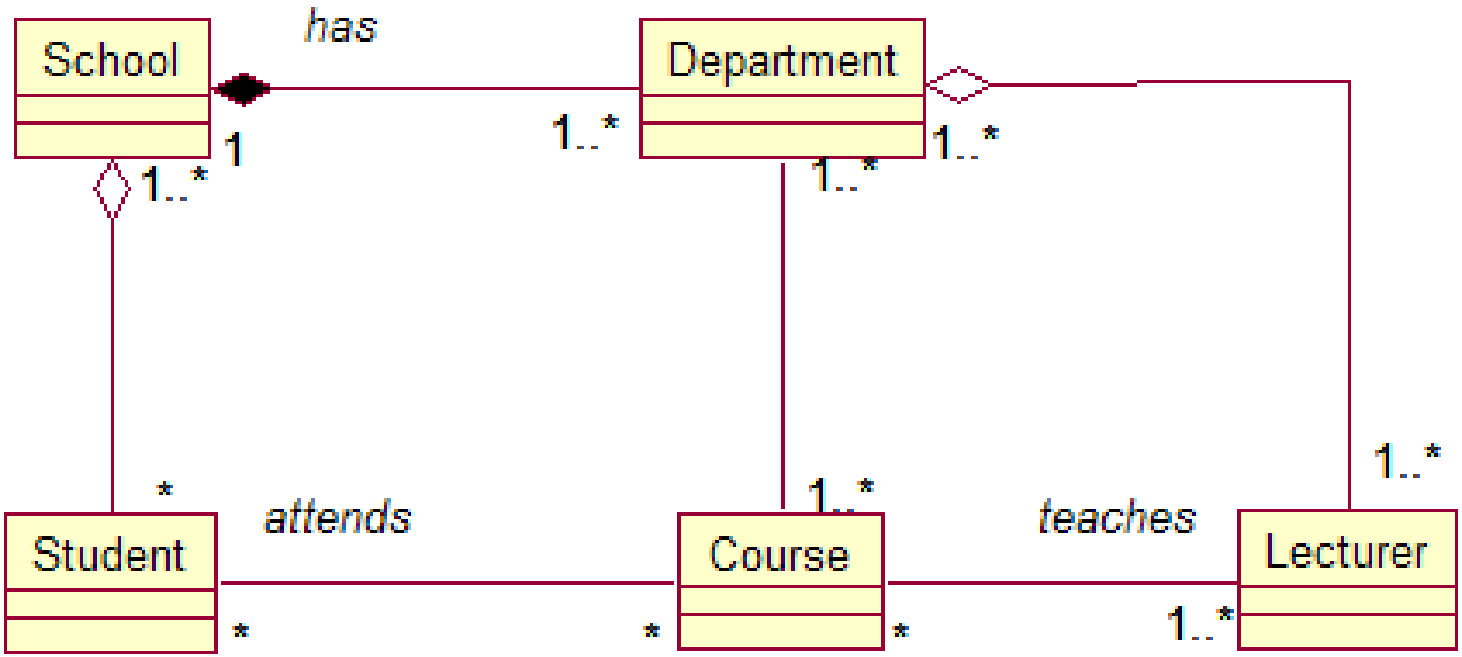
- UML include many different diagrams.

  Class Diagrams :

  - Shows set of *classes*, *interfaces*, and *collaborations* and their relationships.
  - Most common diagram found in modelling object-oriented system.
  - Address the static view of a system.

  class: a category or group of things that have the same attributes and the same behaviours.

1.6 Introduction to Unified Modeling Language (UML)



Class Diagram (UML 1.*)

e.g. School =Faculty

1.6 Introduction to Unified Modeling Language (UML)

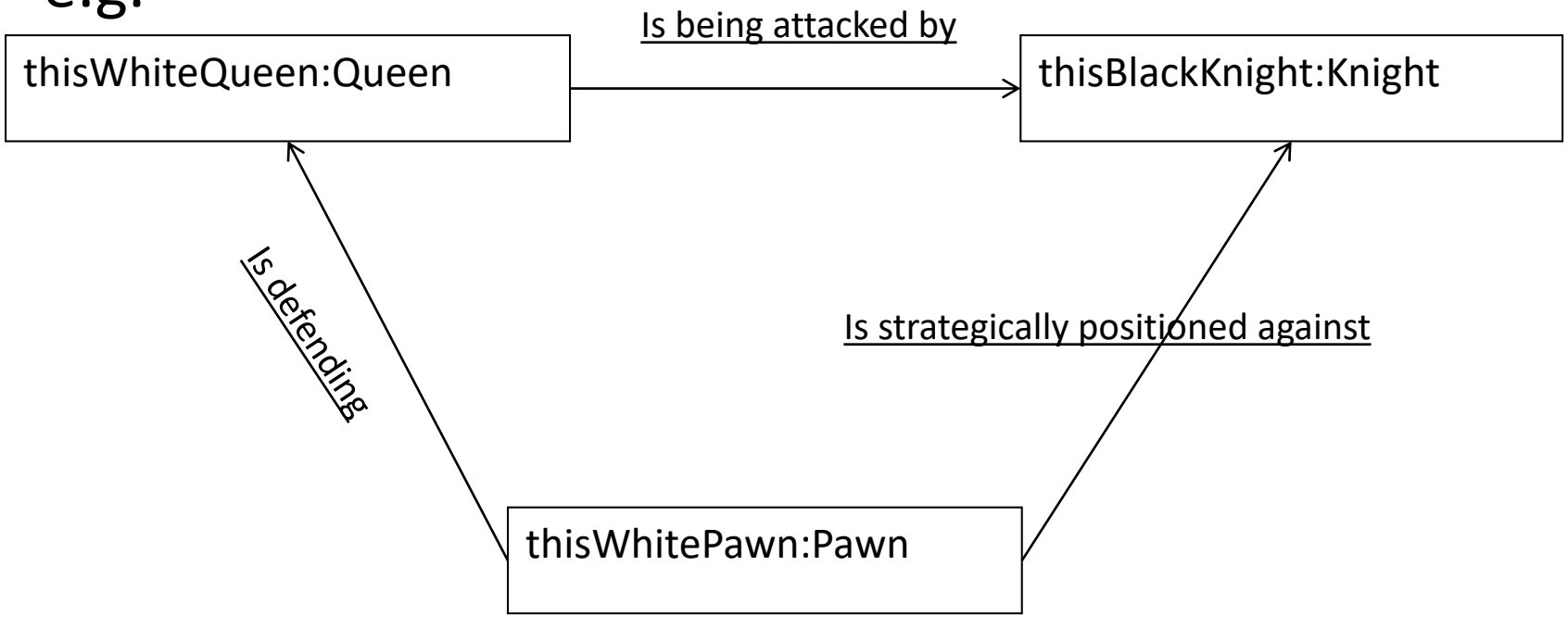
Object Diagrams :

- Models actual object instances with current attribute values.
- Shows a set of *objects* and their relationships.
- Provides a snap shot of the system's objects at one point in time.

1.6 Introduction to Unified Modeling Language (UML)

Object Diagram

e.g.



1.6 Introduction to Unified Modeling Language (UML)

Use Case Diagrams :

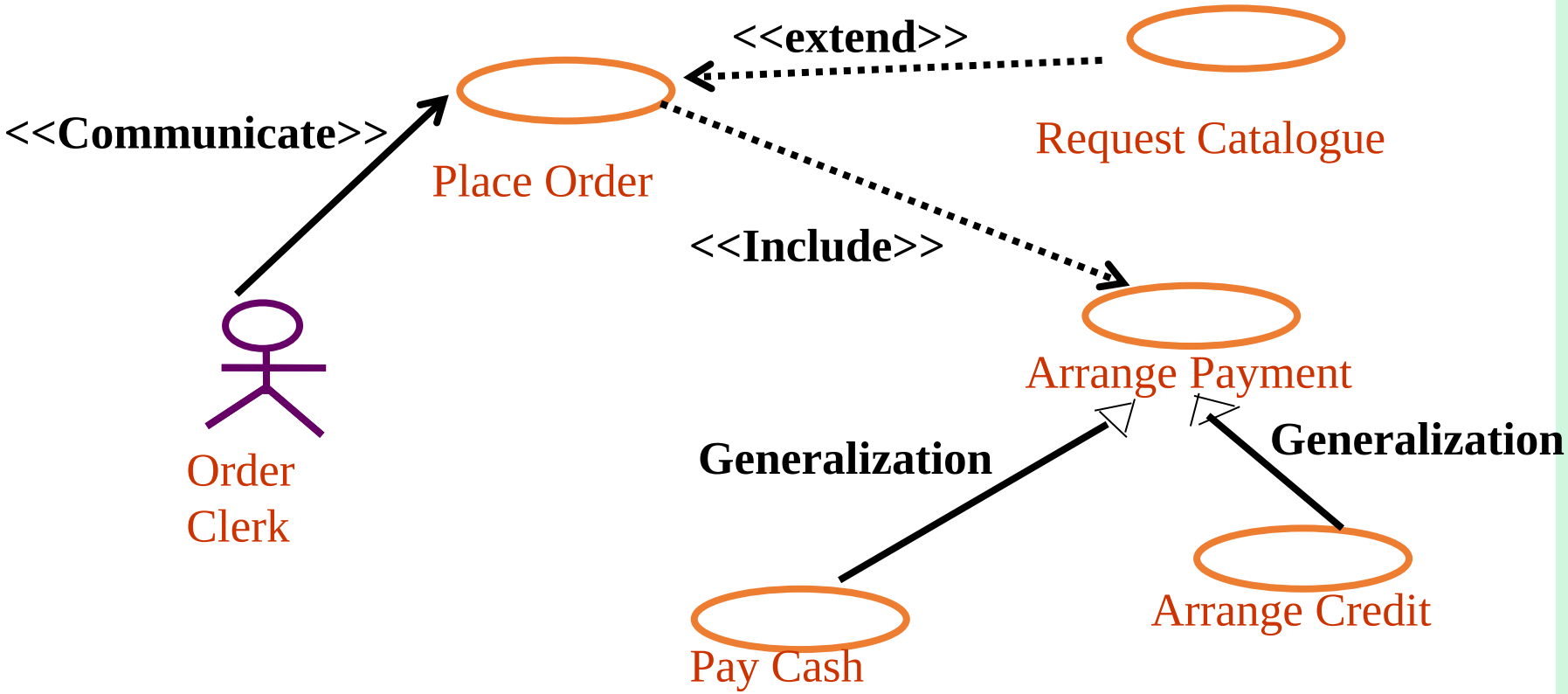
- Shows a set of *use cases* and *actors* and their relationships.
- A tried-and-true technique for gathering information.

- Graphically describes ***who will use the system*** and ***in what ways the user expects to interact*** with the system.

1.6 Introduction to Unified Modeling Language (UML)

Use Case Diagram

Order Processing



1.6 Introduction to Unified Modeling Language (UML)

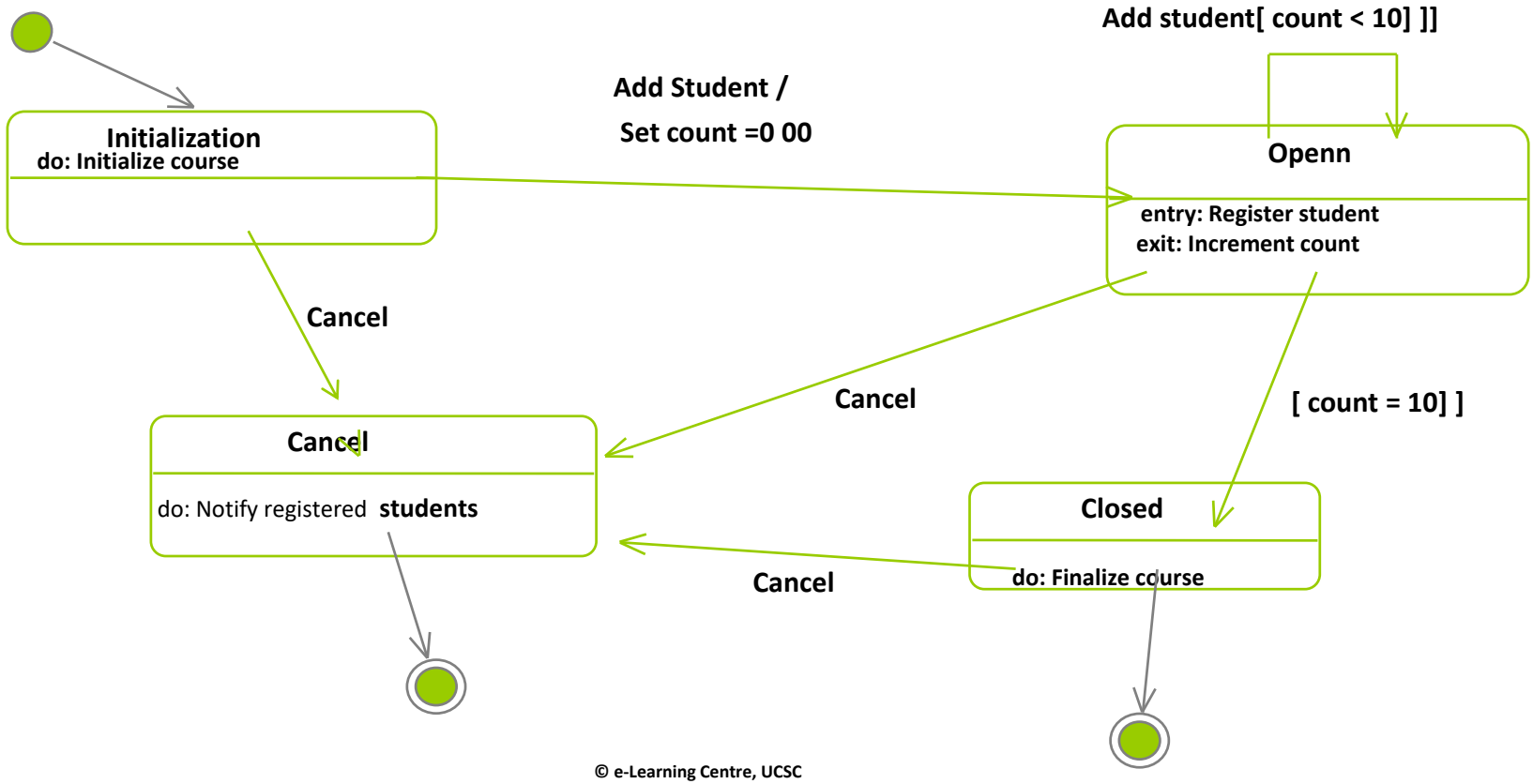
State Diagrams :

- Shows a state machine consisting of states, transitions, events and activities.
- Models how events can change the state of an object over its lifetime,
- show both the various states of an object and the transactions between those states.

1.6 Introduction to Unified Modeling Language (UML)

State Diagrams

e.g. Course Registration



1.6 Introduction to Unified Modeling Language (UML)

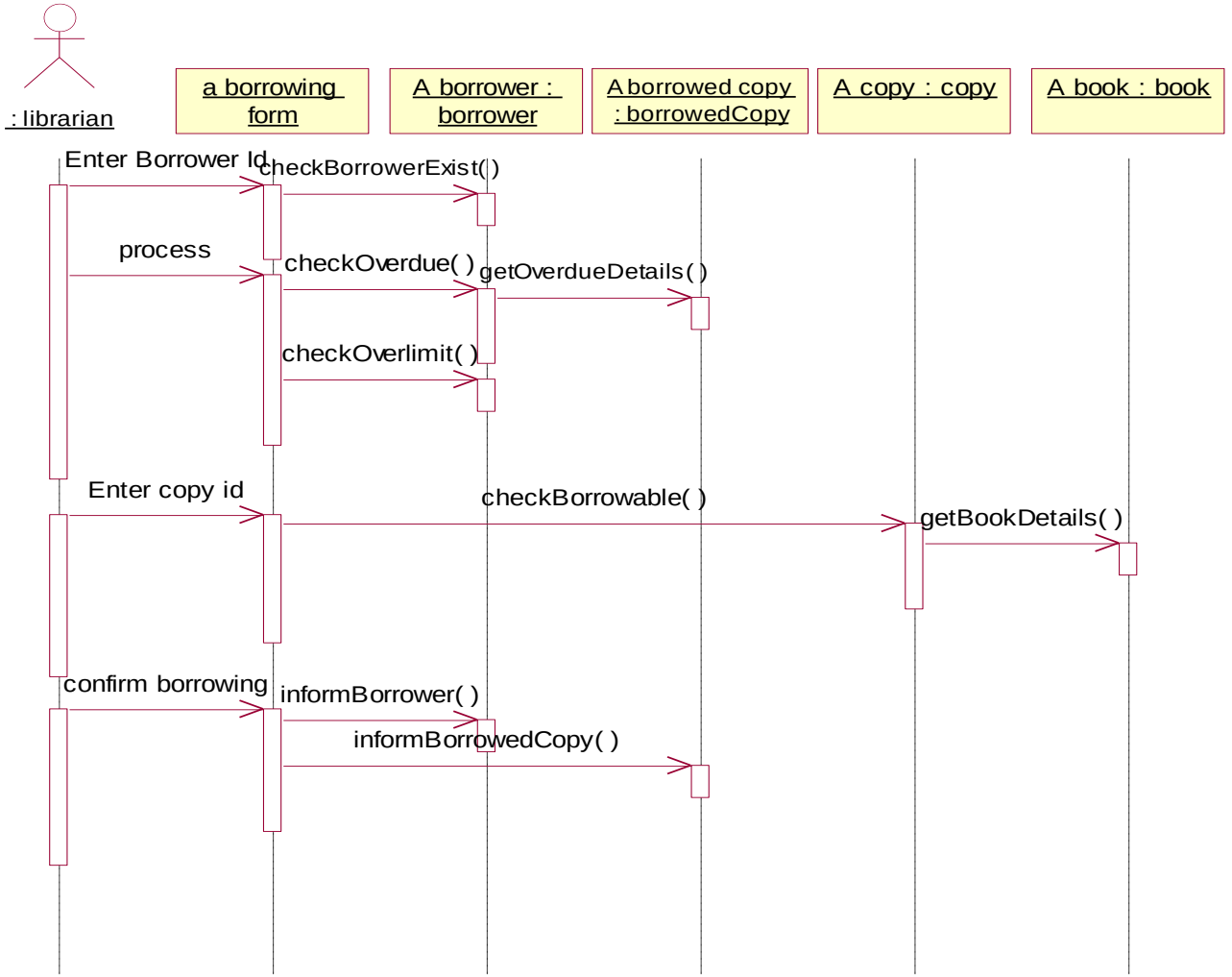
Sequence Diagrams:

- An Interaction diagram that emphasizes the time-ordering of messages.
- Graphically depicts how objects interact with each other via messages in the execution of a use case or operation.
- Illustrates how messages are sent and received between objects and in what sequence.

1.6 Introduction to Unified Modeling Language (UML)

Sequence Diagrams

e.g. Lending Books in a Library



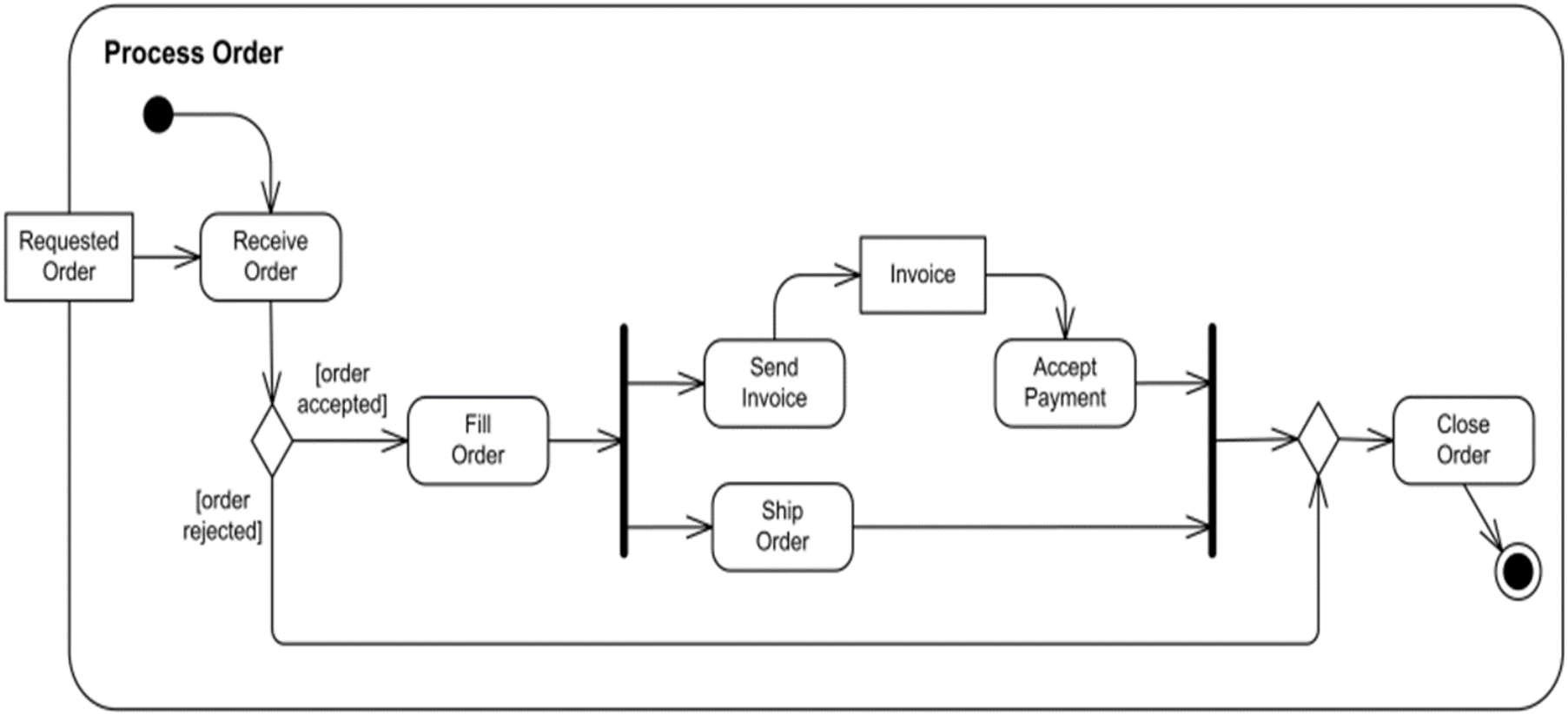
1.6 Introduction to Unified Modeling Language (UML)

Activity Diagram :

- A diagram that shows the flow from activity to activity within a system.
- Address the dynamic view of a system.

1.6 Introduction to Unified Modeling Language (UML)

Activity Diagram



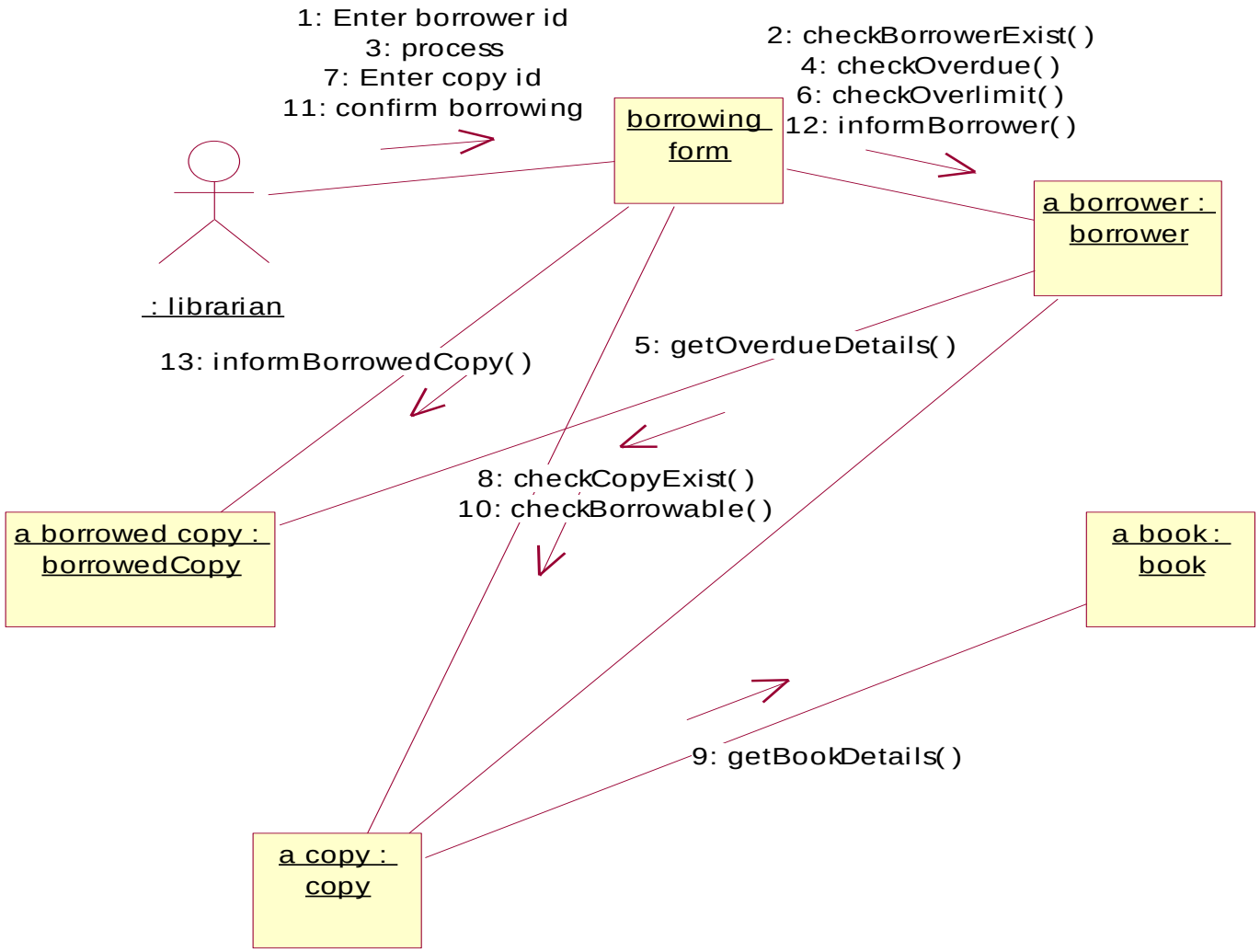
1.6 Introduction to Unified Modeling Language (UML)

Communication Diagrams :

- An interaction diagram that emphasizes the structural organization of the objects that send and receive messages.
- Shows the interaction of objects via messages.
- Also known as collaboration diagram in UML 1.X.

1.6 Introduction to Unified Modeling Language (UML)

Communication Diagrams : e.g. Lending of Books



1.6 Introduction to Unified Modeling Language (UML)

Component Diagram :

- Shows the organizations and dependencies among a set of components.

- Address the static implementation view of a system.

1.6 Introduction to Unified Modeling Language (UML)

Component Diagrams in UML 1.* and 2.*

UML1.x

UML 2.0

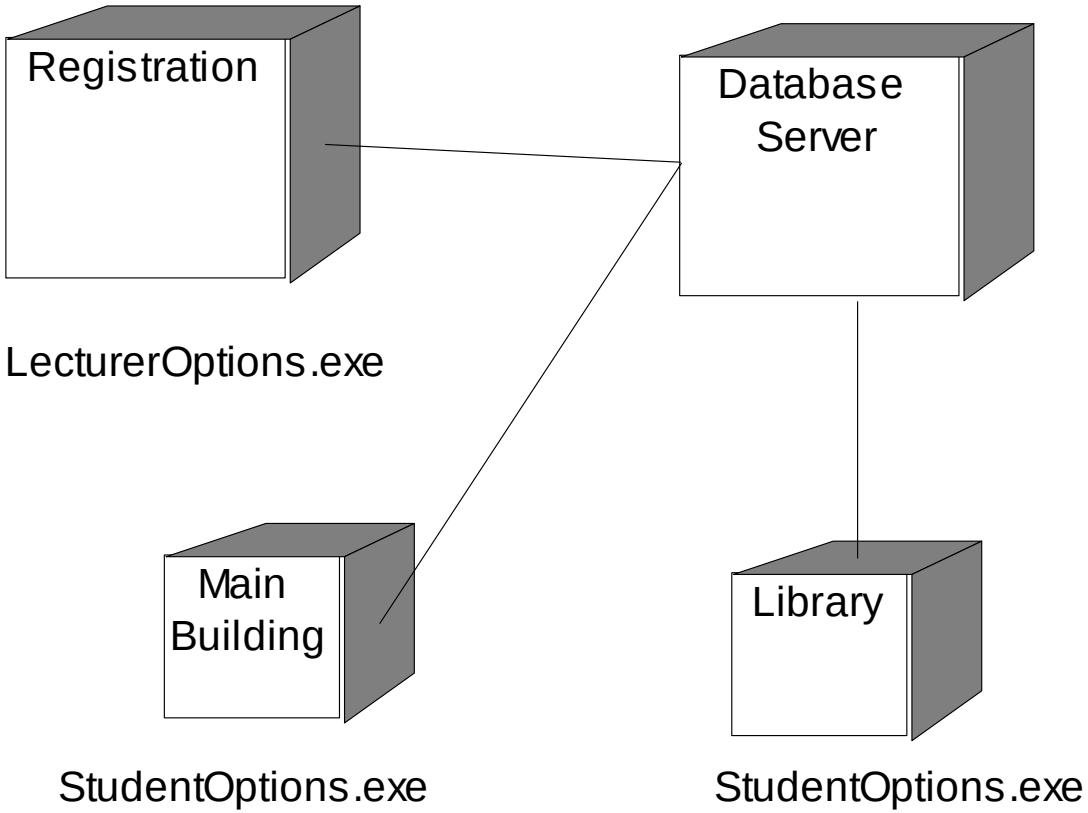
1.6 Introduction to Unified Modeling Language (UML)

Deployment Diagram

- Shows the configuration of run time processing nodes and the components live on them.
- Shows the physical architecture of a computer-based system.
- It can show computers, their connections with one another, and show the software that sits on each machine.

1.6 Introduction to Unified Modeling Language (UML)

Deployment Diagram



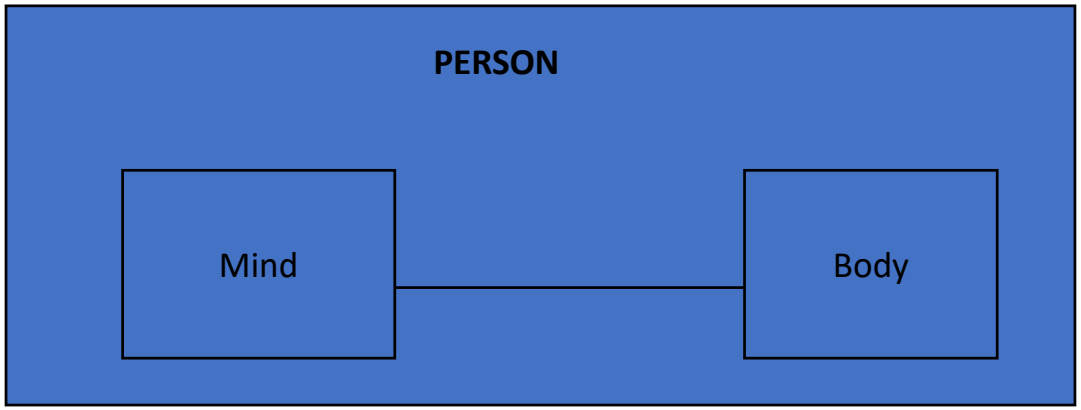
1.6 Introduction to Unified Modeling Language (UML)

Composite Structure Diagrams :

- Decomposes the internal structure of a class.
- New in UML 2.0.

- A diagram which shows something about the class's internal structure.

e.g.



1.6 Introduction to Unified Modeling Language (UML)

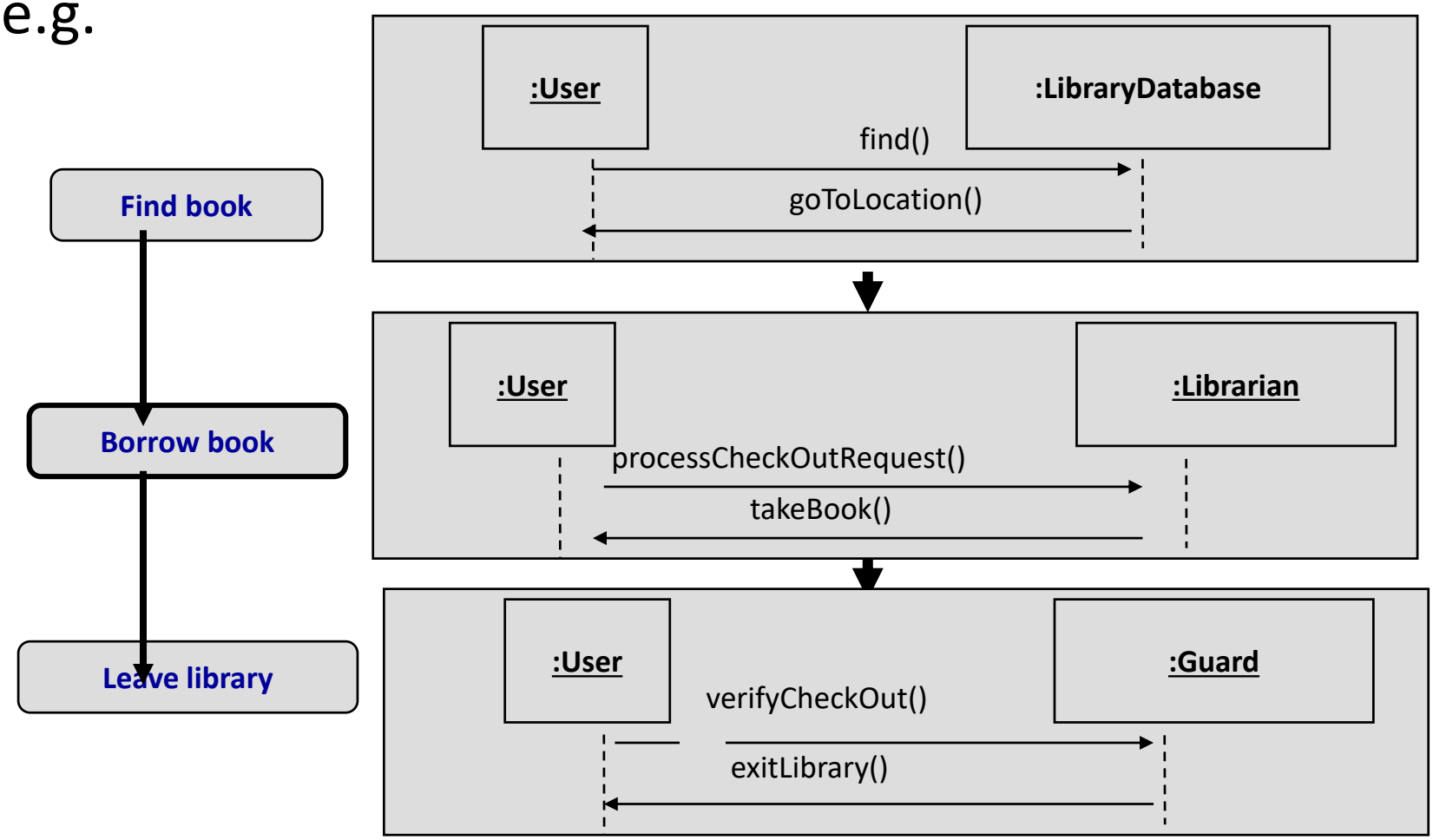
Interaction Overview Diagrams :

- New in UML 2.0.
- Replace some of the activities in the **activity** diagram with **sequence / communication** diagrams (or a combination of the two)
- Shows how objects interact within each activity of a use case.

1.6 Introduction to Unified Modeling Language (UML)

Interaction Overview Diagram

e.g.



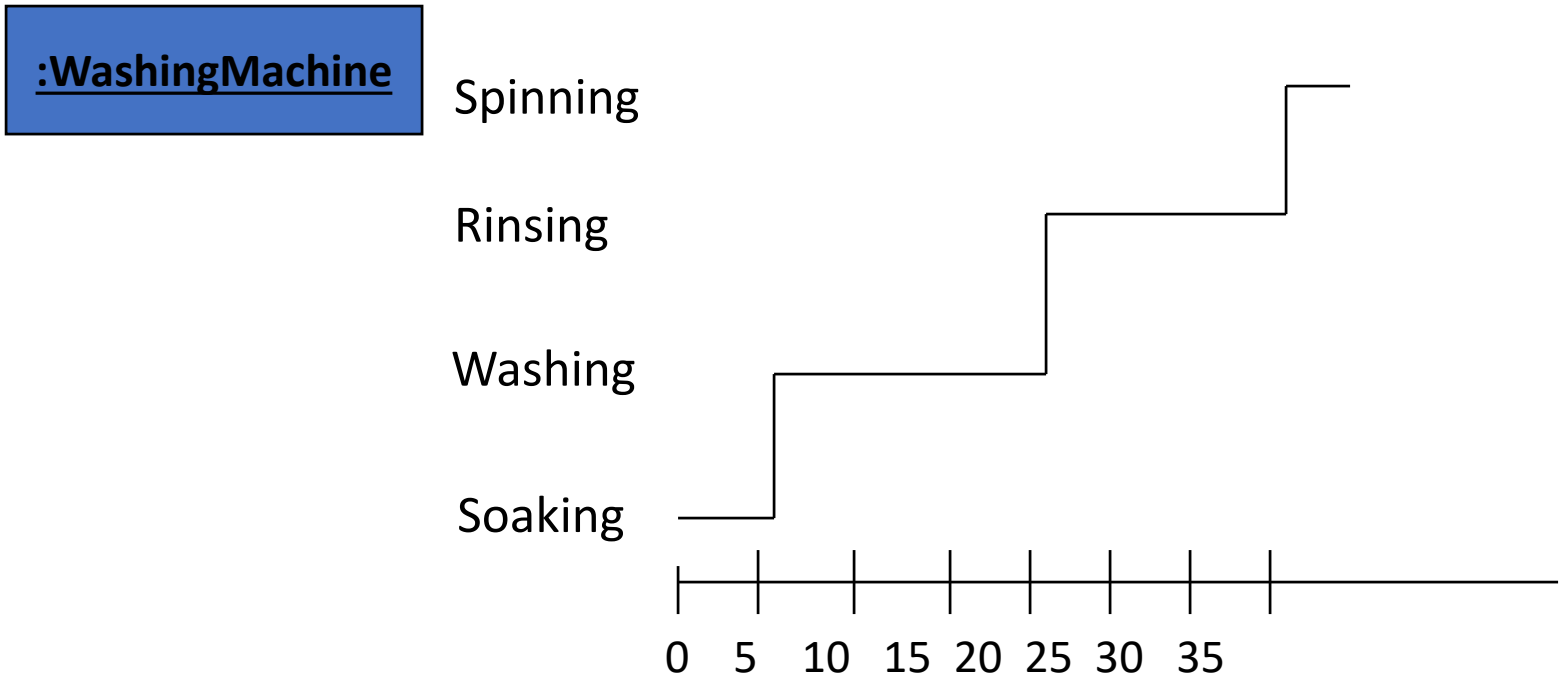
1.6 Introduction to Unified Modeling Language (UML)

Timing Diagrams :

- New in UML 2.0.
- A diagram which shows how long an object is in a particular state.
- Specially useful when designing embedded software for devices.

1.6 Introduction to Unified Modeling Language (UML)

Timing Diagram : e.g.



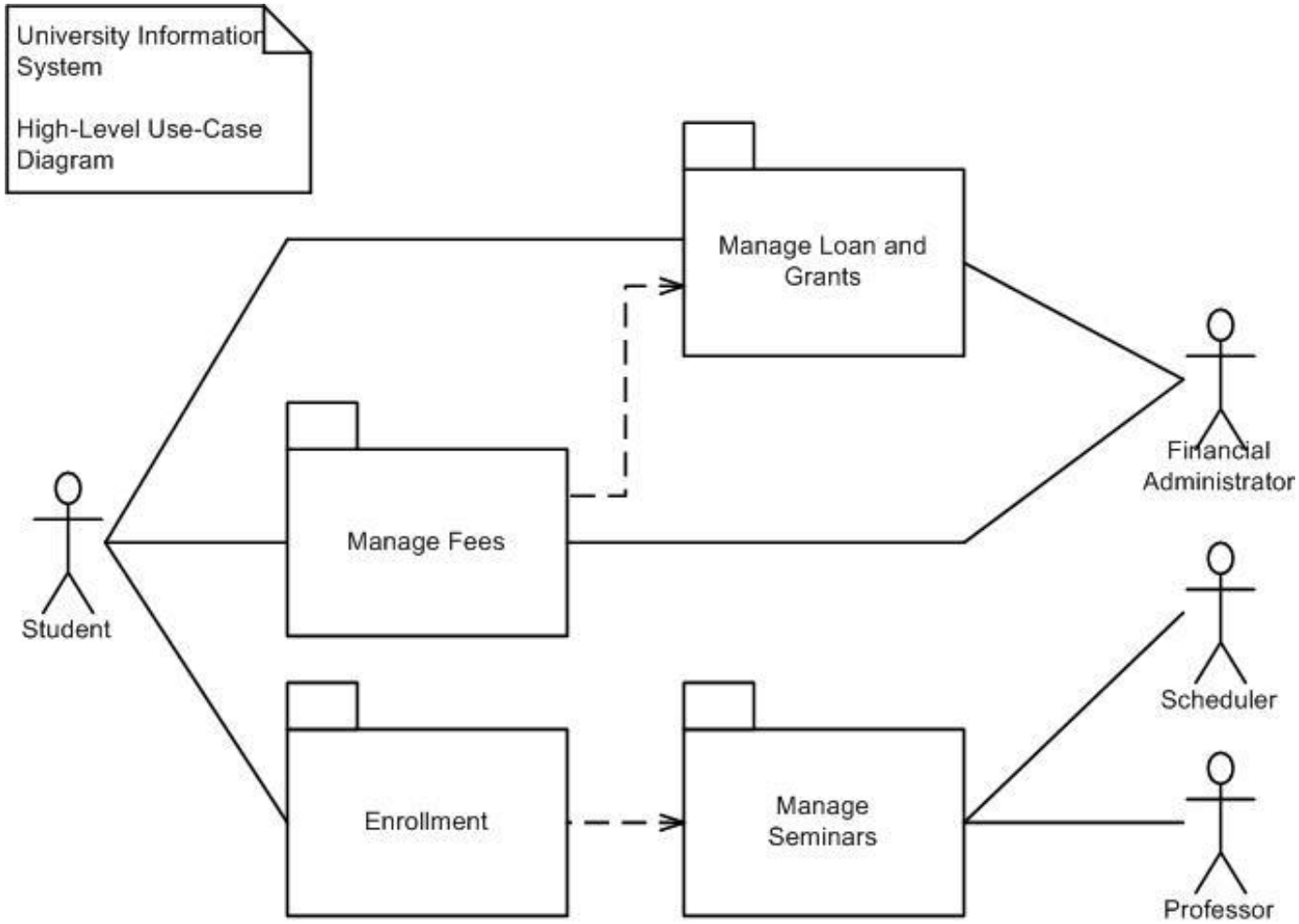
1.6 Introduction to Unified Modeling Language (UML)

Package Diagrams:

- A diagram which combines a number of classes or components into a subsystem

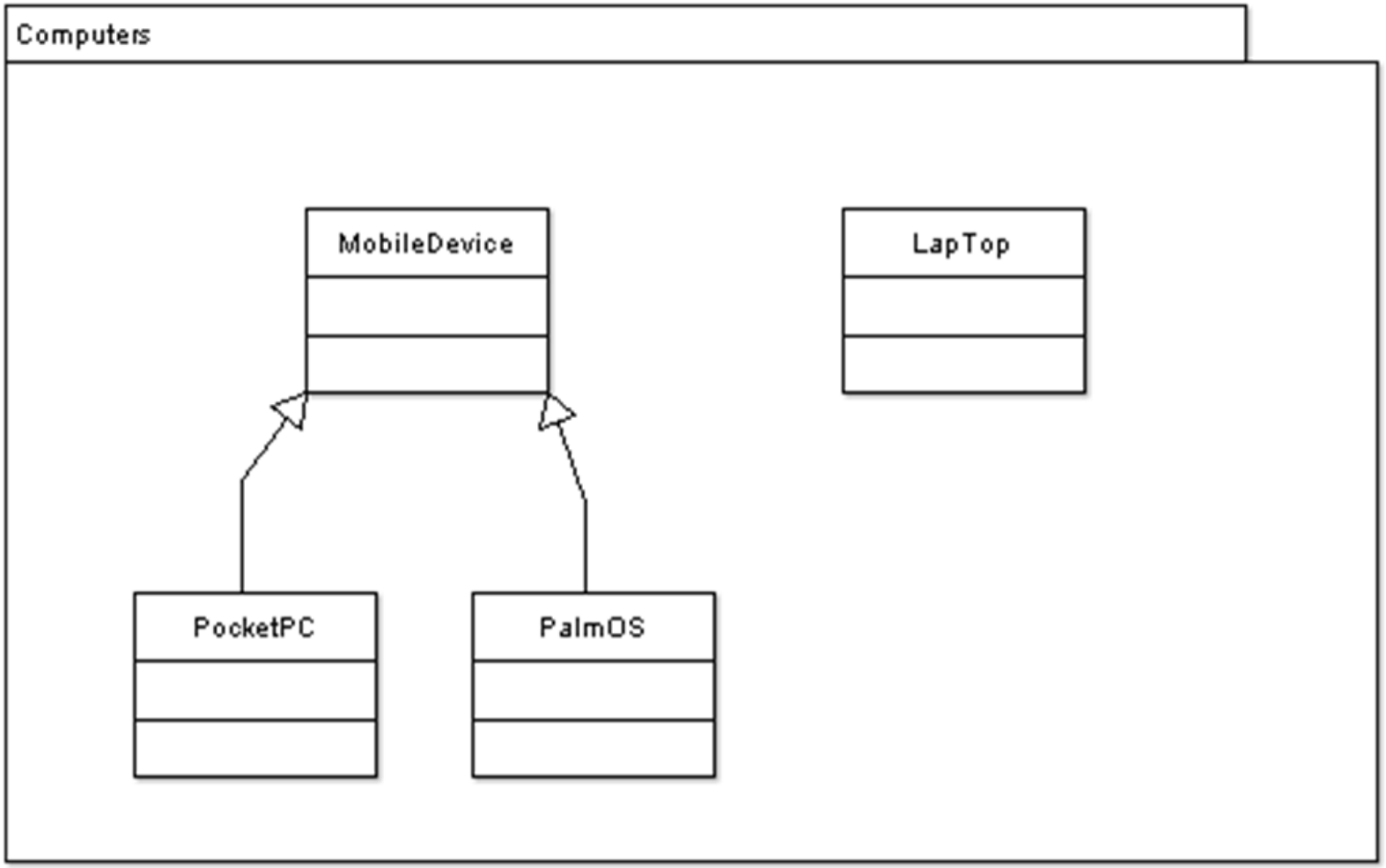
1.6 Introduction to Unified Modeling Language (UML)

Package Diagram e.g.



1.6 Introduction to Unified Modeling Language (UML)

Package in a Package Diagram



1.6 Introduction to Unified Modeling Language (UML)

Profile Diagram

Describes **lightweight extension mechanism** to the UML

Profiles allow adaptation of the UML meta-model for different:

- **platforms** (such as J2EE or .NET), or
- **domains** (such as real-time or business process modeling).

1.6 Introduction to Unified Modeling Language (UML)

Profile Diagram eg.