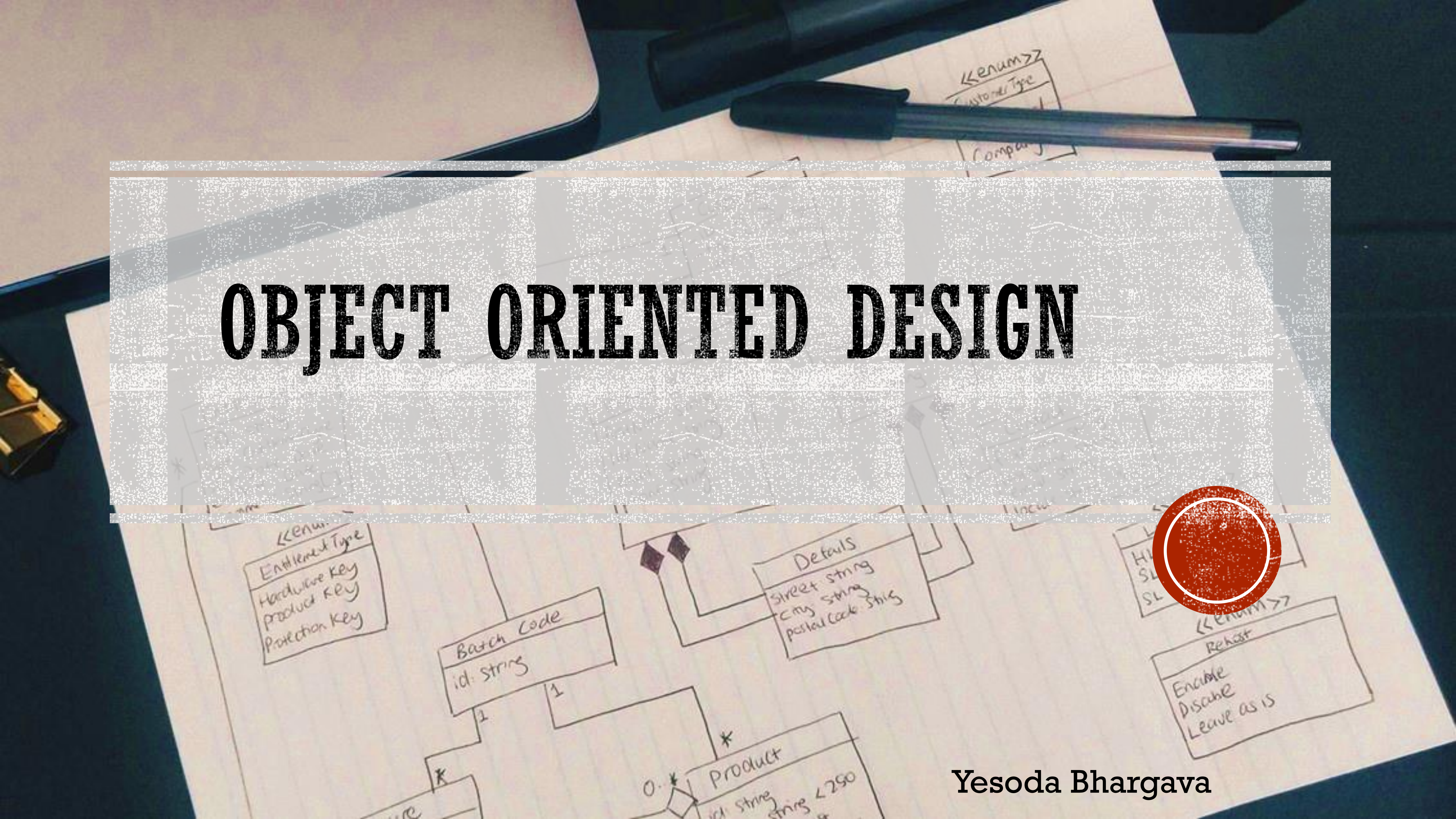


OBJECT ORIENTED DESIGN

Yesoda Bhargava

DESIGN PHASE

- Specifying the structure of how a software system will be written and function, without actually writing the complete implementation.
- A transition from "what" the system must do, to "how" the system will do it.
  - ➢ What classes we need to implement a system that meets our requirements?
  - ➢ What fields and methods will each class have?
  - ➢ How will the classes interact with each other?

BUT, WHAT EXACTLY IS A MODEL?

- A model is a simplification of reality.

WHY DO WE BUILD A MODEL?

- So that we can better understand the system we are developing.
- Modelling helps us to visualize the system as it is or as we want it to be.
- Permits us to specify the structure or behaviour of the system.
- Provides template that guides us in constructing the system.
- Model reflects the decision you have made in the process of developing your software.

FOUR PRINCIPLES OF MODELING

- Choice of model has a profound influence on how a problem is attacked and how a solution is shaped.
- Every model may be expressed at different level of precision.
- The best models are connected to reality.
- No single model is sufficient. Every non trivial system is best approached through a small set of nearly independent models.

TWO TYPES OF APPROACH TO MODELING

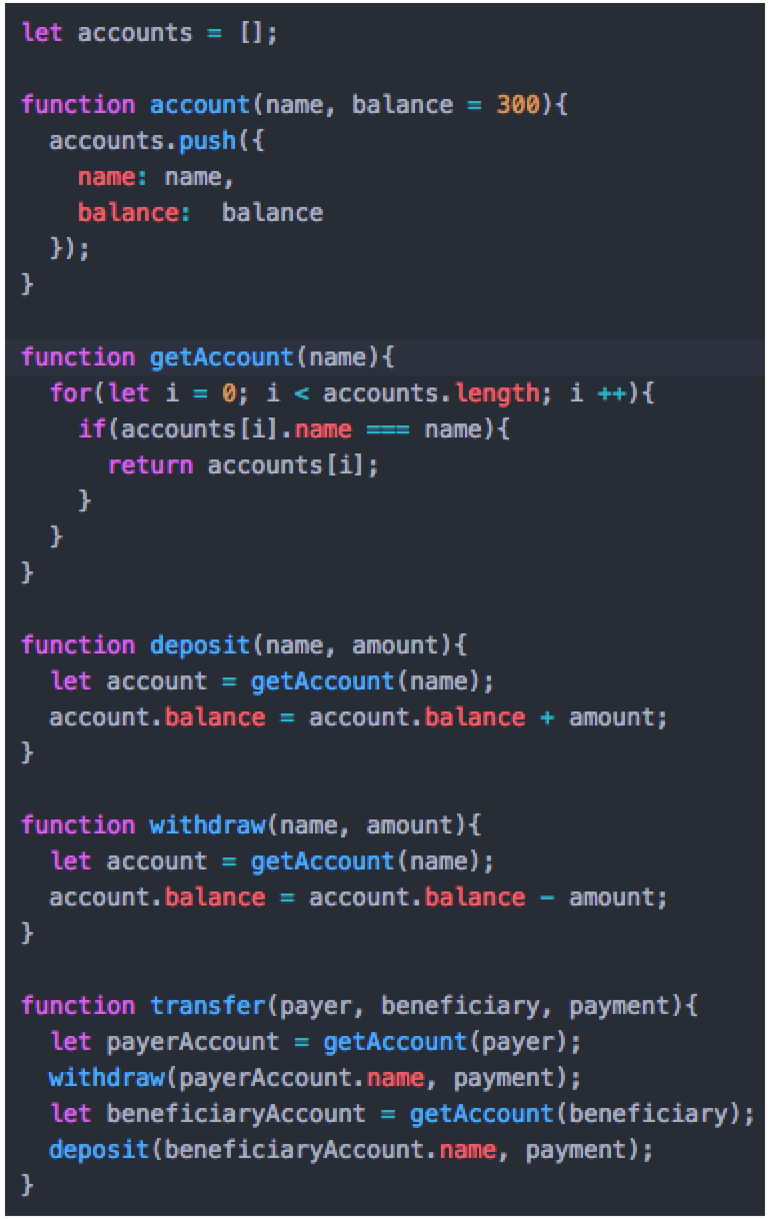
Traditional Approach

- Used procedural programming.
- Collection of procedures (function)
- Focuses on function and procedures, different styles and methodologies for each step of the process.
- Data are global and not encapsulated within any model object or with the functions.
- Moving from one phase to another phase is complex.
- With the addition of new data, all the functions have to be modified.
- Increases duration of the project.
- Increases complexity.
- Does not have a way to hide data. Less secure.

Object-Oriented System Development

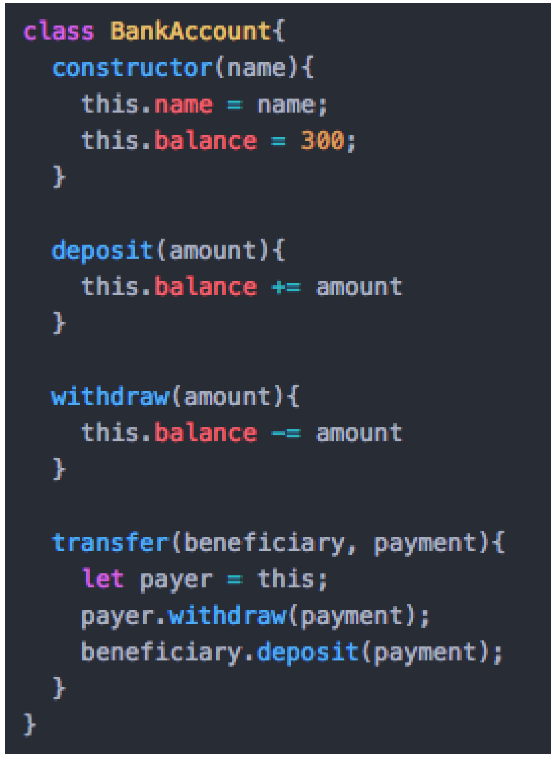
- Combination of data and functionality.
- Focuses on objects, classes, modules that can be easily replaced, modified and reused.
- Moving from one phase to another is easier.
- Decreases duration of the project.
- Reduced time to market, greater product flexibility, and schedule predictability.
- Reduces complexity and redundancy.
- Provides Data hiding.

Develop a simple Bank Account App



```
let accounts = [];

function account(name, balance = 300){
  accounts.push({
    name: name,
    balance:  balance
  });
}

function getAccount(name){
  for(let i = 0; i < accounts.length; i ++){
    if(accounts[i].name === name){
      return accounts[i];
    }
  }
}

function deposit(name, amount){
  let account = getAccount(name);
  account.balance = account.balance + amount;
}

function withdraw(name, amount){
  let account = getAccount(name);
  account.balance = account.balance - amount;
}

function transfer(payer, beneficiary, payment){
  let payerAccount = getAccount(payer);
  withdraw(payerAccount.name, payment);
  let beneficiaryAccount = getAccount(beneficiary);
  deposit(beneficiaryAccount.name, payment);
}
```

procedurally



```
class BankAccount{
  constructor(name){
    this.name = name;
    this.balance = 300;
  }

  deposit(amount){
    this.balance += amount
  }

  withdraw(amount){
    this.balance -= amount
  }

  transfer(beneficiary, payment){
    let payer = this;
    payer.withdraw(payment);
    beneficiary.deposit(payment);
  }
}
```

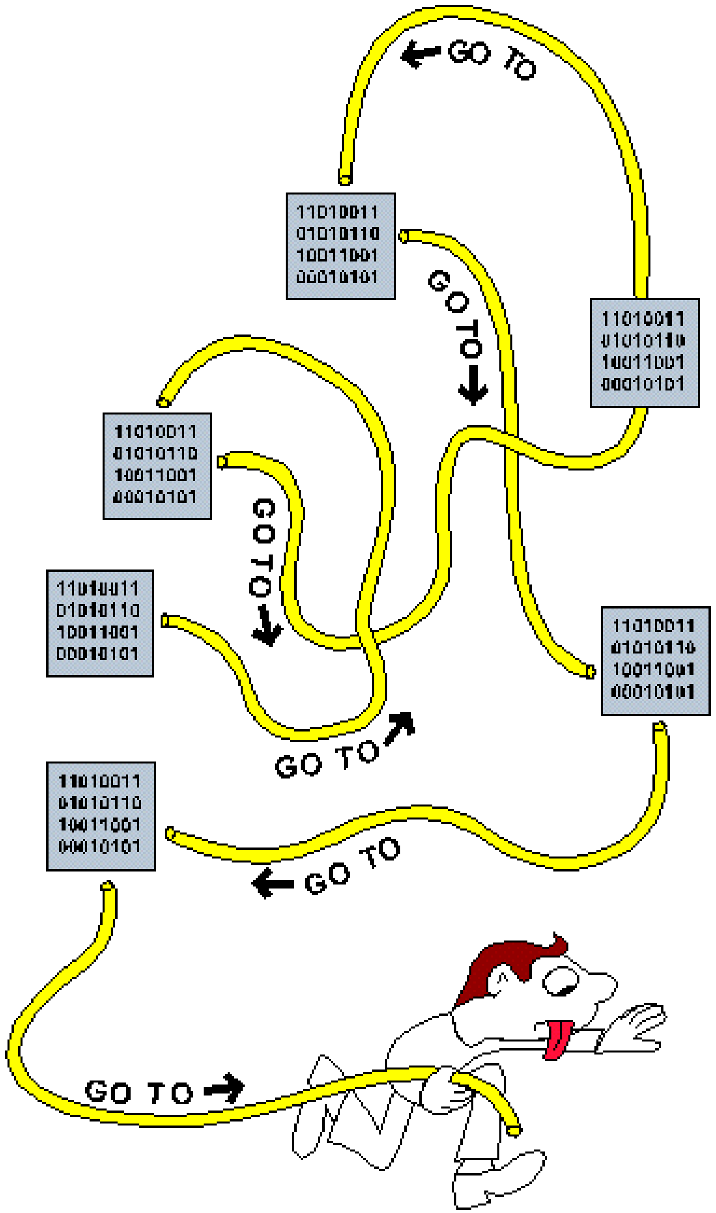
Object-oriented manner

Video link

PROCEDURAL DESIGN

- Large program is divided into smaller programs called procedures, also referred to as subprograms, subroutines, methods, or functions.
- Each function solves part of the problem.
- What happens when the size of the project increases?
- Difficult to protect the data from inadvertent changes since most of the data is generally global leading to the problem of *spaghetti code.*
- Function is the most important component of the procedural design and the data does not get the due attention.
- There are no access modifiers introduced in procedural programming.
- Suffers from lack of inheritance, absence of code reusability. No way to encapsulate data.

- Procedural programming is all about the idea of getting things done in a sequence of steps.
- This involves thinking about the functioning of your code as a step-by-step course of action that needs to be executed.
- Here, your code isn’t organized in any logical groups or object-like entities.
- You just think about the different operations that need to happen in succession and code them down.



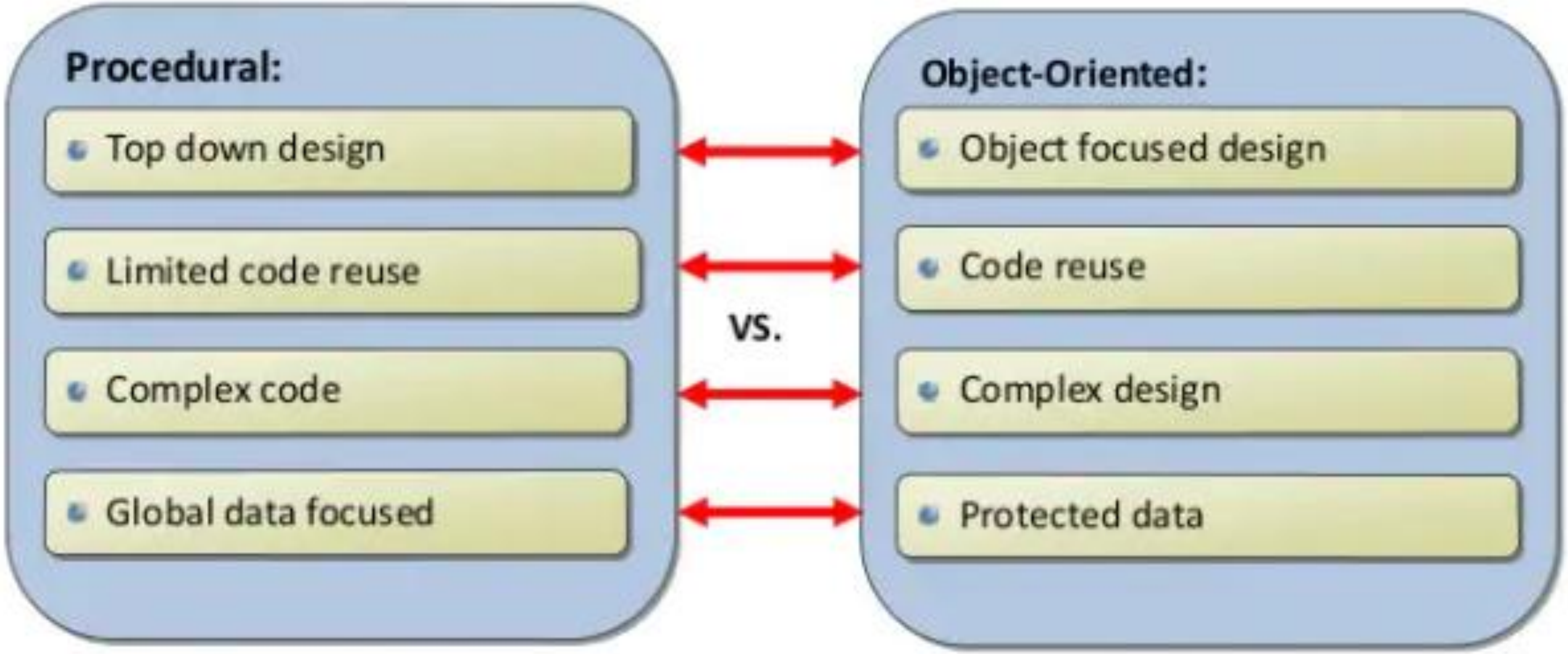
Spaghetti code/software

- Code written without a proper coherent structure.
- This usually results in somewhat unplanned, convoluted coding structures that favor GOTO statements over programming constructs, resulting in a program that is not maintainable in the long-run.

EXAMPLE

- You are working for a vehicle parts manufacturer that needs to update its online inventory system. Your boss asks you to program two similar but separate forms for a website, one form that processes information about cars and other does similar for trucks.
- Attribute of cars required: Color, Engine size, Transmission type, Number of doors.
- For trucks: Color, Engine size, Transmission type, Cab size, Towing capacity.
- Suppose you suddenly also need to add a bus form, that records the following information: Color, Engine Size, Transmission Type, Number of passengers.
- Procedural approach: An entire new forms needs to be created.
- OOP: You can extend the vehicle class with a bus class and add the method *numberOfPassengers*.

Procedural vs. Object-Oriented Programming



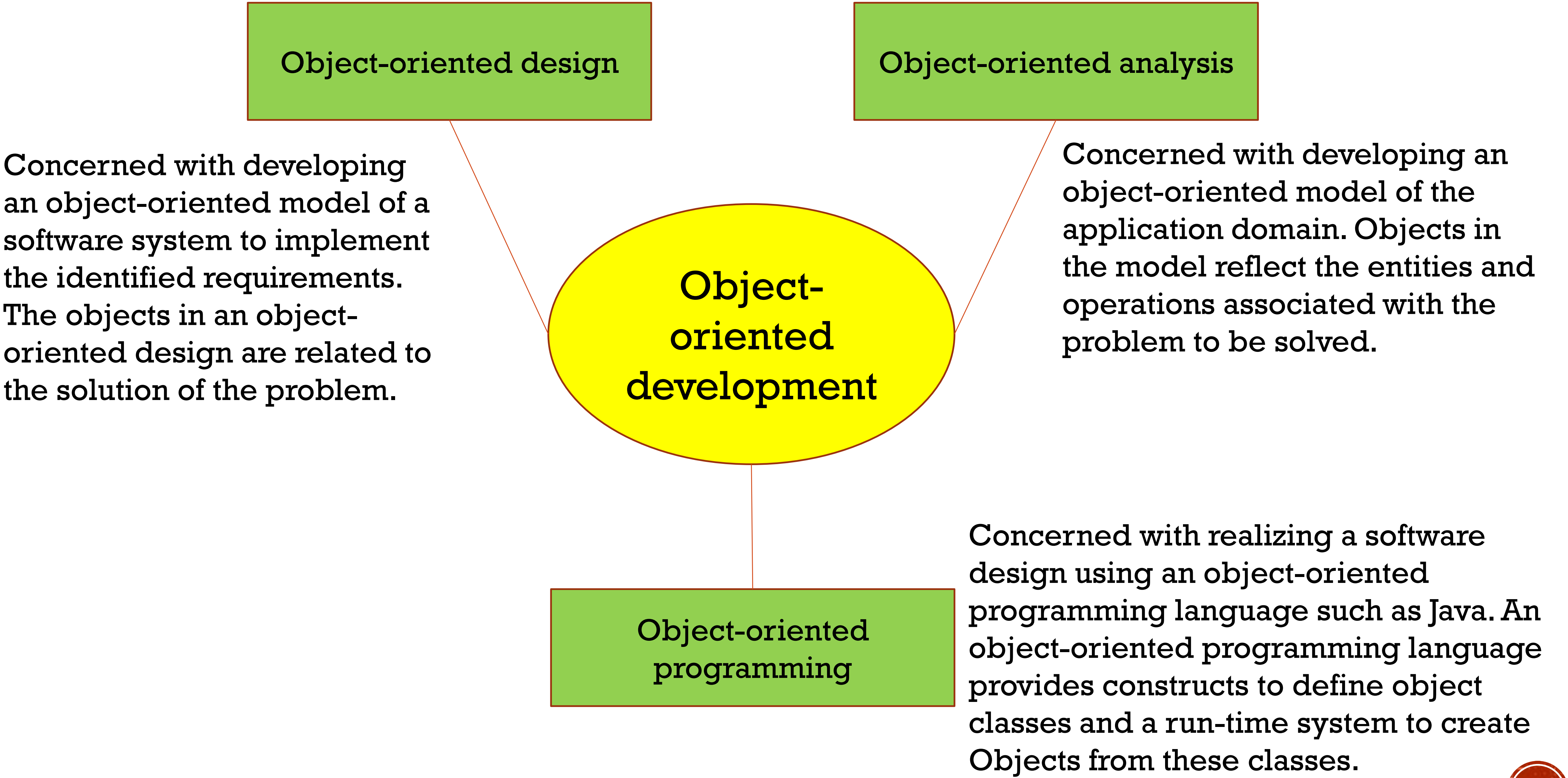
Procedural:	vs.	Object-Oriented:
Top down design		Object focused design
Limited code reuse		Code reuse
Complex code		Complex design
Global data focused		Protected data

OBJECT ORIENTED DESIGN

- **Object oriented software development**
- Before 1975, most software organizations used no specific techniques.
- Each individual worked his/her way.
- Between approximately 1975 and 1985, *structured* or *classical paradigm* was developed.
- This included structured programming and structured testing.
- But with time these proved to be less successful and less acceptable because of:
  - The technique was unable to cope with the increasing size of software products. It was suitable for software products up to 5000 lines of code. The classical techniques could not scale up to handle large products involving 5 million or more lines of codes.
  - The classical approach is either operation oriented or attribute (Data) oriented, but not both.
- Object-oriented approach considers bot attributes and operations to be equally important.
- An object incorporates both attribute and operation.

INTRODUCTION

- **Object-oriented analysis and design (OOAD)** is a recent approach to system development which is becoming popular.
- The object-oriented approach combines data and process (call methods) into a single entity called *object.*
- Objects usually correspond to the real things, an information system deals with, such as customers, suppliers, contracts, and rental agreements.
- **Object-oriented design** processes involve designing object classes and the relationship between the classes.
- These classes define the objects in the system and their interactions.



Object-oriented development

Object-oriented design

Concerned with developing an object-oriented model of a software system to implement the identified requirements. The objects in an object-oriented design are related to the solution of the problem.

Object-oriented analysis

Concerned with developing an object-oriented model of the application domain. Objects in the model reflect the entities and operations associated with the problem to be solved.

Object-oriented programming

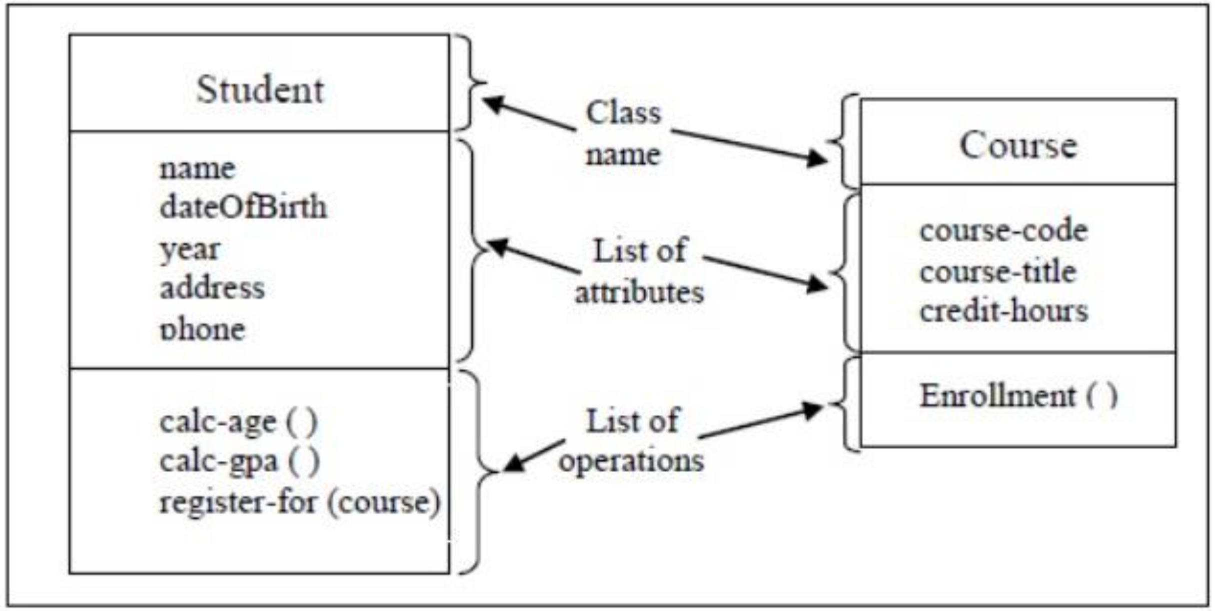
Concerned with realizing a software design using an object-oriented programming language such as Java. An object-oriented programming language provides constructs to define object classes and a run-time system to create Objects from these classes.

OBJECT ORIENTED ANALYSIS

- Looks at the problem domain to produce a conceptual model of the information that exists in the area being analysed.
- Analysis models do not consider any implementation constraints, rather they are dealt during the object-oriented design.
- The result of object-oriented analysis is a description of *what* the system is functionally required to do, in the form of a conceptual model.
- This will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams.

CLASS DIAGRAM

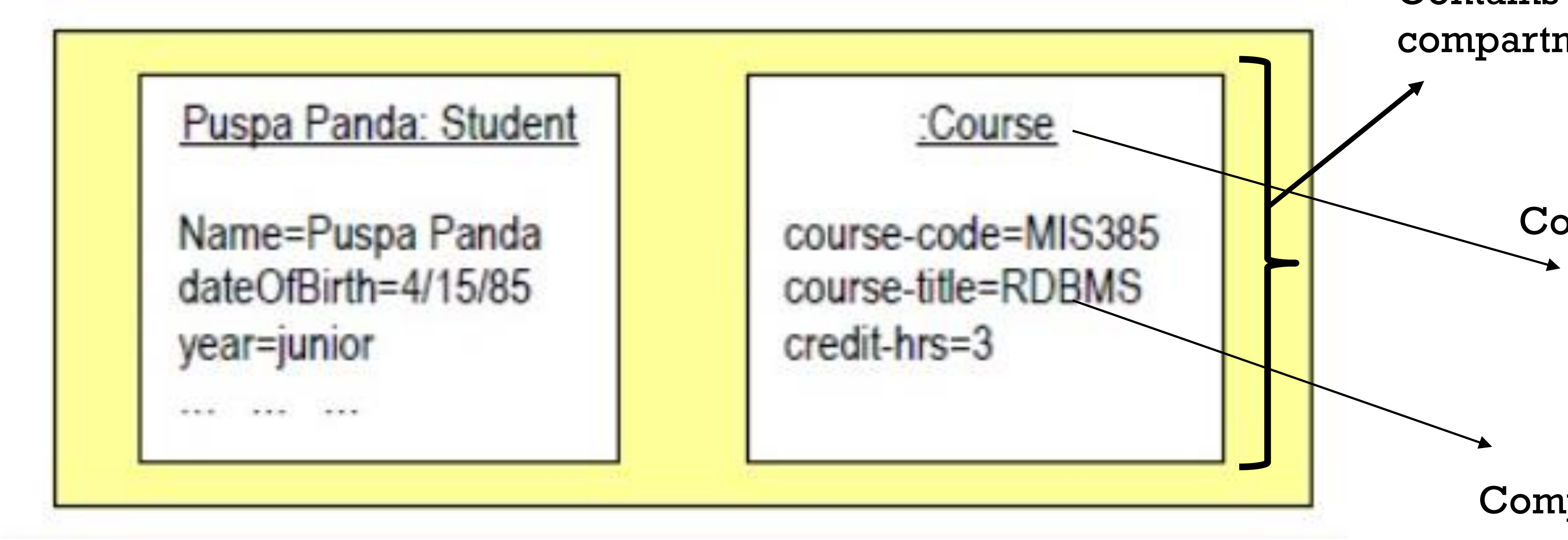
- Shows the static-structure of an object-oriented model: the object classes, their internal structure, and the relationship in which they participate.
- IN **UML,** a class is represented by a rectangle with three compartments separated by horizontal lines.
- The class name appears on the top compartment, list of attributes in them idle and the list of operations at the bottom compartment of the box.
- A class provides a template or schema for its instance. Each object knows that it belongs to the *Student* class.



Class diagram showing two classes.

OBJECT DIAGRAM

- Also known as instance diagram, is a graph of instance that are compatible with a given class diagram.
- A **static object diagram** is an instance of a class diagram.

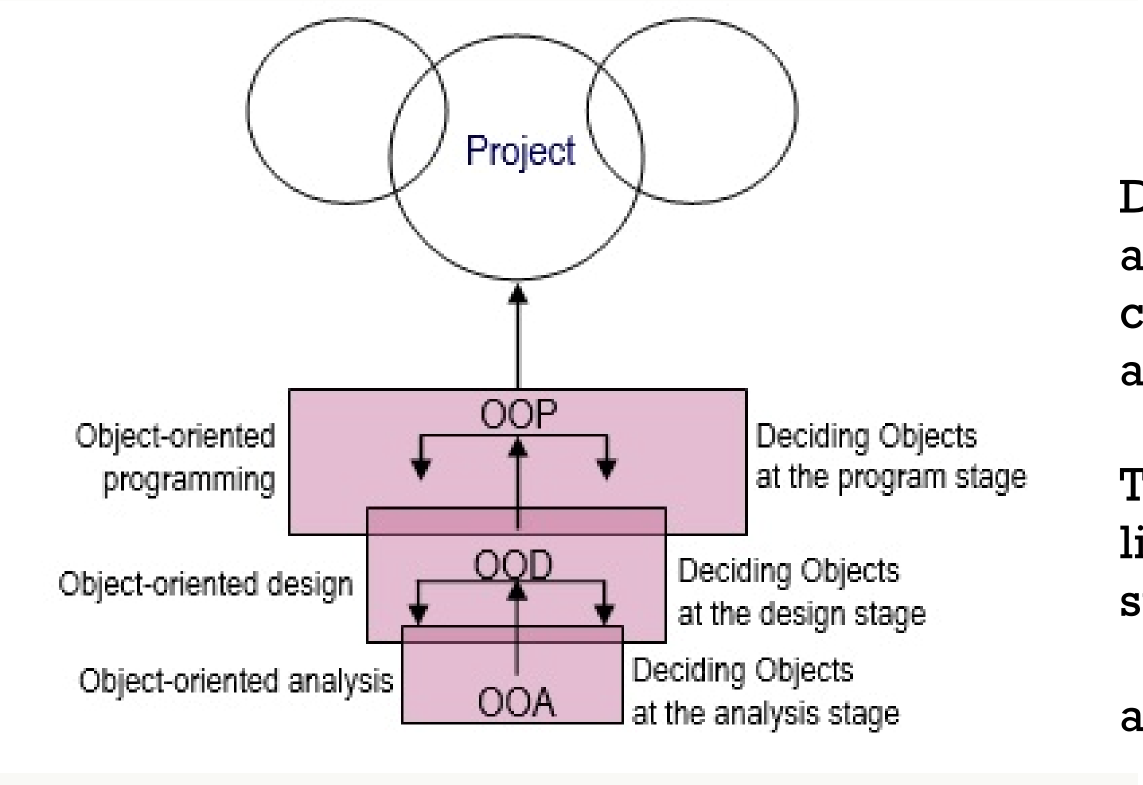


Rectangle for each object.
Contains two compartments.

Compartment 1

Compartment 2

Object diagram with two instances.


Project
OOP
Object-oriented programming
Deciding Objects at the program stage
OOD
Object-oriented design
Deciding Objects at the design stage
OOA
Object-oriented analysis
Deciding Objects at the analysis stage

Series of stages in project development

Development reaches a higher level only to fall back to a previous level and then again climbing up till completion of the project. The process contains overlap and feedback.

The project goes through an evolutionary development lifecycle containing objects decided at three different stages, namely:

a. Objects decided at the stage of object-oriented analysis (OOA).
b. Objects designed at the stage of object-oriented design (OOD).
c. Objects finalized at the stage of final programming (OOP).

- In **OOA**, objects are decided, their behaviour and their interactions meeting the requirements of the project are decided.
- In **OOD**, we draw hierarchies from which the objects can be created.
- Finally, in **OOP** we implement the programs in **C++** or any other **OOP** languages using objects.
- The object-oriented development life cycle consists of progressively developing an object representation through three phases – *analysis, design, and implementation*- similar to the systems development life cycle.

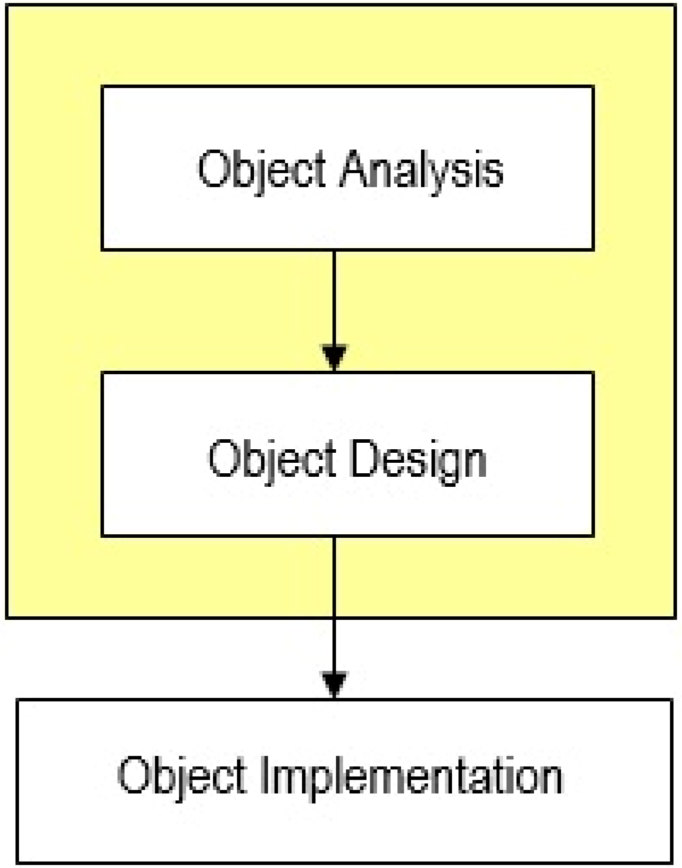
What is the difference between SDLC and OODLC?

OODLC is more like

than

In the early stages (or core) of development, the model you build is abstract, focusing one external qualities of the application system.

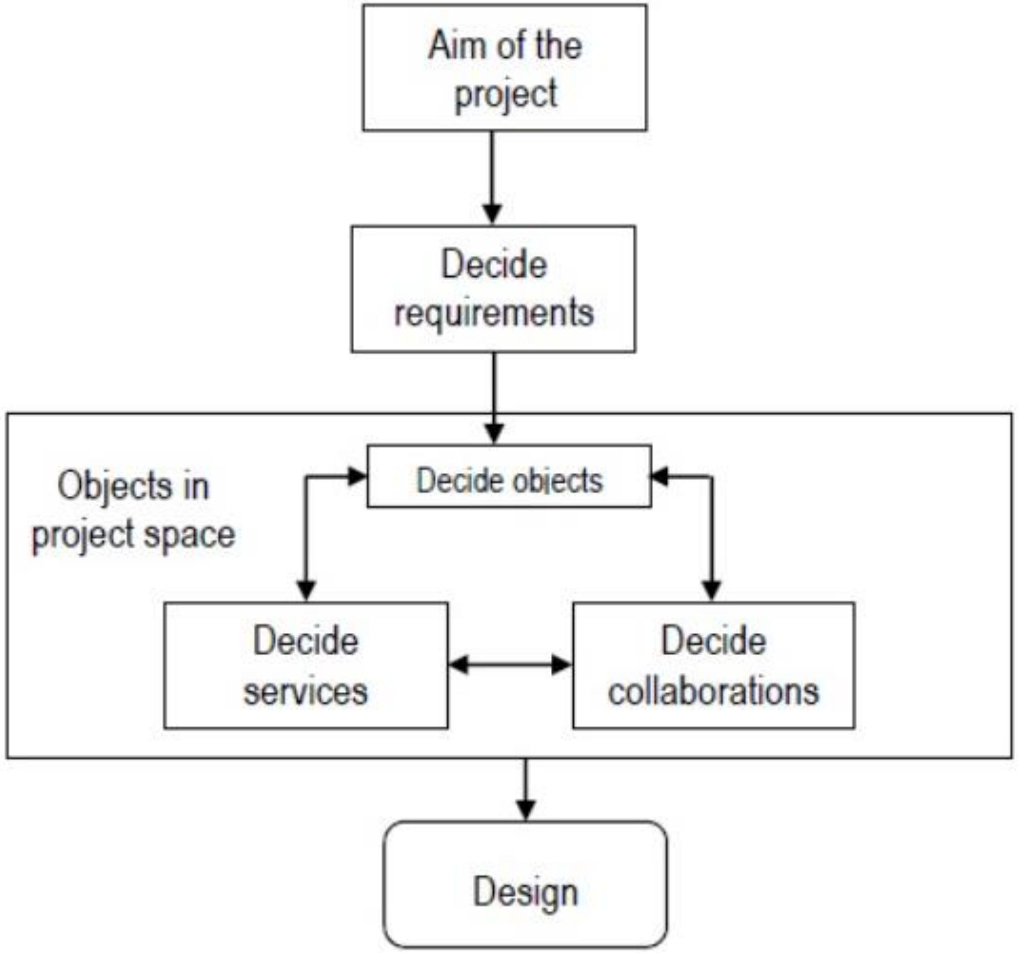
As the model evolves, it becomes more and more detailed, shifting the focus to how the system will be built and how it should function – system architecture, data structure, and algorithms.



Object-oriented life cycle

OBJECT ANALYSIS

- In Object-oriented analysis, object are identified, which are the building blocks of the project to be developed.
- Analysis is performed using these objects.
- Objects are considered to be independent entities with their own local goals.
- These independent objects are then unified to achieve the global goal of the large system.
- In OOA, following points are considered:
  a. Understand the requirements of the project.
  b. Write the specifications of the requirements of the user and the software.
  c. Decide the object and their attributes.
  d. Establish the services that each object is expected to provide.
  e. Determine interconnections among the objects in terms of services required and the services rendered.
- Steps c, d, and e, need not be in the order in which they are mentioned above, as they are interdependent.



Actions of object-oriented analysis.

IDENTIFICATION OF OBJECTS

- Identifying objects is one of the important requirements of object-oriented project analysis.
- Following criteria are used to identify objects in the system:

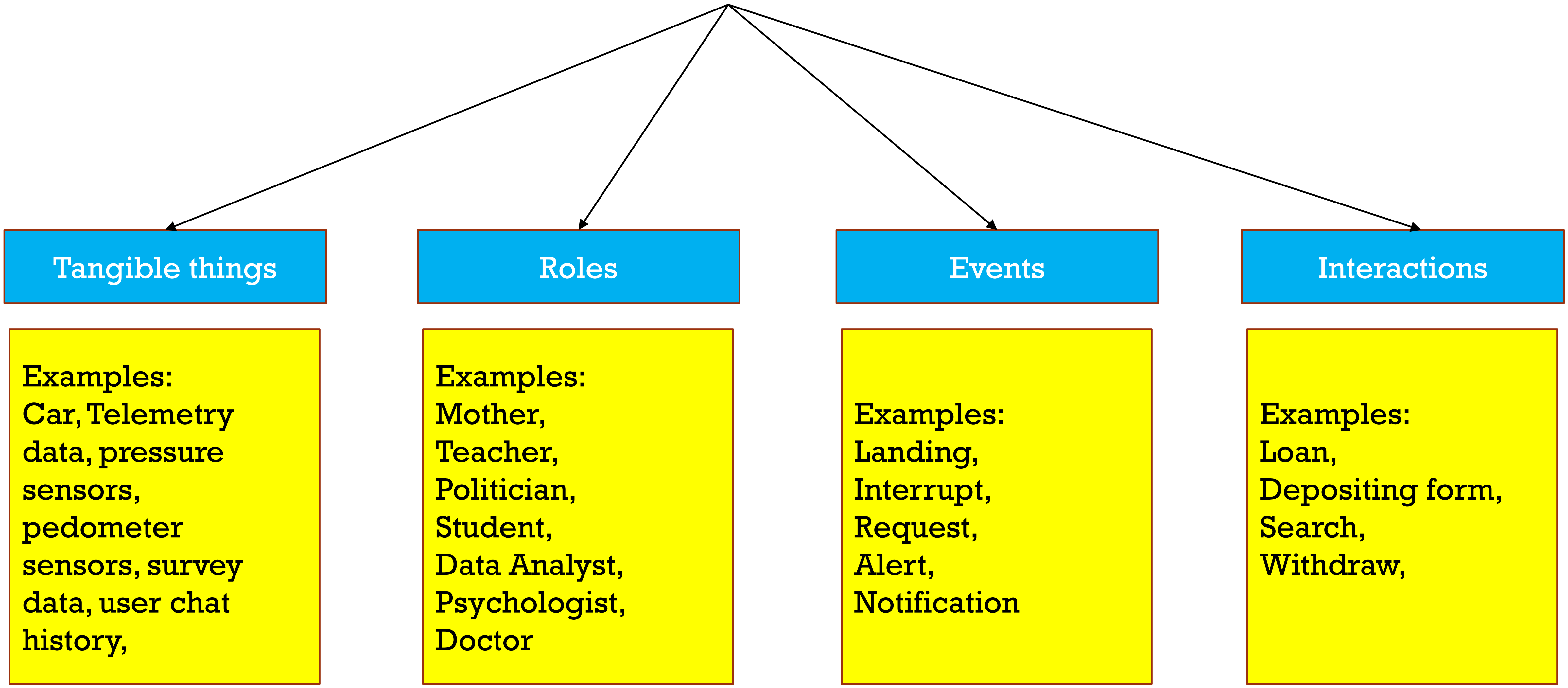
a. **An object must perform some service in the system.** You should be able to assign specific responsibilities to various objects in the system.

b. **An object must have attributes whose values are examined and used in performing the service assigned to it.** There must be several relevant attributes.

c. **An object must be essential for the functioning of the system.** This is judged by examining whether it is essential to remember information about the object.

d. There must be a common set of attributes and operations which are necessary for all occurrences of the object.

Classes and objects can come from these sources



Tangible things

Examples:
Car, Telemetry data, pressure sensors, pedometer sensors, survey data, user chat history,

Roles

Examples:
Mother,
Teacher,
Politician,
Student,
Data Analyst,
Psychologist,
Doctor

Events

Examples:
Landing,
Interrupt,
Request,
Alert,
Notification

Interactions

Examples:
Loan,
Depositing form,
Search,
Withdraw,

Financial Management System

Objects can be???

1. Investment,
2. Account
3. Transaction
4. Customer

Post object identification, you may identify their general properties and specialize them.
E.g. Types of investments (real estate, investment in shares), Types of customer (Married, Old Single, Physically challenged etc.), Types of Account (Savings, Money market, Checking Account, Certificates of Deposit)

All investments have common features like, Evaluation of return on investment and some features which are specific to each.

DETERMINING OBJECT RESPONSIBILITIES

- Post object identification, next step is to **determine each object's responsibilities, i.e., operations it can carry out on its own.**
- Then determination of operations which will be performed by multiple objects in collaboration.
- Thus, the system can be modelled as a collection of independent objects and then these objects can be allowed to communicate with each other.
- Communication may be done through transfer of messages.
- Eg. A client object sends a message requesting a service from the server object.
- The message activates a process or method on receiving the object.
- This method will carry out the required processing and return the response.

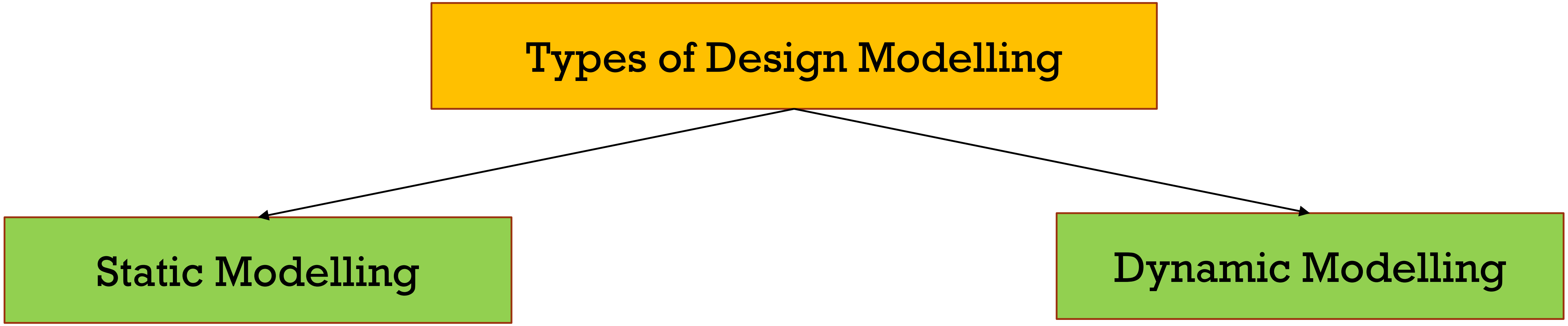
OBJECT MODELLING USING UML AND DIFFERENT SYSTEM VIEWS

UML, the Unified Modeling Language, is used to create diagrams describing the various aspects and uses your our application before you begin coding, to ensure that you have everything covered.

User's view	Structural view	Behavioural view	Implementation view
Defines the functionalities (facilities) made available by the system to its users. Internal structure, dynamic behaviour of system components, the implementation is not visible.	Defines the kinds of objects (classes) important to the understanding of the working of a system and its implementation. Captures relationship among classes (objects).	Captures how objects interact with each other to realize the system behaviour that captures the dynamic behaviour of the system.	Captures the important components of the system and its dependencies.

WHY IT IS NECESSARY TO HAVE NUMEROUS VIEWS OF A SYSTEM?

- A system has different stakeholders – people who have interests in different aspects of the system.
- E.g. Washing Machine,
- If you are designing a washing machine's motor, you have one view of the system.
- If you are writing the operating instructions, you have another.
- If you are designing the machine's overall shape, you see the system different if you just want to wash the clothes.
- Conscientious system design involves all the possible viewpoints, and each UML diagram gives a way of incorporating a particular view.
- The objective is to communicate clearly with every type of stakeholder.



Types of Design Modelling

Static Modelling

- Describes the layout of data or arrangement of stored data (i.e. data structure) but it does not show what happens to the various parts of the system.
- Deciding what the logical or physical parts of the system should be and how they should be connected together.
- How the system is constructed and initialized.

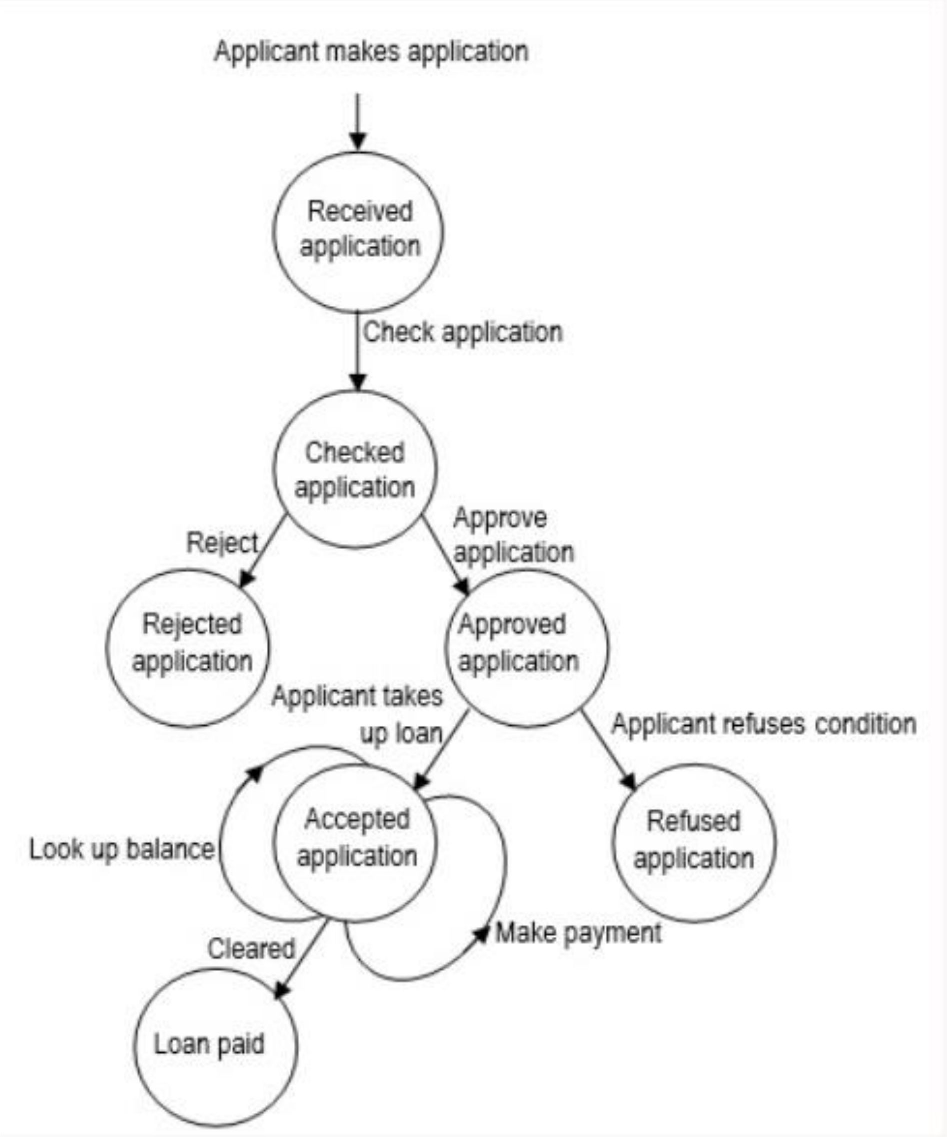
E.g. Class diagram.

Dynamic Modelling

- Represents the states of an object.
- About deciding how the static parts should collaborate.
- Aims to produce a State Chart, a description of the target similar to finite state machine.
- How the system should behave when it is running.

E.g. State Diagram, Sequence Diagram.

DYNAMIC MODELLING : THE STATE DIAGRAM



State diagram for personal loan application in bank.

The application object goes through several states, namely the received application state, the checked application state, the approved application state and finally the accepted application state.

The movement from one state to another is known as a *State transition.*

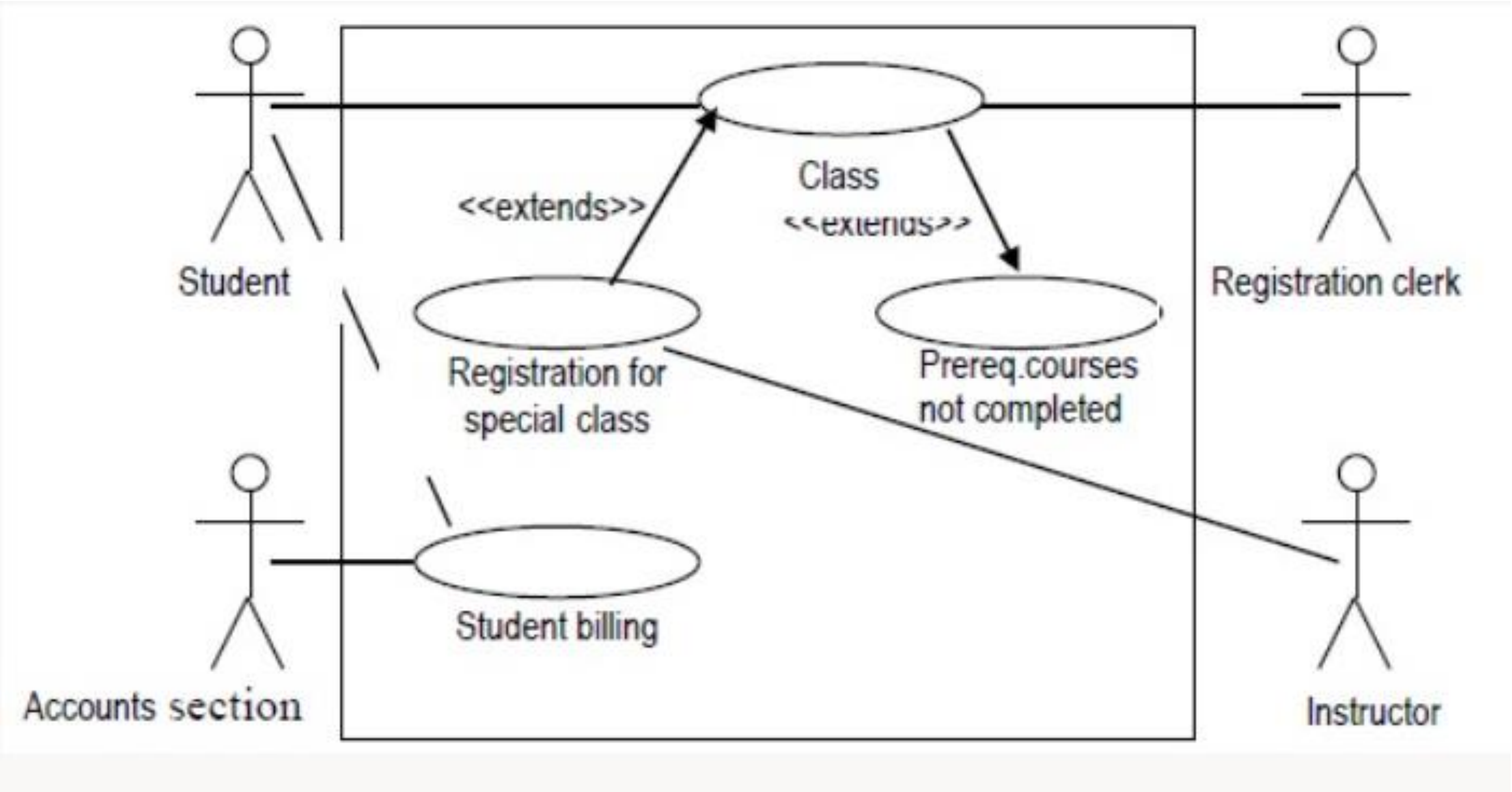
Each *state transition* is activated by an internal event or by a message from another object.

Each arrow would become a method in the application object.

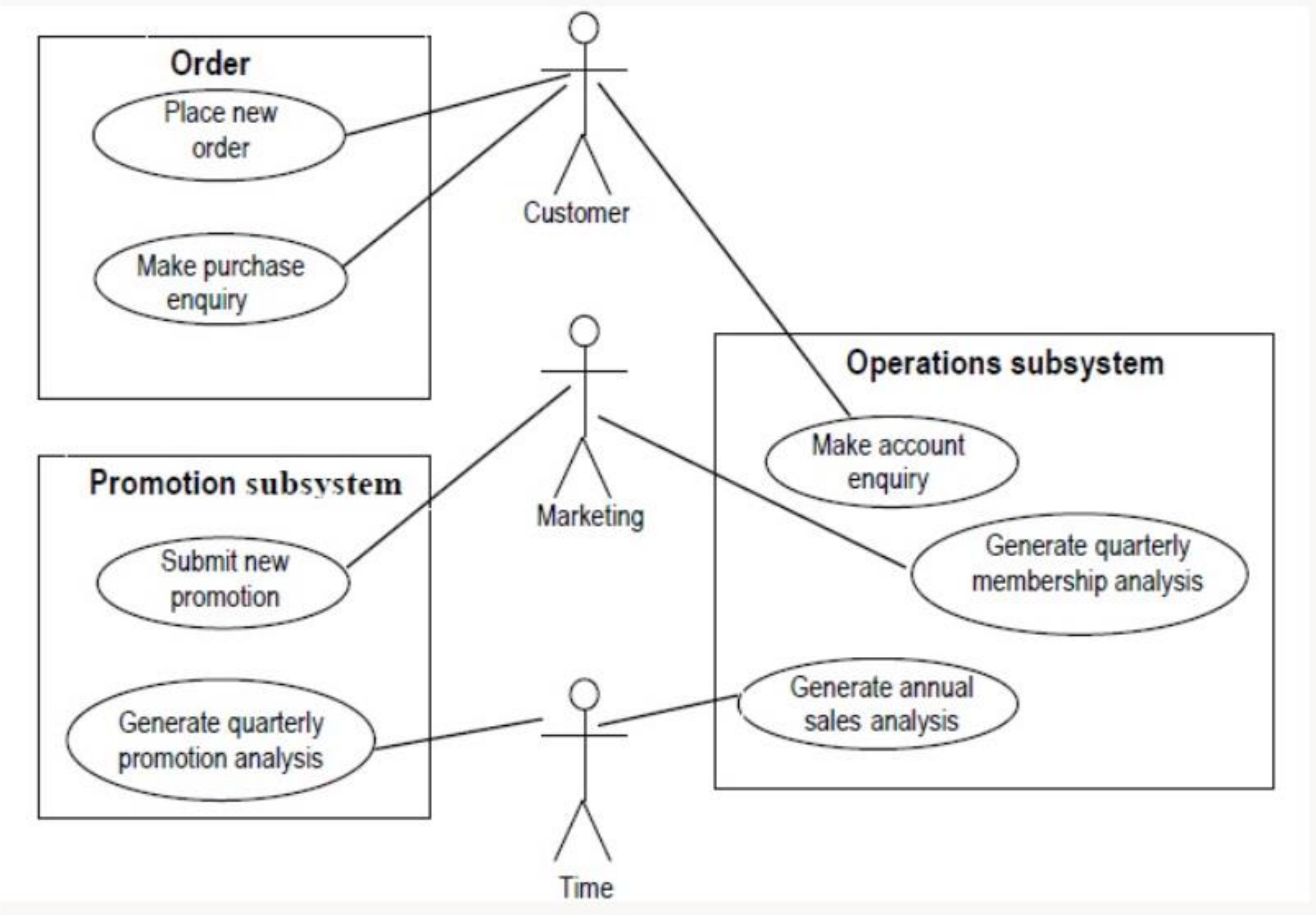
Eg. The method *accept application* or *reject application* which determine the transition from the *checked application.*

USE CASE MODELING

- Use case is a description of a system's behaviour from a user's standpoint.
- **For system developers, use case is a tool for gathering system requirements from a user's point of view**.
- Obtaining information from the user's point of view is important if the goal is to build a system that real people can use.
- Use case modelling is done in the early stages of system development to help developers gain a clear understanding of the functional requirements of the system.
- An actor is an external entity that interacts with the system.
- A *use case* represents a sequence of related actions initiated by an actor.
- Difference between actor and user? **A user is anyone who uses the system, an actor represents a role that a user can play. The actor's name should indicate that role.**
- An actor is a type or class of users.
- A user is a specific instance of an actor class.



Use case diagram for university registration



Sample use case diagram of an ordering system.

CLASS DIAGRAM

- Provide a structural view of the system.
- Capture the static structure of object-oriented systems, or how they are structured rather than how they behave.
- Class diagrams represent the basics of Object-Oriented systems.
- They identify what **classes** there are, how they **interrelate** and how they **interact**.
- Gives an overview of a software system by displaying classes, attributes, operations, and their relationships between each other.
- Class Diagram helps construct the code for the software application development.

CLASS DIAGRAM BASICS

- Classes
  - Basic class components
  - Attributes and Operations
- Class Relationships
  - Associations
  - Generalizations
  - Aggregations and Compositions

CLASS

- Think over things in the world around you.
- The things that surround you have attributes (properties) and they behave in certain ways.
- These behaviours can be seen as a set of operations.

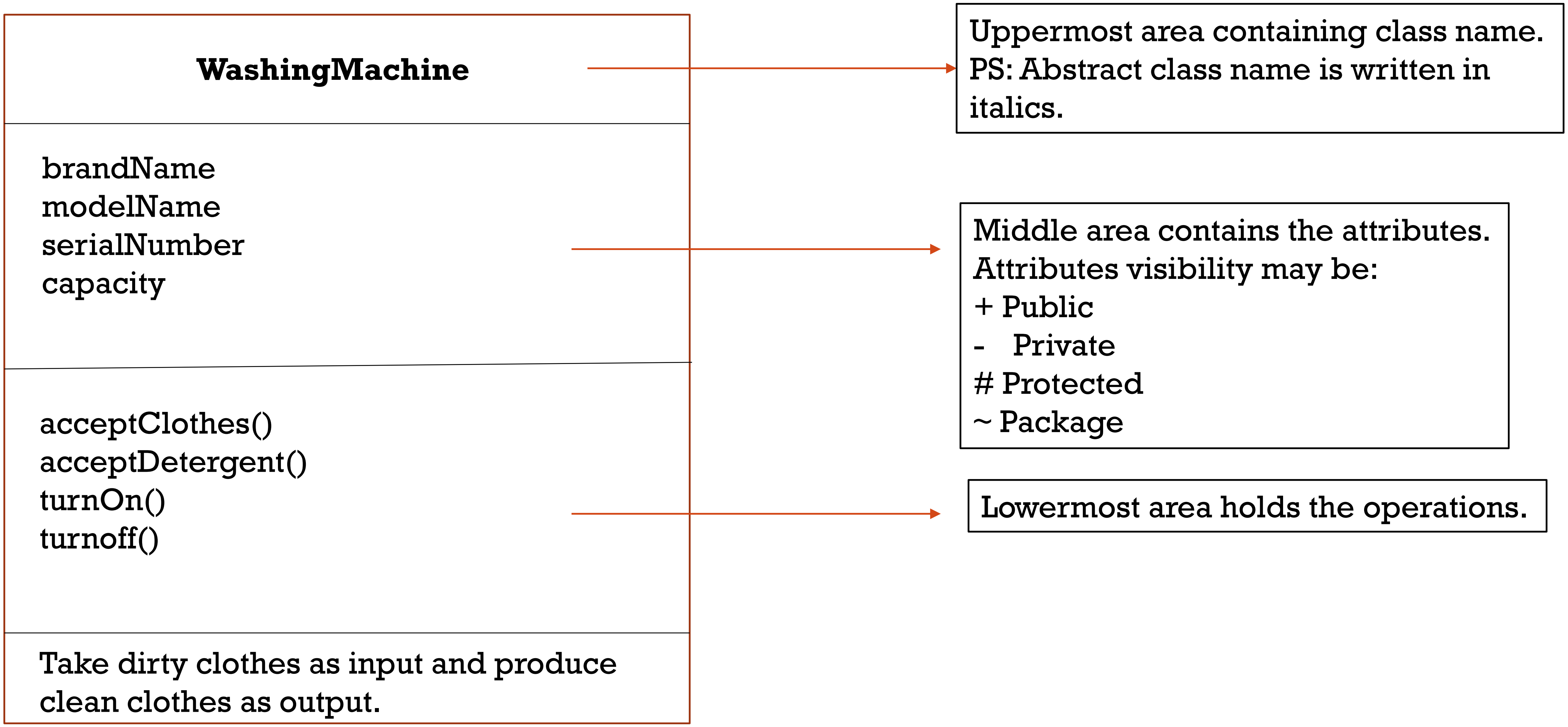
Bicycle

Watch

Washing Machine

EXAMPLE : WASHING MACHINE

- These categories in which things in the world occur are referred to as *classes.*
- Thus, a class is a category or group of things that have the same attributes and the same behaviours.
- Example: Attributes of a washing machine.
  - Brand name
  - Model
  - Serial number
  - Capacity
- Behaviours:
  - Accept clothes
  - Accept detergent
  - Turn on
  - Turn off



WashingMachine

brandName
modelName
serialNumber
capacity

acceptClothes()
acceptDetergent()
turnOn()
turnoff()

Take dirty clothes as input and produce clean clothes as output.

Uppermost area containing class name. PS: Abstract class name is written in italics.

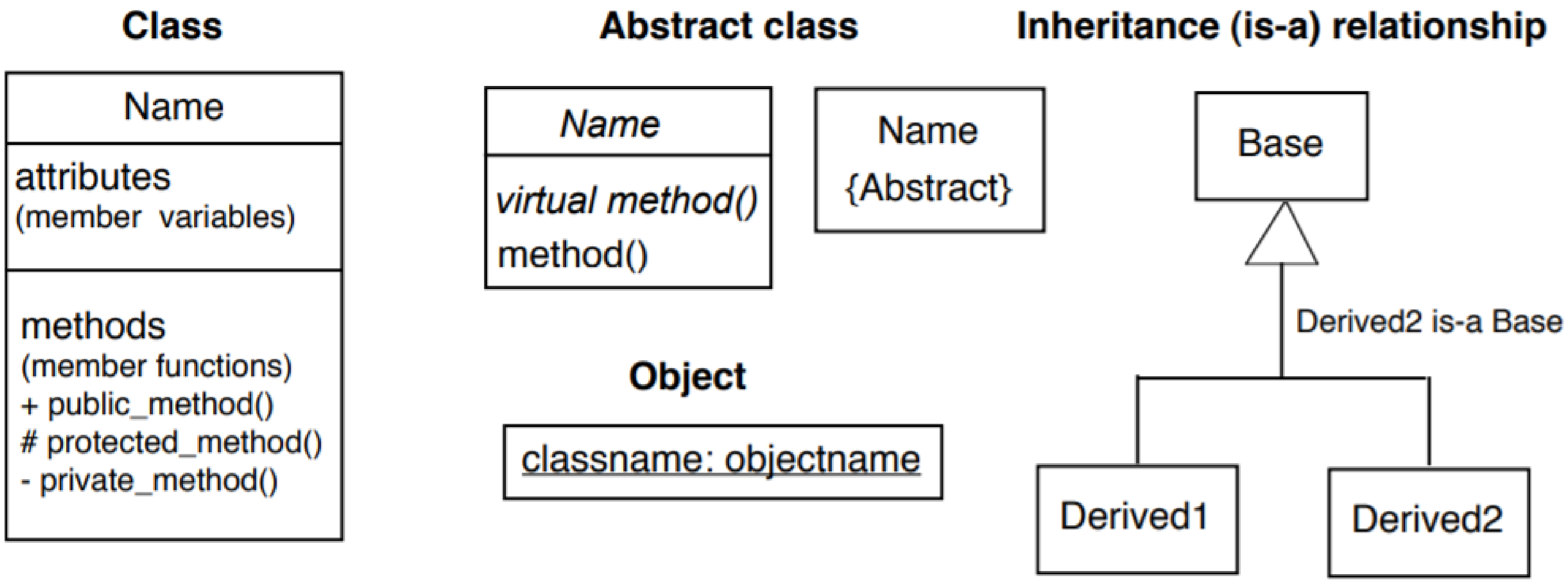
Middle area contains the attributes. Attributes visibility may be:

+ Public
- Private
# Protected
~ Package

Lowermost area holds the operations.

Representation of class icon of WashingMachine class.

BASIC UML CLASS DIAGRAM NOTATION



Class

Name

attributes
(member variables)

methods
(member functions)
+ public_method()
# protected_method()
- private_method()

Abstract class

Name

virtual method()
method()

Name
{Abstract}

Object

classname: objectname

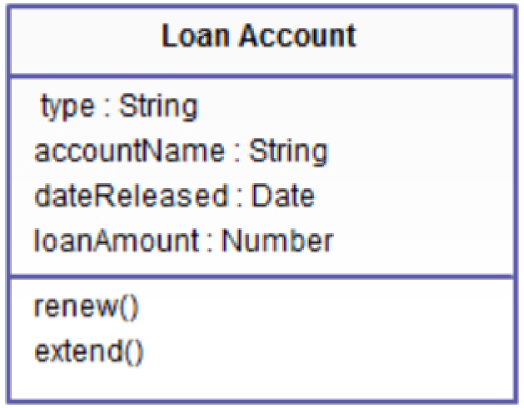
Inheritance (is-a) relationship

Base

Derived2 is-a Base

Derived1

Derived2

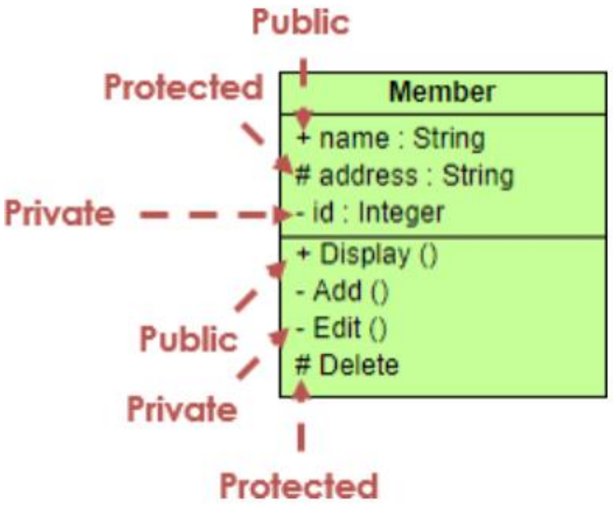


Loan Account

type : String
accountName : String
dateReleased : Date
loanAmount : Number

renew()
extend()

Simple class diagram with attributes and operations

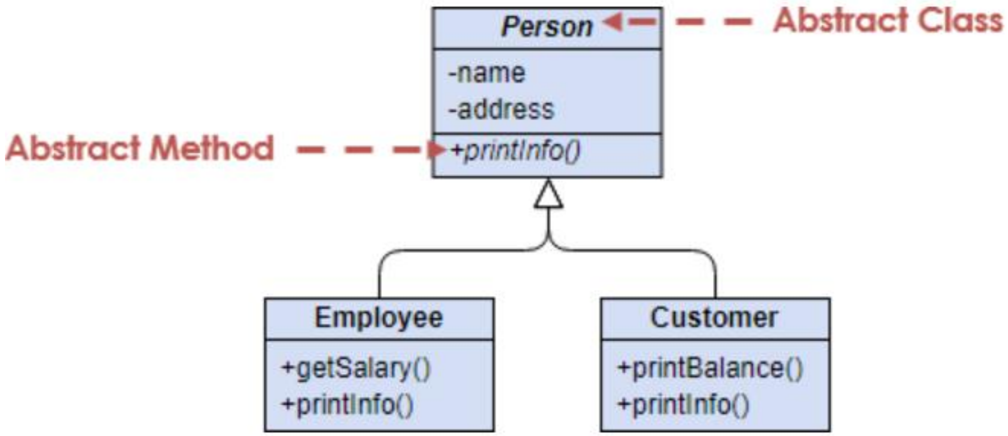


Member

+ name : String
# address : String
- id : Integer

+ Display ()
- Add ()
- Edit ()
Delete

Public
Protected
Private
Public
Private
Protected



Person

-name
-address

+*printInfo()*

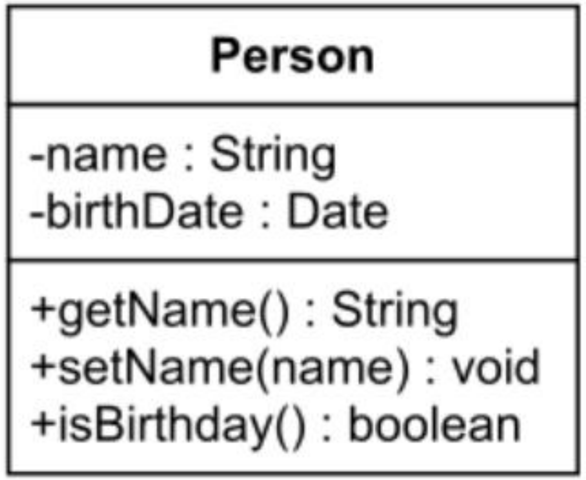
Employee

+getSalary()
+printInfo()

Customer

+printBalance()
+printInfo()

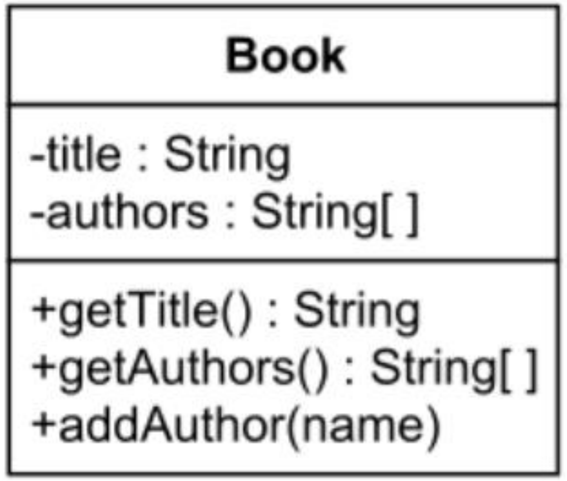
Abstract Class
Abstract Method



Person

-name : String
-birthDate : Date

+getName() : String
+setName(name) : void
+isBirthday() : boolean

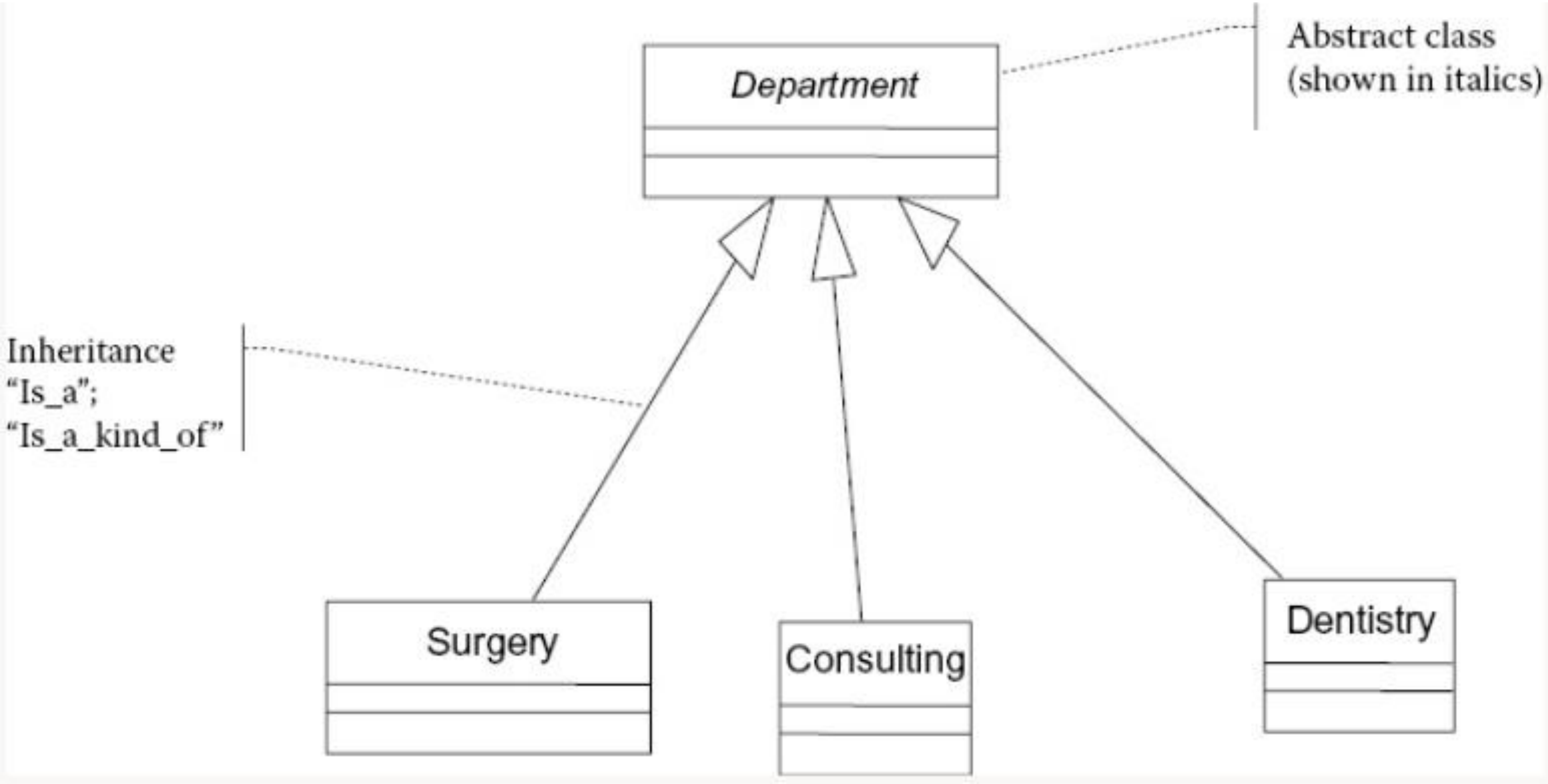


Book

-title : String
-authors : String[ ]

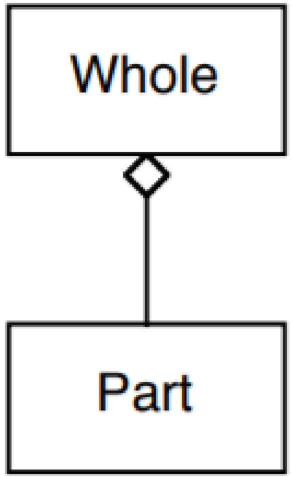
+getTitle() : String
+getAuthors() : String[ ]
+addAuthor(name)

INHERITANCE RELATIONSHIP

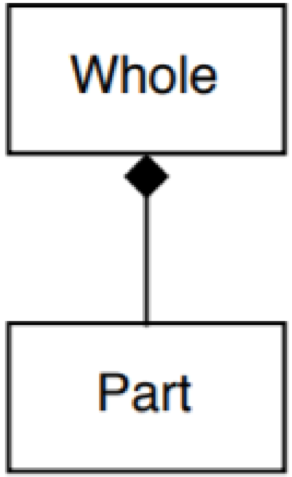


If two classes in a model need to communicate with each other, there must be a connection between them. This connection can be represented by an association connector in UML.

Aggregation and Composition (has-a) relationship



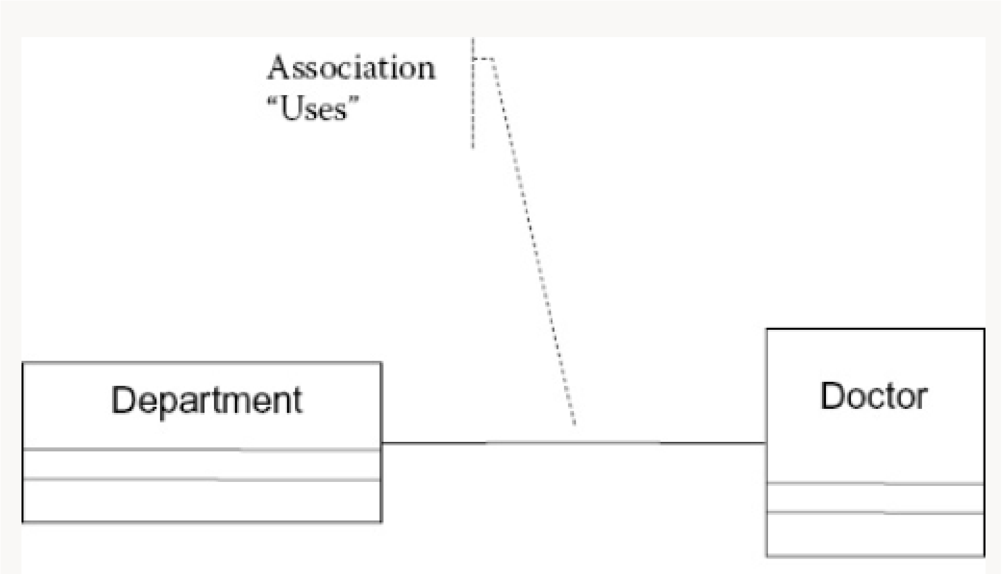
Whole has Part as a part;
lifetimes might be different;
Part might be shared with other Wholes.
(aggregation)



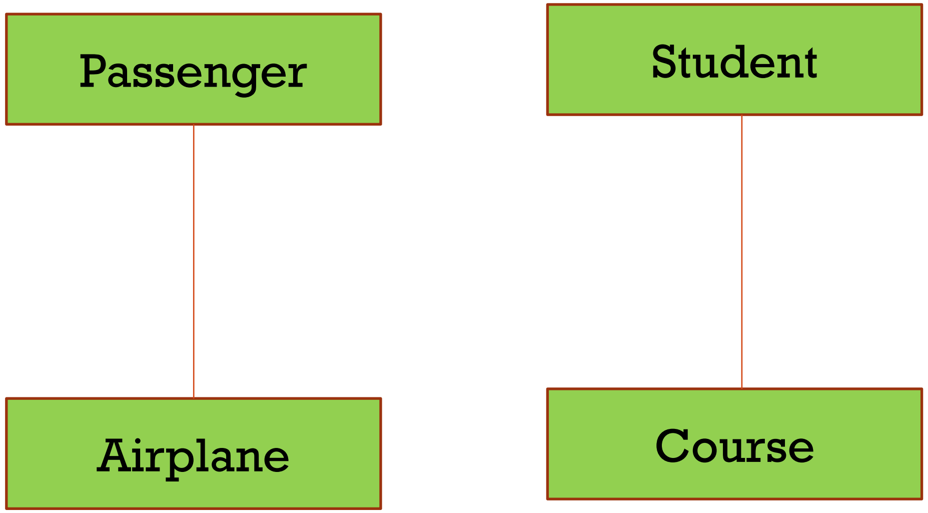
Whole has Part as a part;
lifetime of Part controlled by Whole,
Part objects are contained in one Whole object.
(composition)

ASSOCIATION RELATIONSHIP

- Can be categorized as "uses" – because they represent one class using another class in some way.
- The most basic and the most common relationship between two peer classes in object-oriented design.
- Most associating classes work together to achieve a goal in the system.

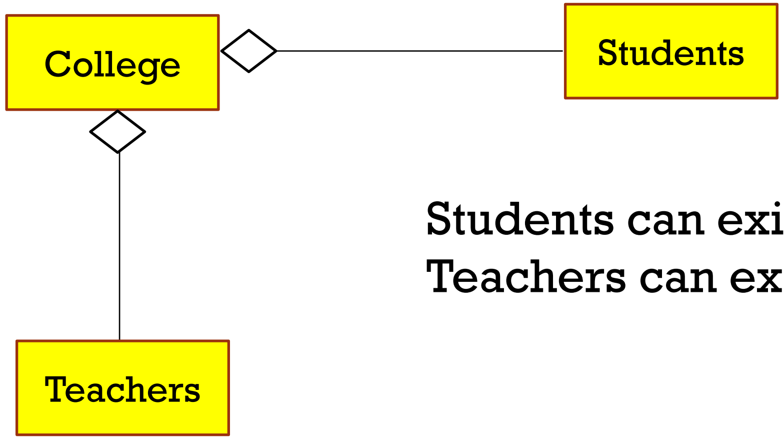


Doctor *uses* Department; Department also uses Doctor; The Relationship is *loose*.

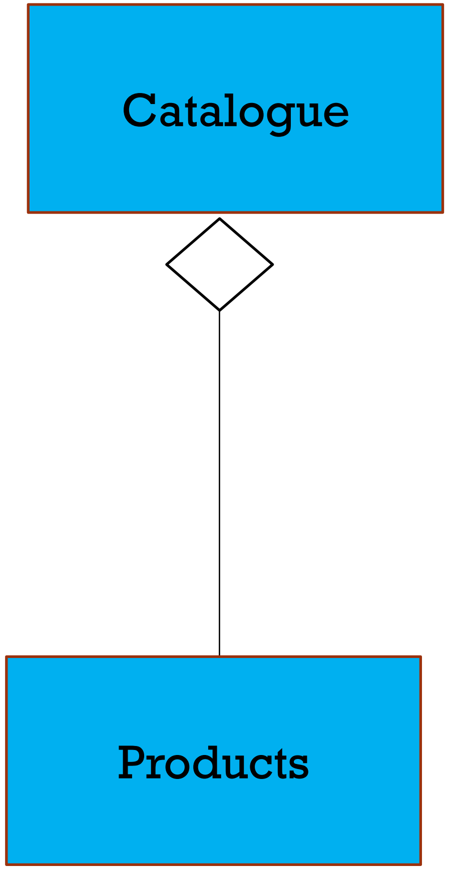


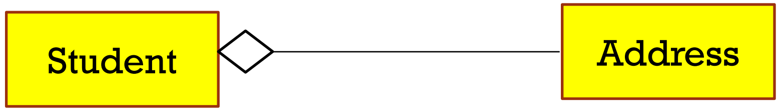
AGGREGATION RELATIONSHIP

- The association relationship **indicates a relatively loose relationship between two classes.**
- This means the objects belonging to these associated classes can exist independently of each other.
- However, if the relationship between two associated classes is tight, then it is represented by aggregation.
- Aggregation, is thus a special form of association. One class *contains* another class.
- Example: A Hospital has Departments.
- Aggregation also represents *composition* – that is, one class made up of another class or classes.
- The senior level aggregated class (Hospital) in the aggregation relationship has a diamond on its right-hand side.
- The "Whole" can exist without the "Part" and vice versa.
- The relationship cannot be reciprocal. Hospital has Department, Department does not have Hospital.
- Thus, it is a **unidirectional association**.



Students can exist without college.
Teachers can exist without college.

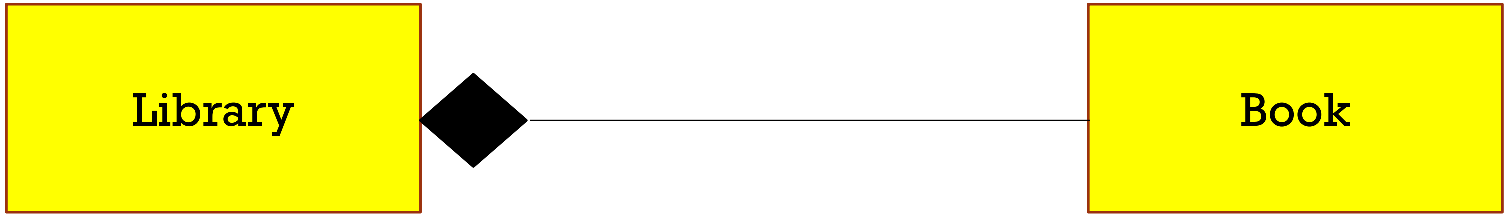




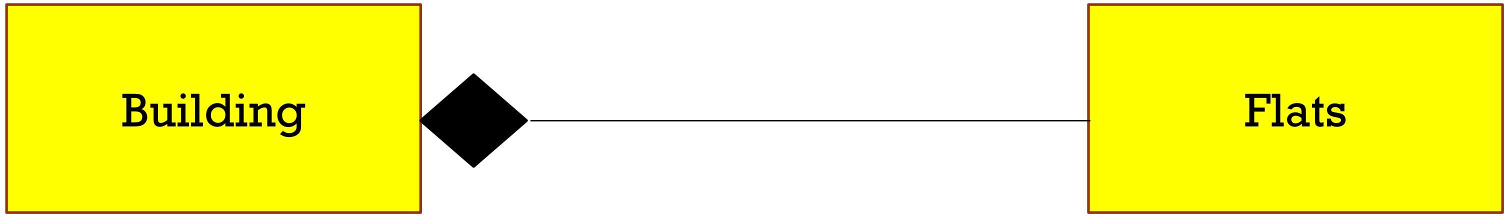
Student has address.

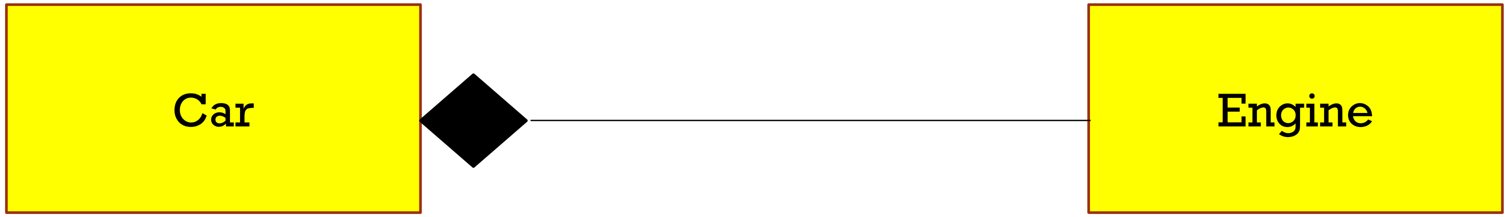
COMPOSITION RELATIONSHIP

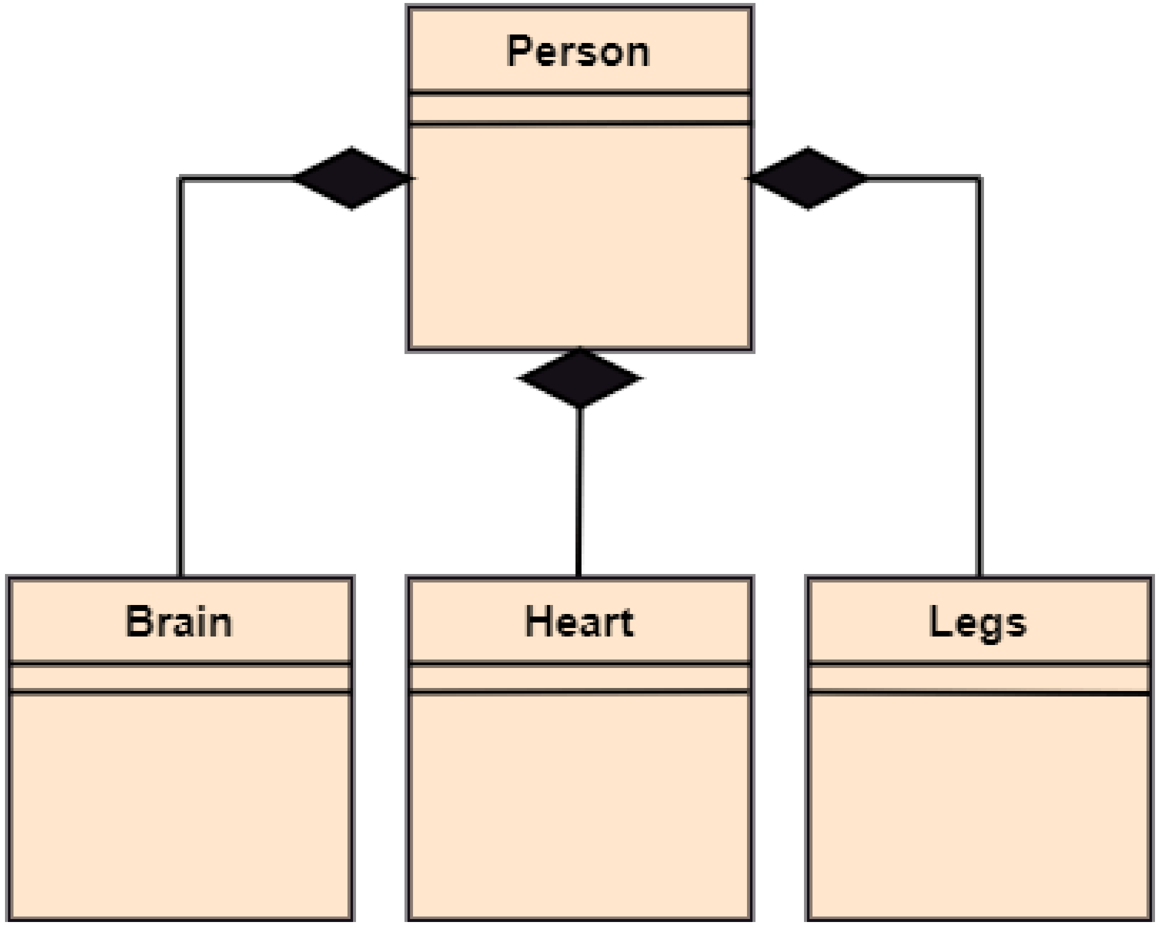
- Restricted form of aggregation, in which two entities are highly dependent on each other.
- It represents **part-of** relationship.
- Both entities are dependent on each other.
- In composition, if there are two classes, class A(considered as a whole) and class B(considered as part) , then the destruction of class object will mean that class B object also does not exist.
- This also means that class B object may be a part of class A object only. So, this is a tight association.
- Composition in UML is drawn as a "filled diamond" at the whole end.
- A composition relationship states that in this relationship the part can belong to only one whole and no sharing.
- If the parent object is destroyed, then the child objects also cease to exist. In composition the life cycle of the part or child is controlled by the whole or parent that owns it.
- In an aggregation relationship, the life cycles of parent objects and child objects are independent. In a composition relationship, the death of a parent object also means the death of its children.
- One class *made up of* another class or classes.



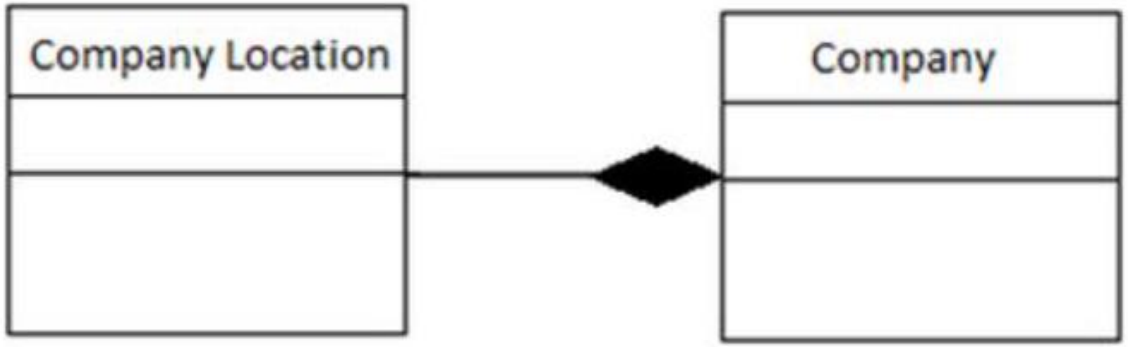
The lifetime of the part classifier is dependent on the lifetime of the whole classifier.
If the library is destroyed, books will be destroyed.

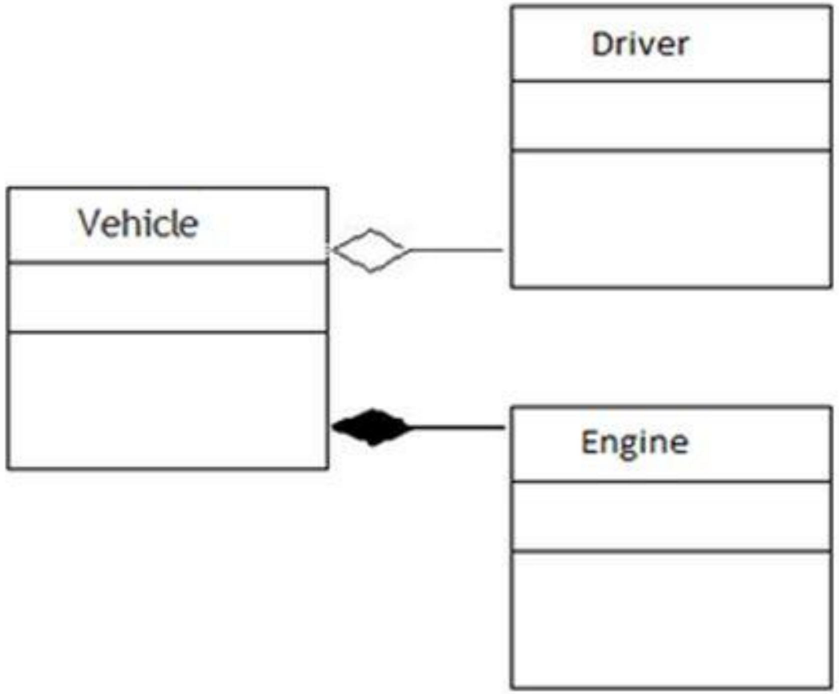


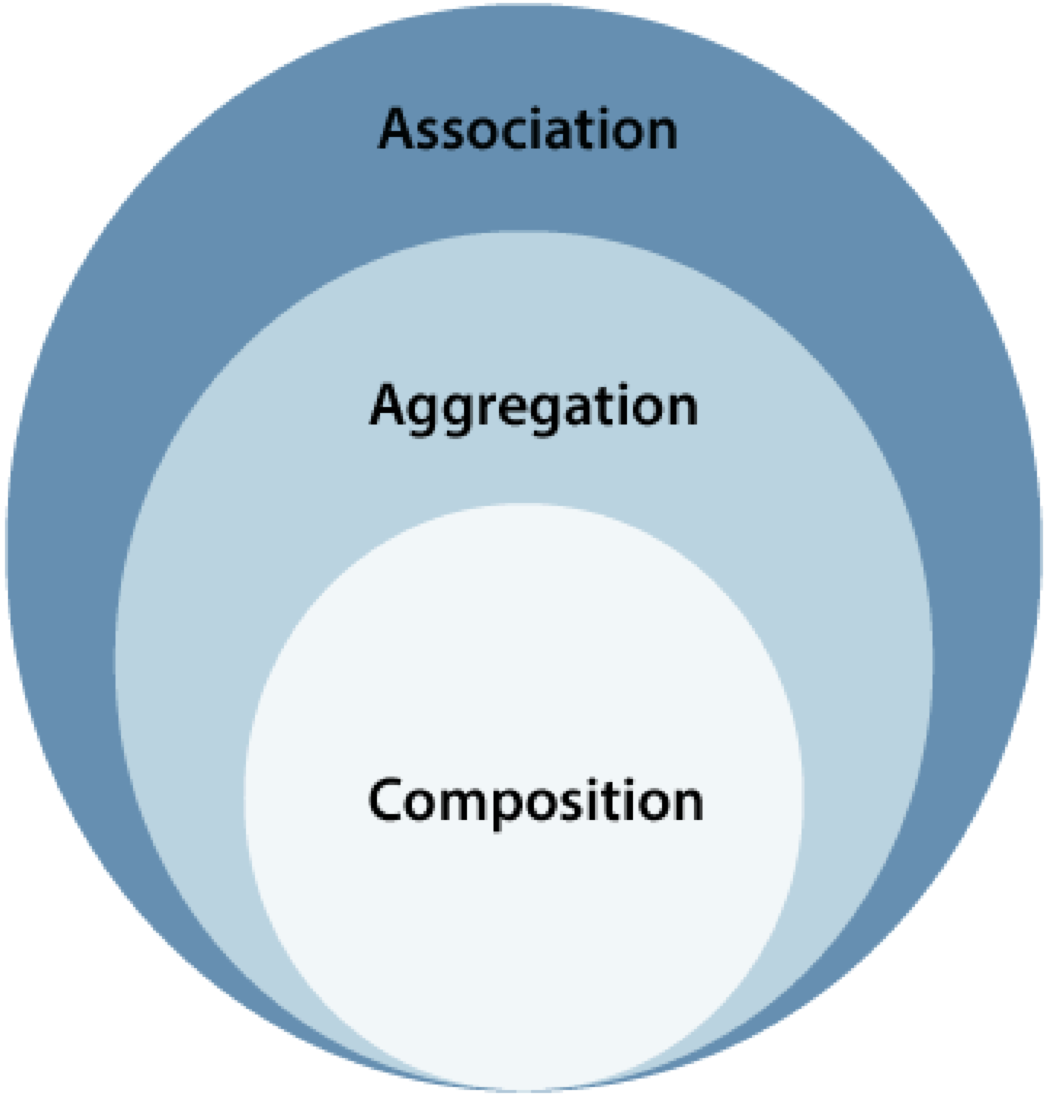




- The composition association relationship connects the Person class with Brain class, Heart class, and Legs class.
- If the person is destroyed, the brain, heart, and legs will also get discarded.
- The life cycle of child depends on the life cycle of Parent, they are not independent of each other.


Company Location
Company


Driver
Vehicle
Engine



- The composition and aggregation are two subsets of association.
- In both of the cases, the object of one class is owned by the object of another class; the only difference is that in composition, the child does not exist independently of its parent, whereas in aggregation, the child is not dependent on its parent i.e., standalone.
- An aggregation is a special form of association, and composition is the special form of aggregation.

Aggregation

- Is a special type of association.
- All objects have their own life cycle.
- Parent class not responsible for creating or destroying the child class.
- “Has –a” relationship.
- Weak association.

Composition

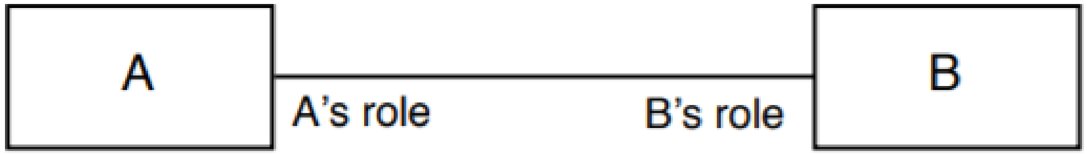
- Is a special type of aggregation.
- Child object does not have its own lifecycle, depends on the parent life cycle.
- The parent child responsible for creating or destroying the child class.
- “Has-a” as well as “Part-of” relationship.
- Strong Association.

HOMEWORK QUESTION

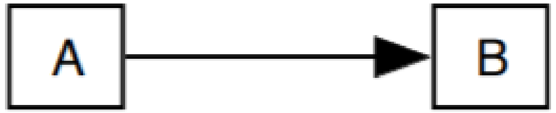
What is the difference between composition and inheritance? Which one is more favourable and why?

Read examples of aggregation and composition implementation in Java/C++.

Association (uses, interacts-with) relationship



Navigability - can reach
B starting from A

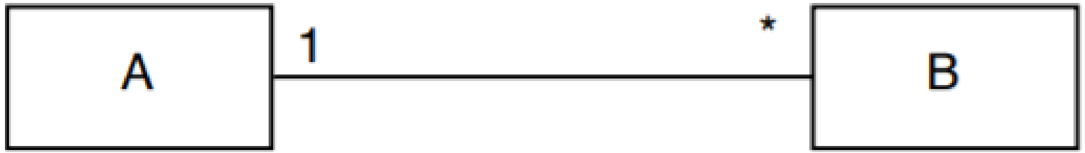


Multiplicity in Aggregation, Composition, or Association

* - any number
1 - exactly 1
n - exactly *n*

0..1 - zero or one
1..* - 1 or more
n .. *m* - *n* through *m*

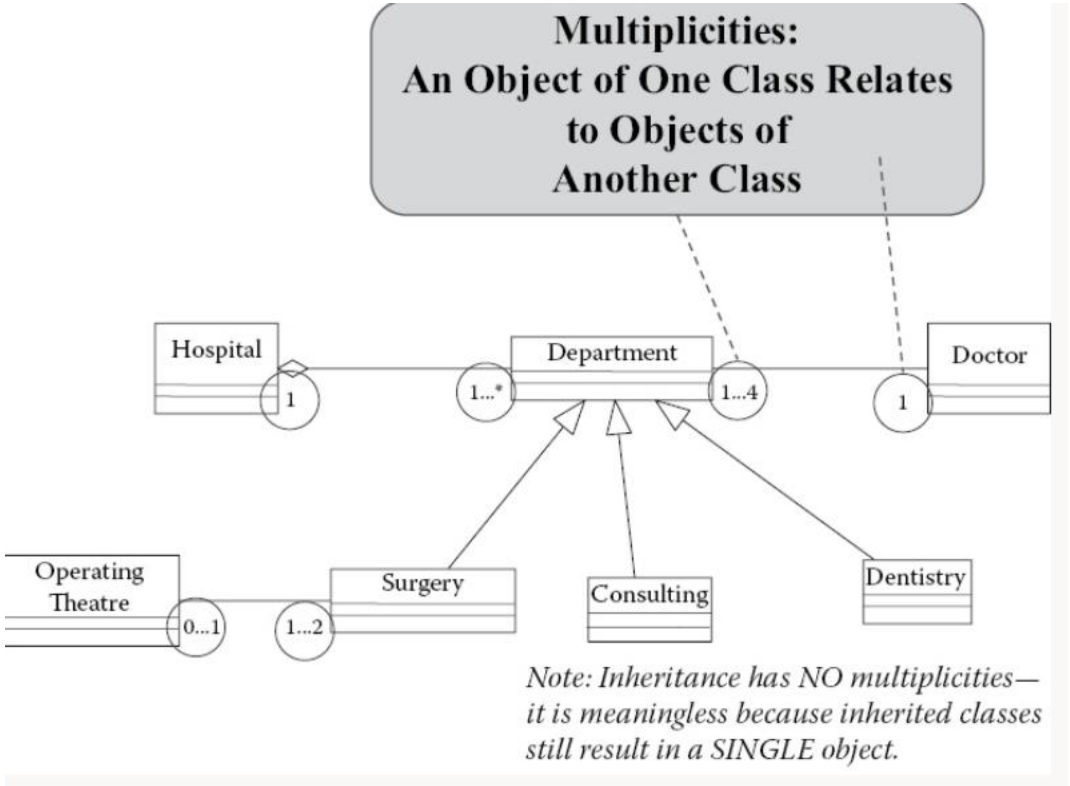
Follow line from start class to end class,
note the multiplicity at the end.
Say “Each <start> is associated with <multiplicity> <ends>”



Each A is associated with any number of B’s.
Each B is associated with exactly one A.

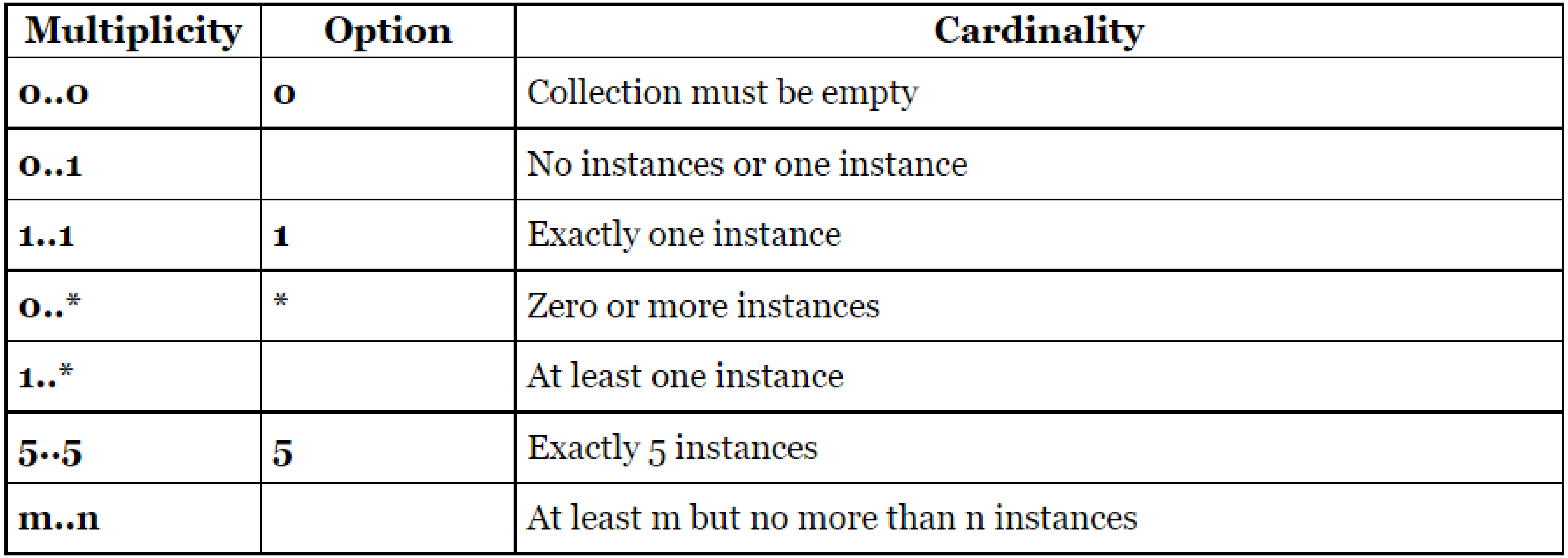
MULTIPLICITIES

- An association relationship between two classes can also carry information on the number of object (instance) counts at each end of the association.
- This object count is called multiplicity.
- Multiplicities indicate the number of objects of one class related to an object of another class.
- Thus, multiplicities make sense when two classes are in association or aggregation.

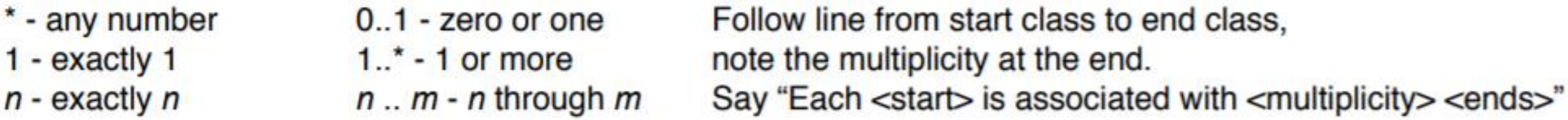


Multiplicities: An Object of One Class Relates to Objects of Another Class

Note: Inheritance has NO multiplicities—it is meaningless because inherited classes still result in a SINGLE object.



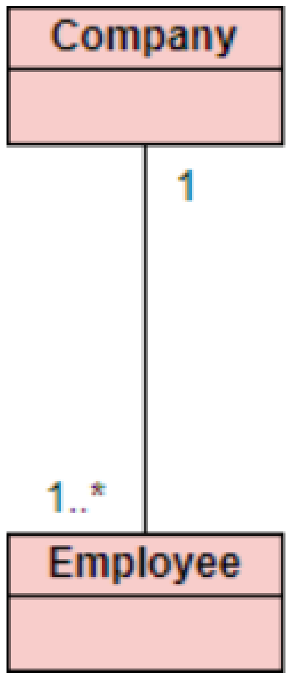
Multiplicity	Option	Cardinality
0..0	**0**	Collection must be empty
0..1		No instances or one instance
1..1	**1**	Exactly one instance
0..*	*	Zero or more instances
1..*		At least one instance
5..5	**5**	Exactly 5 instances
m..n		At least m but no more than n instances

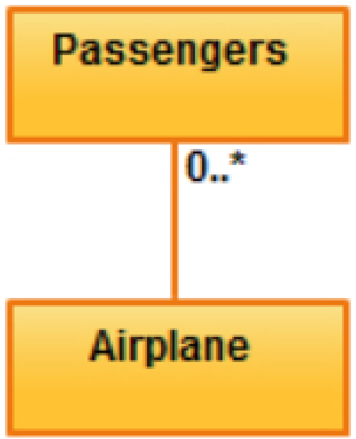


* - any number
1 - exactly 1
n - exactly *n*

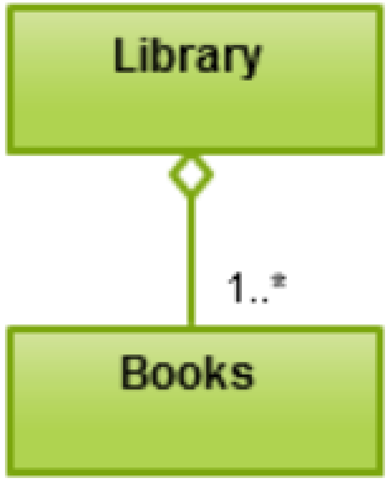
0..1 - zero or one
1..* - 1 or more
n .. *m* - *n* through *m*

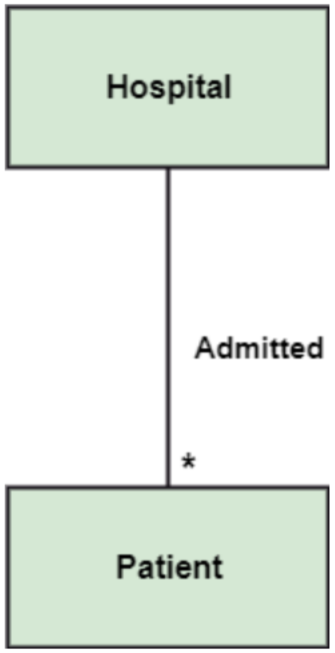
Follow line from start class to end class,
note the multiplicity at the end.
Say “Each <start> is associated with <multiplicity> <ends>”





Airplane is associated with 0 or many passengers.



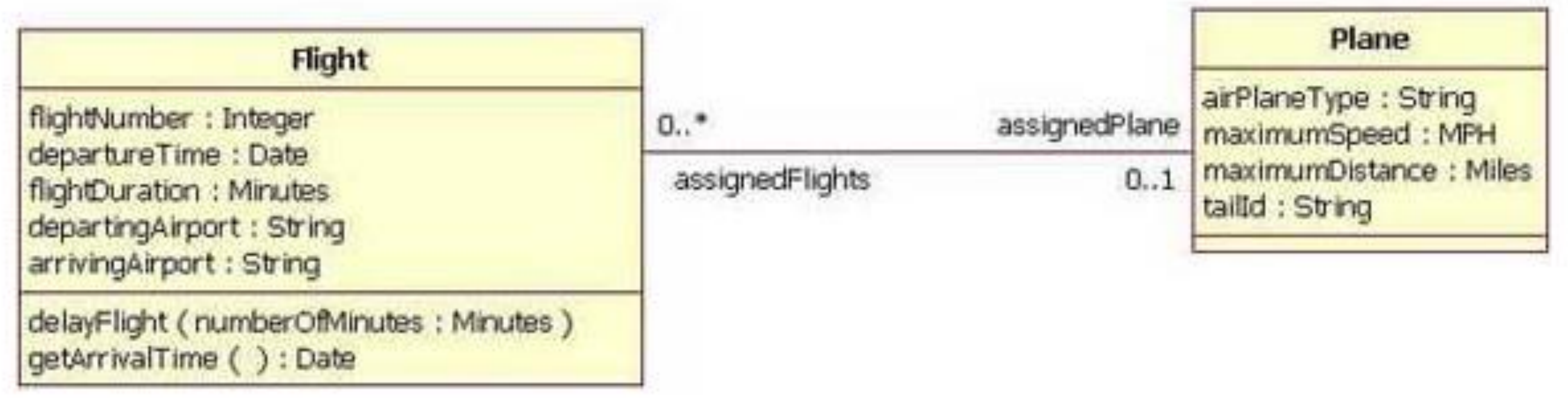


A company is associated with 1 or more employees. An employee is associated with only 1 company.

DIRECTIONAL ASSOCIATIONS

- We will discuss two types of directional associations:
  - Bi-directional
  - Uni-directional
- BI-DIRECTIONAL (STANDARD) ASSOCIATION
- Associations are always assumed to be bi-directional; this means that both classes are aware of each other and their relationship.
- UNI-DIRECTIONAL ASSOCIATION
- In a uni-directional association, two classes are related, but only one class knows that the relationship exists.

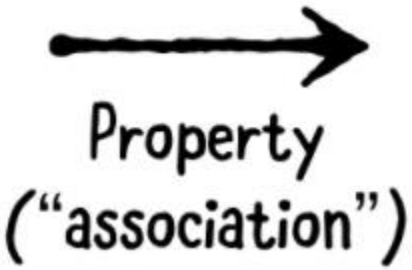
BI-DIRECTIONAL (STANDARD) ASSOCIATION

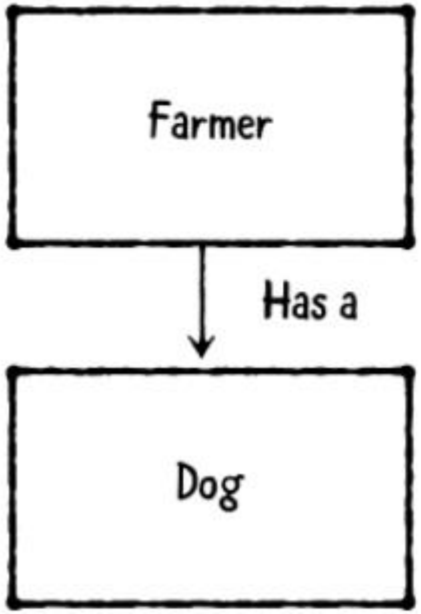


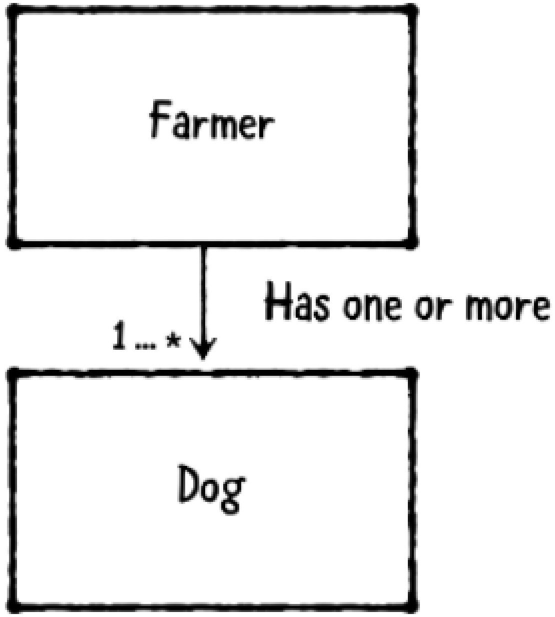
The multiplicity value next to the Plane class of 0..1 means that when an instance of a Flight exists, it can either have one instance of a Plane associated with it or no Planes associated with it (i.e., maybe a plane has not yet been assigned).

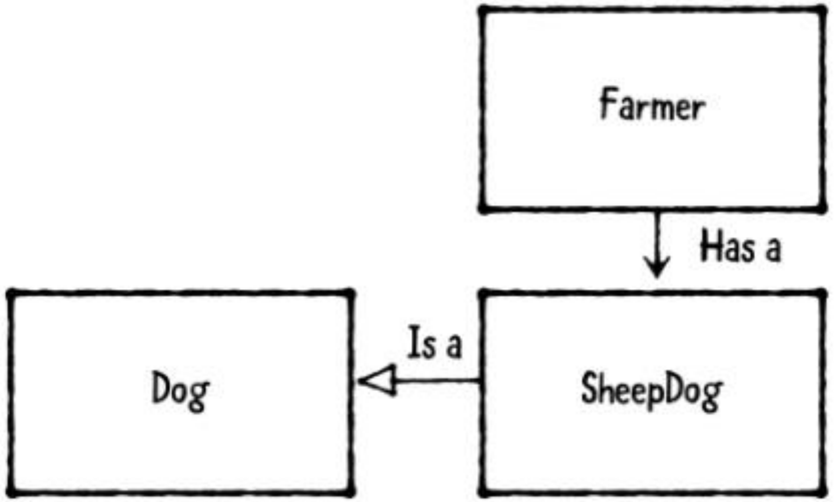
In this association, the Flight takes on the role of “assignedFlights”; the diagram tells us that the Plane instance can be associated either with no flights (e.g., it’s a brand new plane) or with up to an infinite number of flights (e.g., the plane has been in commission for the last five years).

DIRECTION OF ASSOCIATION

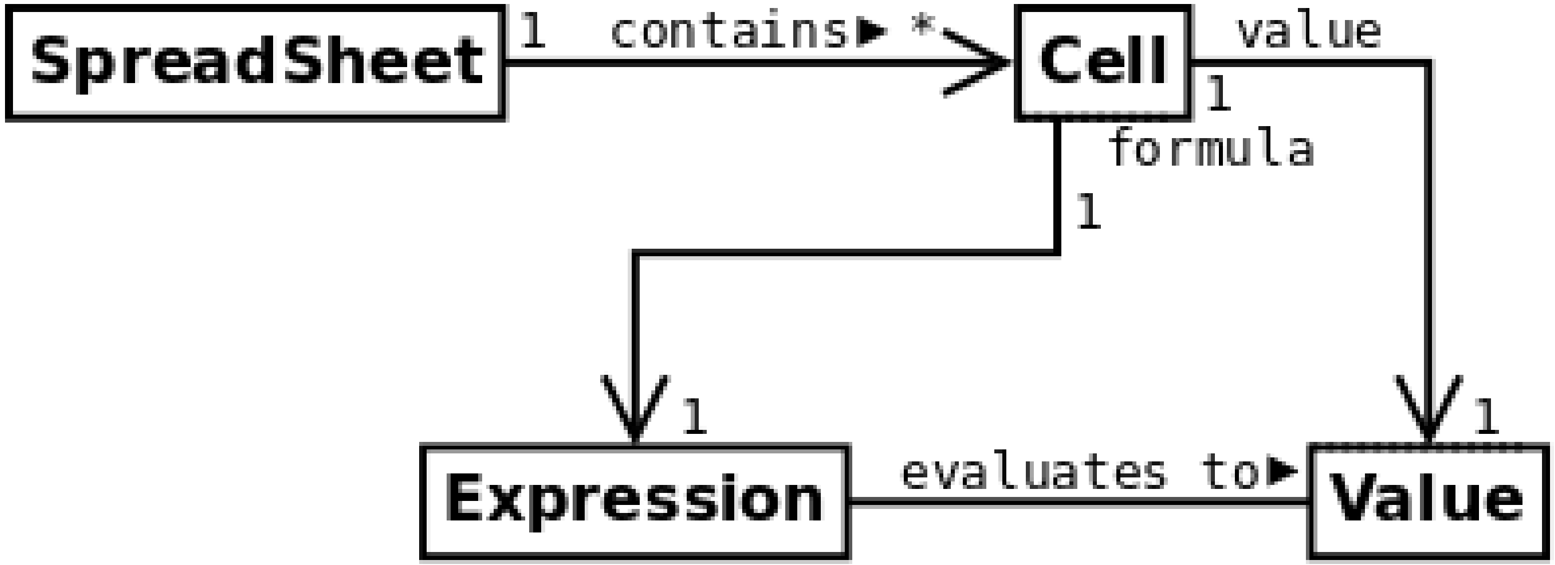




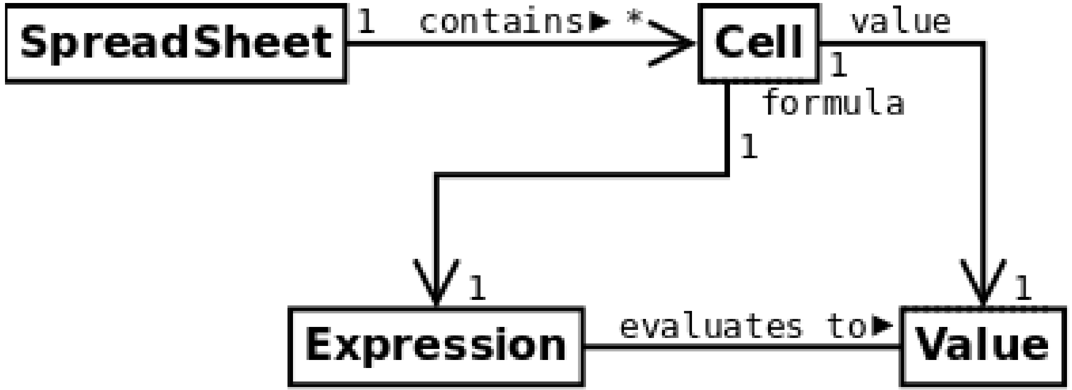




UNI-DIRECTIONAL ASSOCIATION



NAVIGABILITY



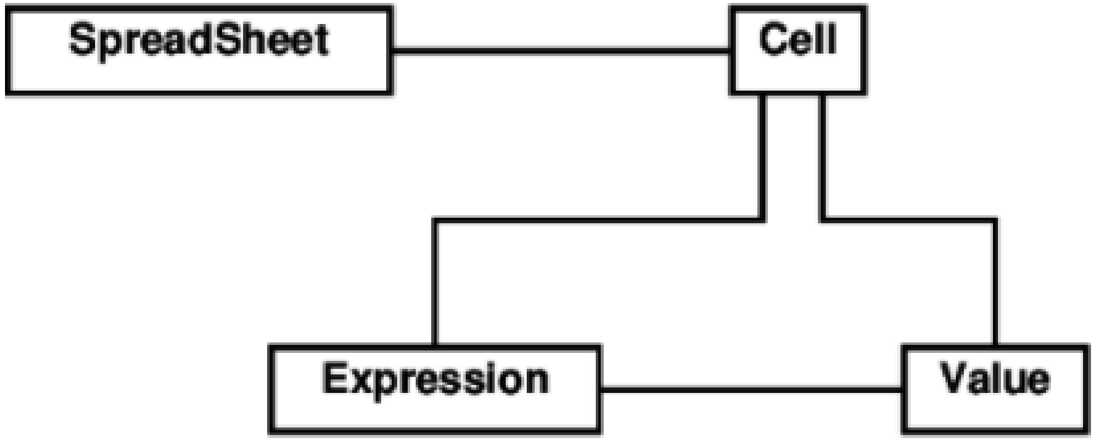
The diagram above suggests that,

- Given a spreadsheet, we can locate all of the cells that it contains, but that
  - we cannot determine from a cell in what spreadsheet it is contained.
- Given a cell, we can obtain the related expression and value, but
  - given a value (or expression) we cannot find the cell of which those are attributes.

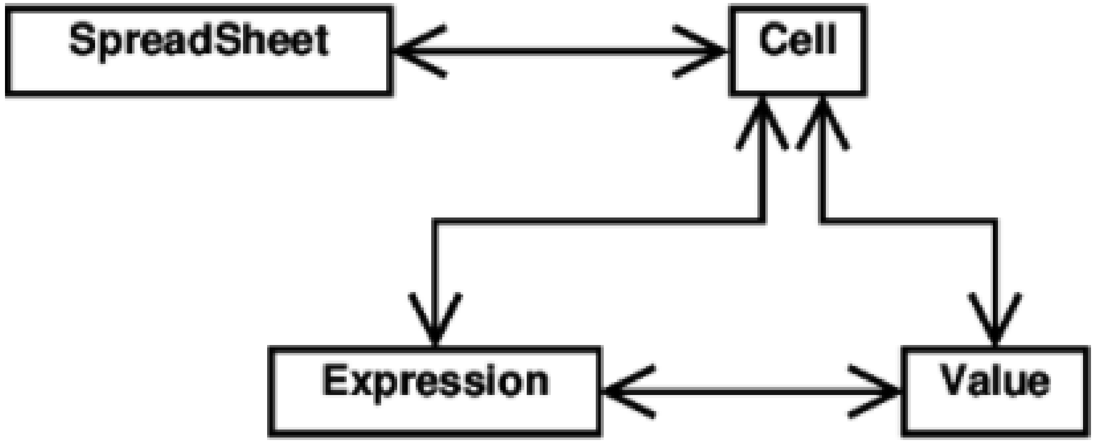
- We would not have a role name for an un-navigable direction of an association.

NAVIGABILITY AND PERSPECTIVES

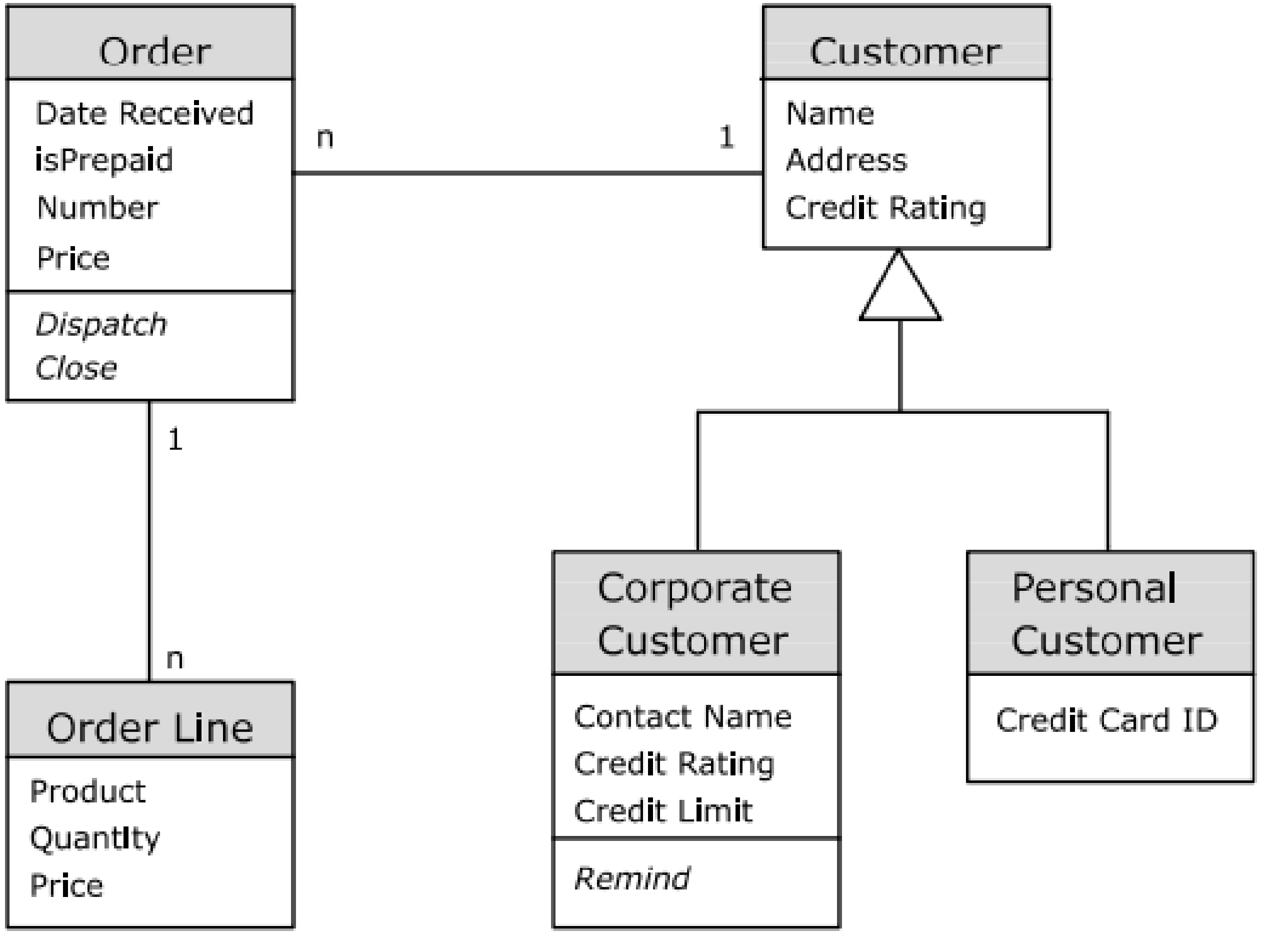
- Navigability is probably not useful in most conceptual diagrams.
- In a specification/implementation diagram, if no arrows are shown on an association, navigability defaults to two-way.



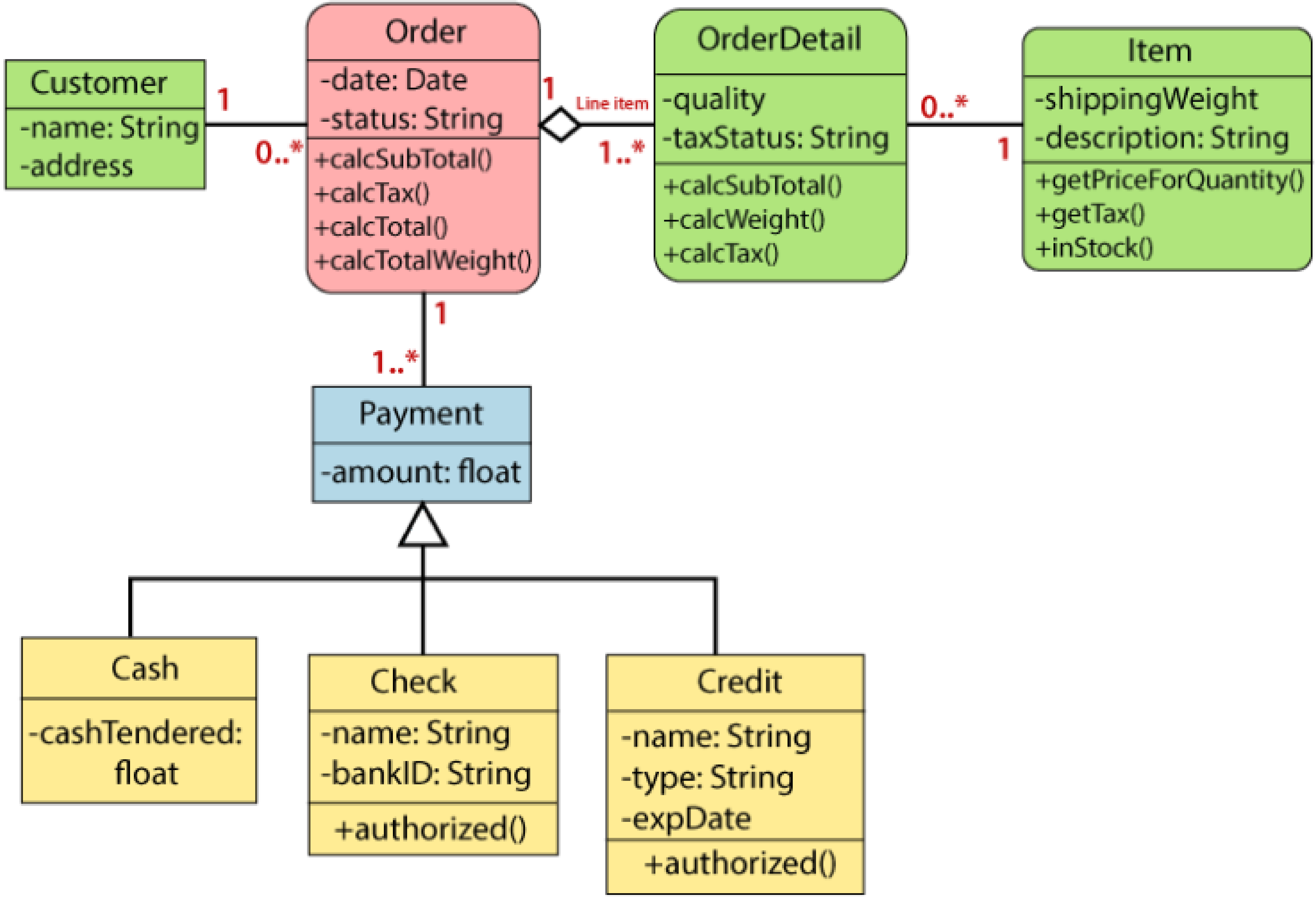
defaults to



CLASS DIAGRAM EXAMPLES



A simple class diagram for a commercial Software application.


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

STEPS TO CREATE CLASS DIAGRAM

- **Identify classes**
  - These are the abstract or physical "things" in our system which we wish to describe. Find all the nouns and noun phrases in the domain descriptions you have obtained through your analysis. Consider these class candidates.
- **Find associations**
  - Now find the verbs that join the nouns. e.g., The professor (noun) teaches (verb) students (noun). The verb in this case, defines an association between the two nouns. Identify the type of association. Use a matrix to define the associations between classes.
- **Draw rough class diagram**
  - Put classes in rectangles and draw the associations connecting the classes.
- **Fill in multiplicity**
  - Determine the number of occurrences of one class for a single occurrence of the associated class.
- **Identify attributes**
  - Name the information details (fields) which are relevant to the application domain for each class.
- **Identify Behaviours**
  - Specify the operations that are required for each class. (assume getter and setter methods for each attribute.)
- **Review your diagram and fine tune it.**
  - Look for inconsistencies and errors. Fix them. Make sure you have captured everything required from the domain you are studying - that your diagram is complete.

EXAMPLE

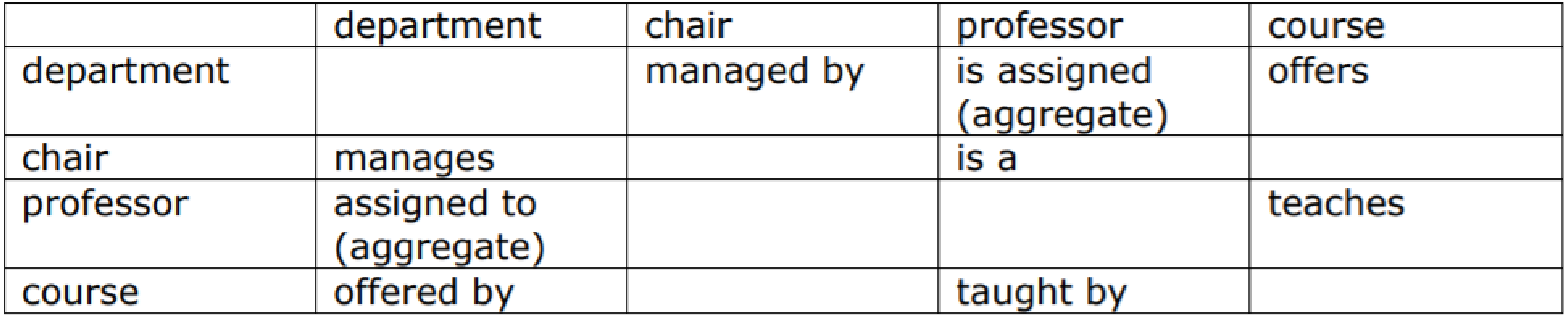
- IIIT Pune has several departments.
- Each **department** is managed by a **chair**, and at least one **professor**.
- Professors must be assigned to one, but possibly more departments.
- At least one professor teaches each **course**, but a professor may be on sabbatical and not teach any course.
- Each course may be taught more than once by different professors.
- We know of the department name, the professor name, the professor employee id, the course names, the course schedule, the term/year that the course is taught, the departments the professor is assigned to, the department that offers the course

IDENTIFY CLASSES

- These are the abstract or physical "things" in our system which we wish to describe. Find all the nouns and noun phrases in the domain descriptions you have obtained through your analysis.
- Consider these class candidates. The class candidates are
- Departments
- Chair
- Professor
- Course

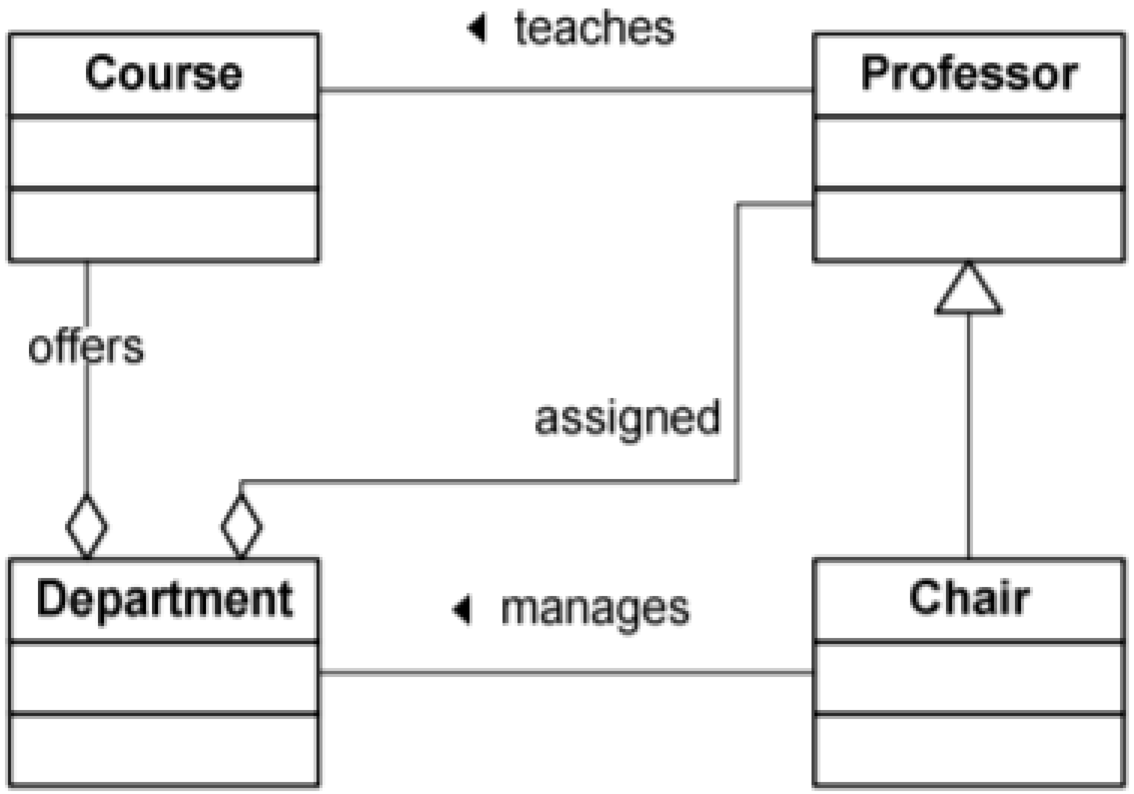
FIND ASSOCIATIONS

- Now find the verbs that join the nouns.
- e.g., The professor (noun) teaches (verb) students (noun).
- The verb in this case, defines an association between the two nouns.
- Identify the type of association. Use a matrix to define the associations between classes.

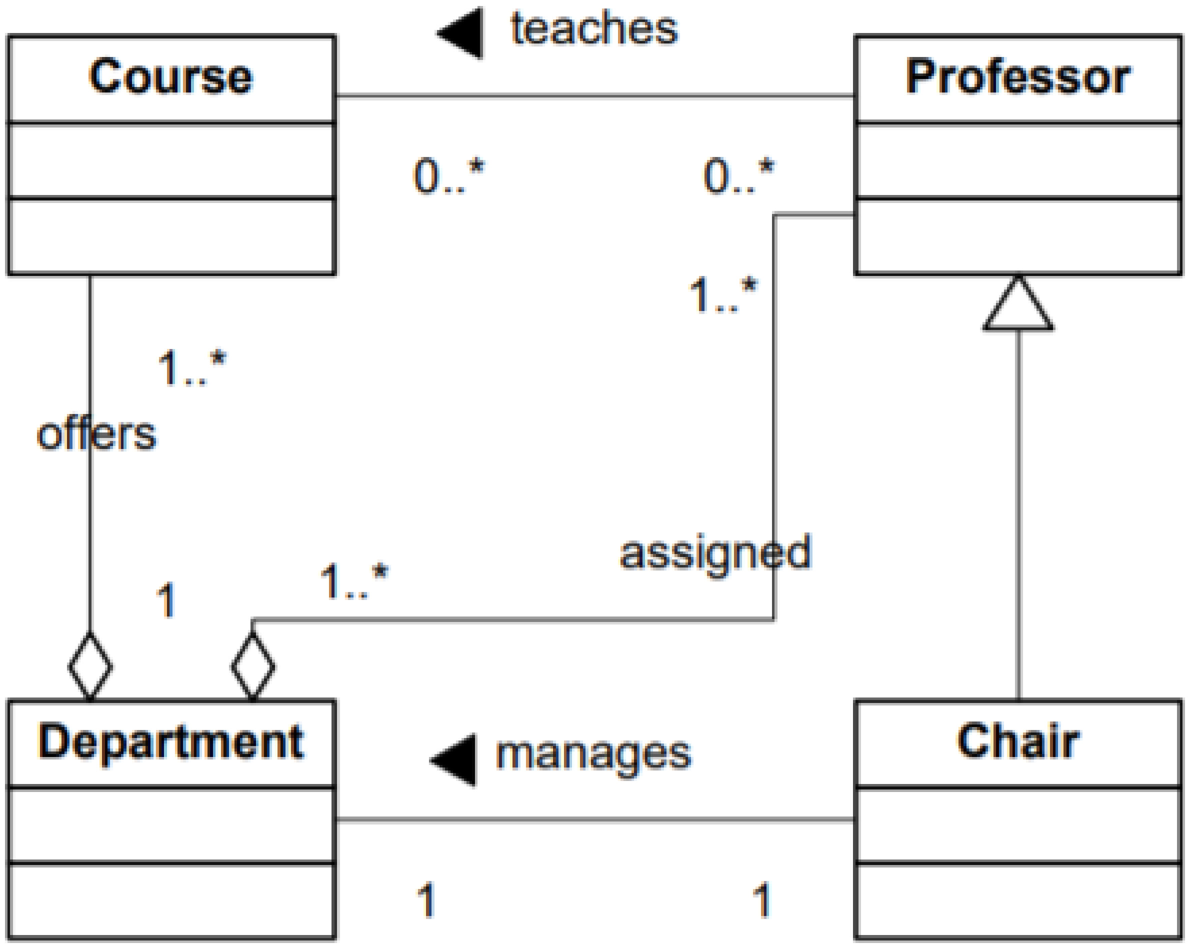


	department	chair	professor	course
department		managed by	is assigned (aggregate)	offers
chair	manages		is a	
professor	assigned to (aggregate)			teaches
course	offered by		taught by	

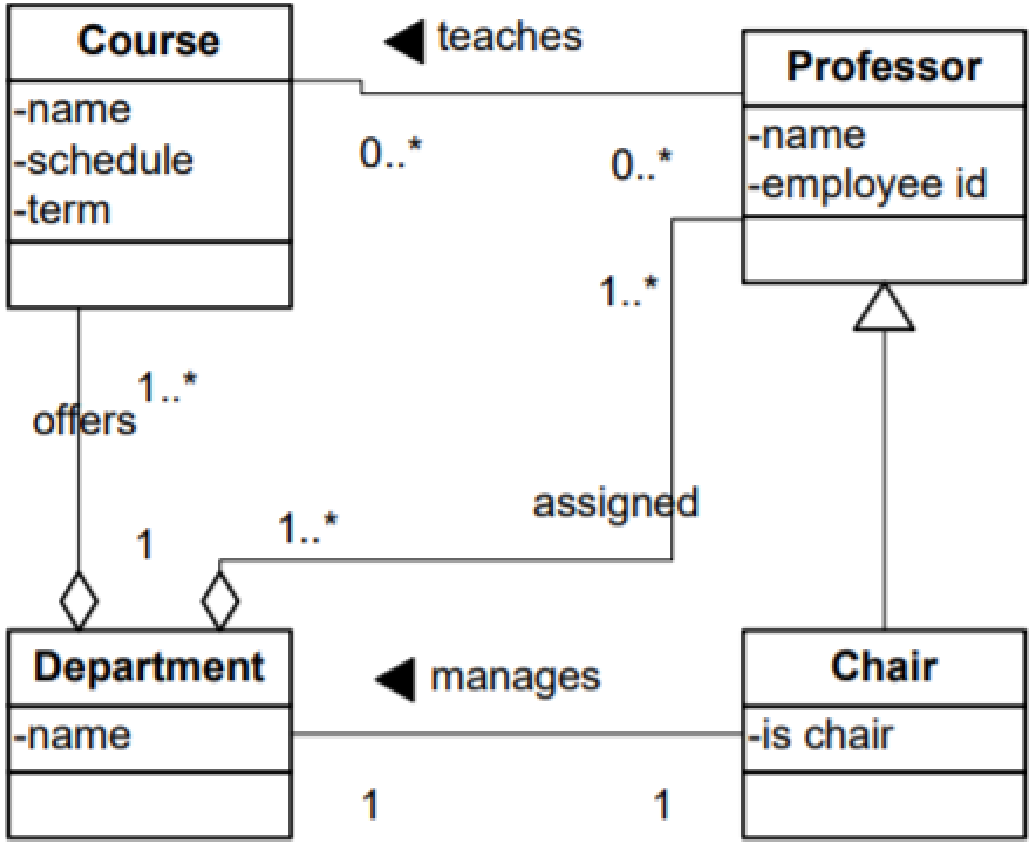
DRAW A ROUGH CLASS DIAGRAM



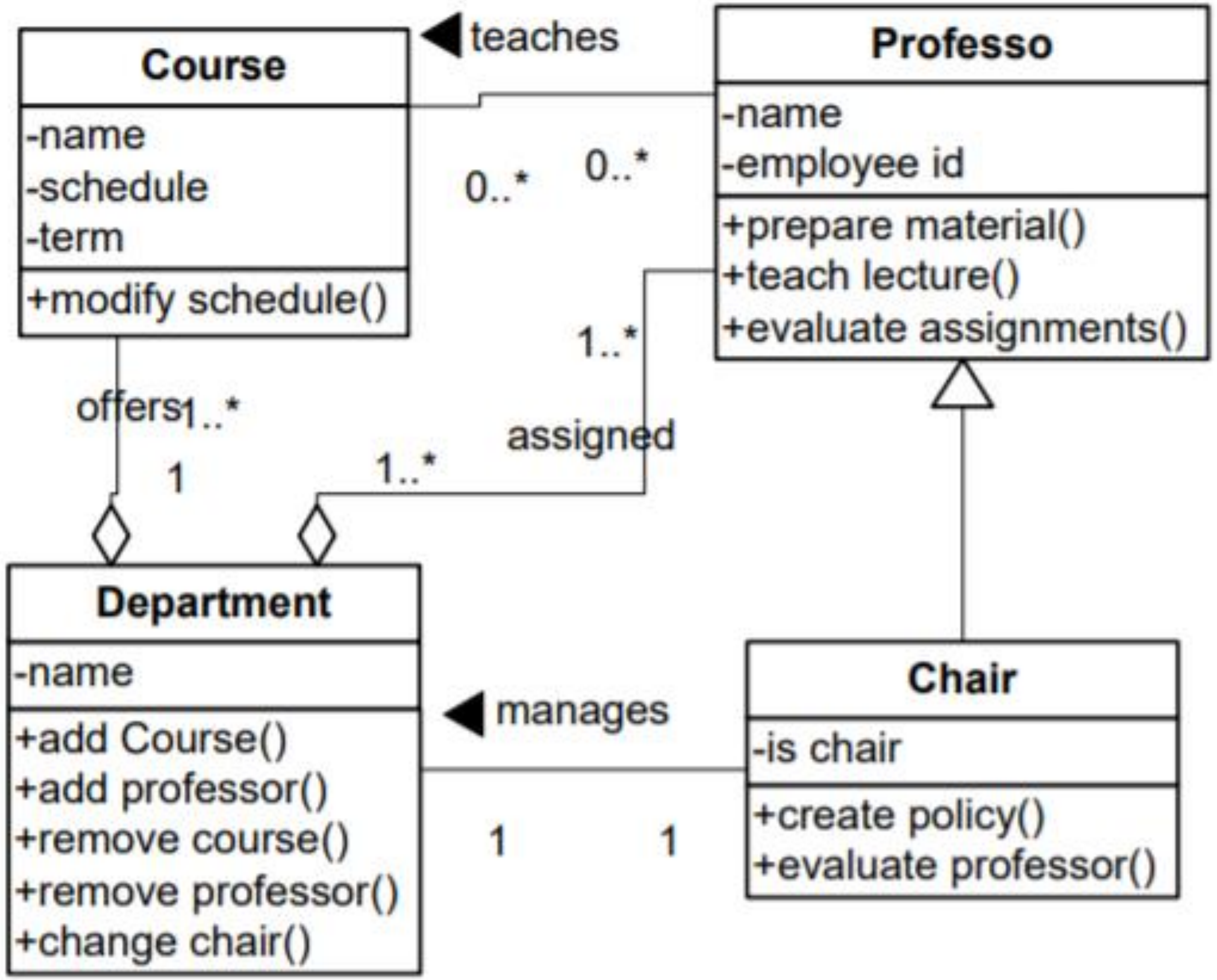
FILL IN MULTIPLICITY



IDENTIFY ATTRIBUTES

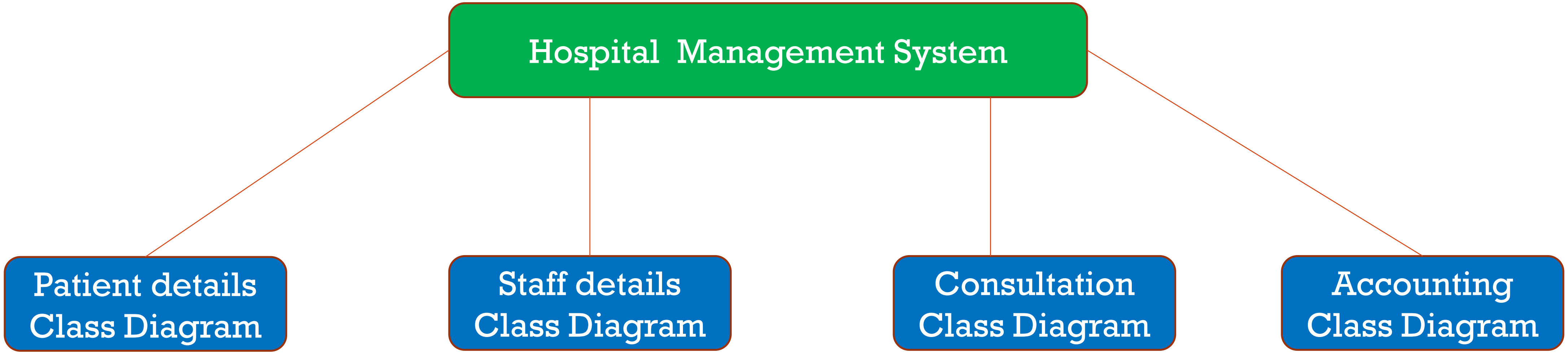


IDENTIFY BEHAVIOURS

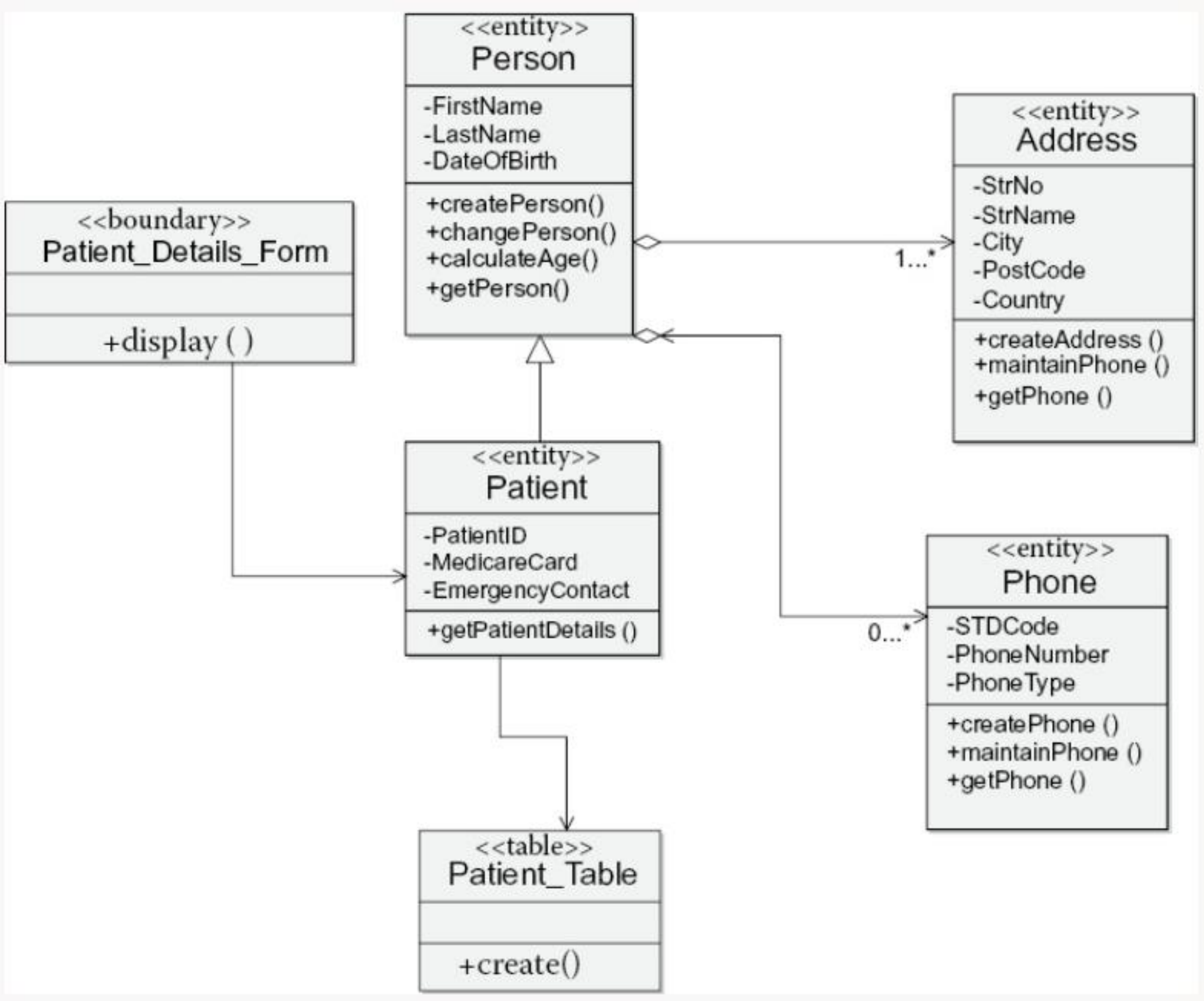


DISAGGREGATING CLASS DIAGRAMS

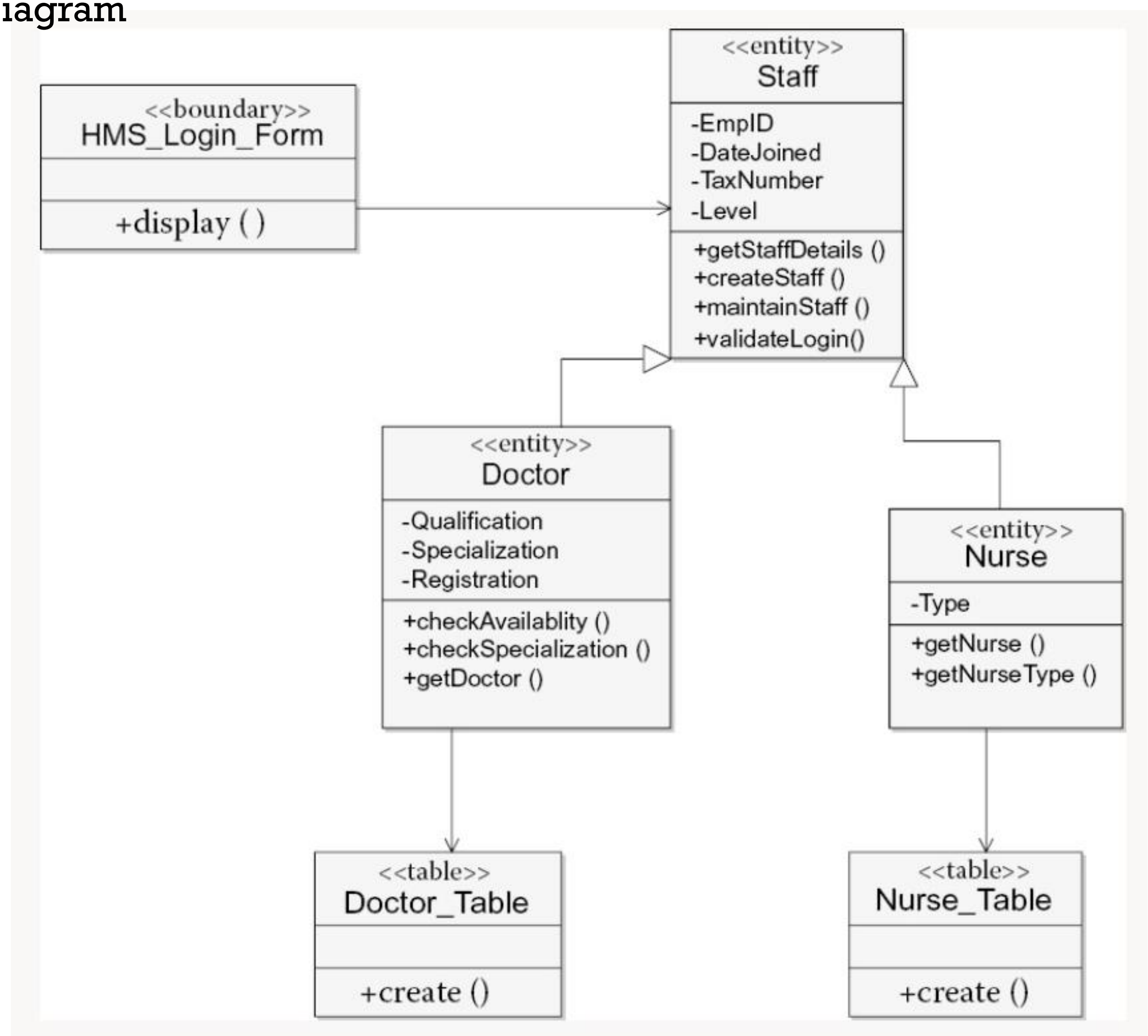
- Sometimes a system may be too complex, so you may need to break your class diagram into sections.


Hospital Management System
Patient details Class Diagram
Staff details Class Diagram
Consultation Class Diagram
Accounting Class Diagram

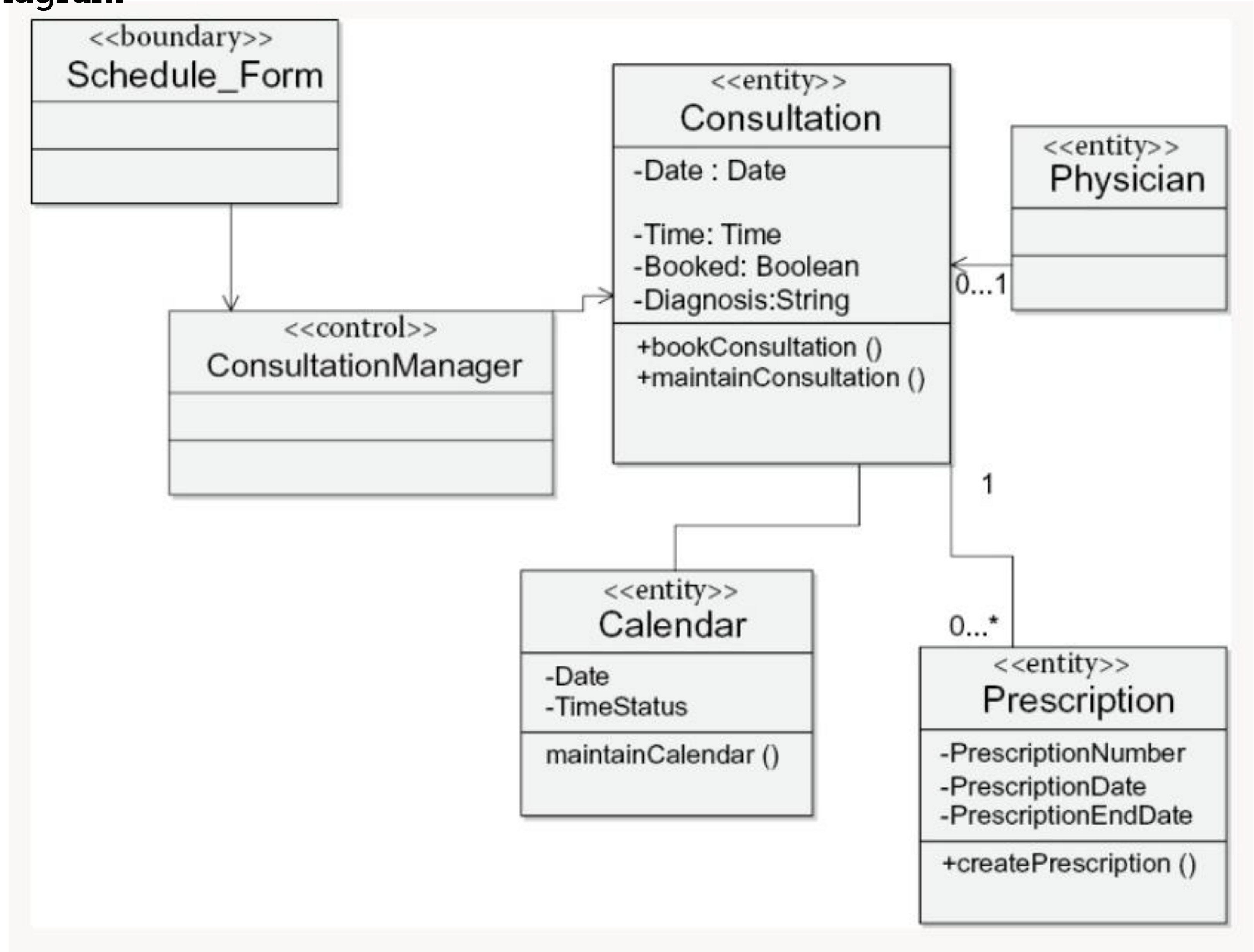
Patient Details Class Diagram



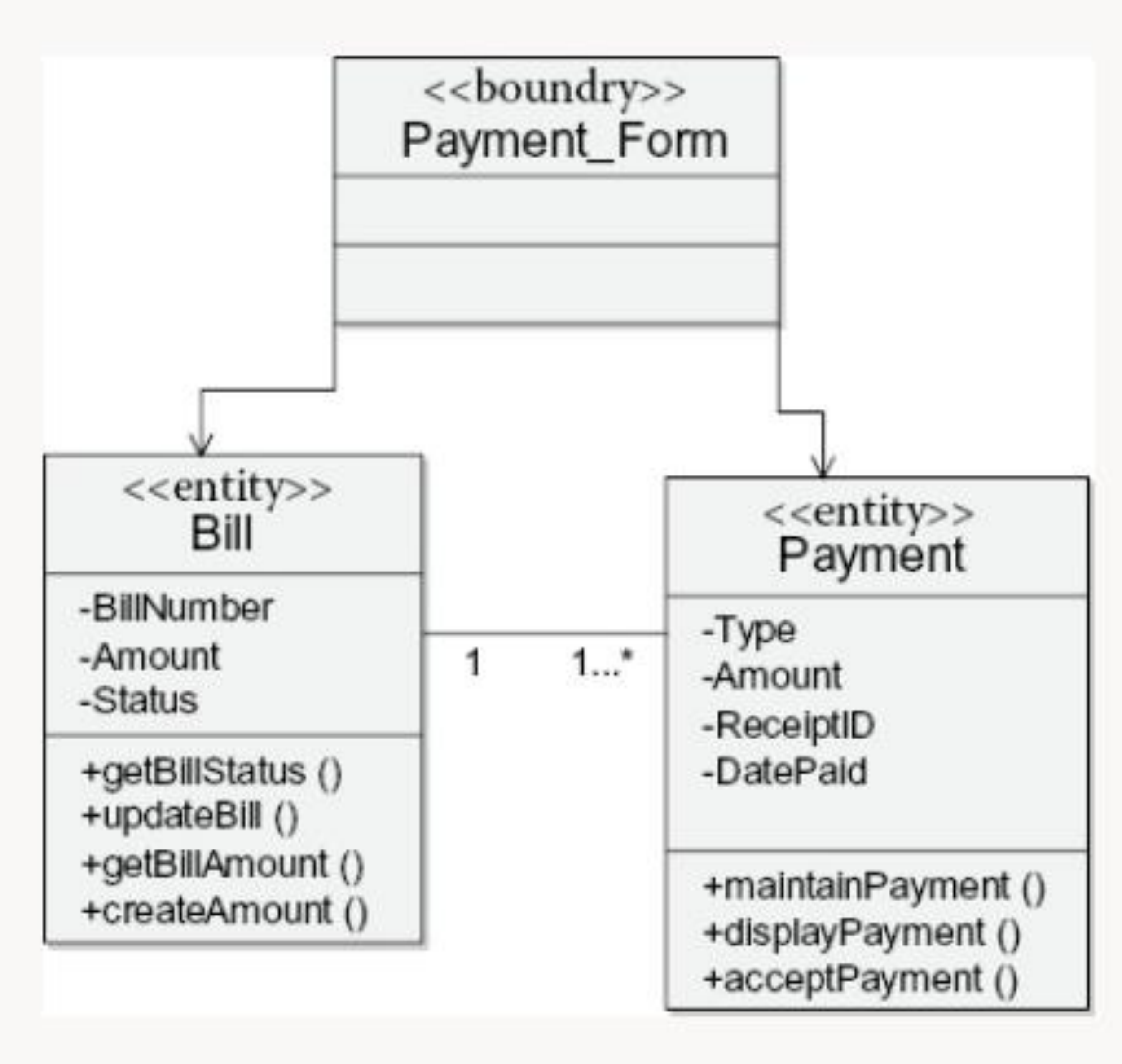
Staff Details Class Diagram



Consulting Class Diagram



Payment Class Diagram



STRENGTHS OF CLASS DIAGRAM

- Diagrams are an excellent structural representation of the system, allowing the modeler to **capture all of the important entities in the problem space**.
- Class relationships demonstrate dependencies between classes that **enable understanding of the sequence in which messages are sent between their corresponding objects.**
- Multiplicities provide valuable information on database modeling.
- Class diagrams in design **provide a modeling construct that is closest to coding.**
- Multiplicities in a class diagram aid in the creation of a relational database schema by showing which tables require foreign keys to create the relation (i.e., classes with many [*] relationships will require a foreign key when mapped to a table).

WEAKNESS OF CLASS DIAGRAMS

- Class diagrams do not show any dynamic/behavioral information and thus do not display any concept of time. **They cannot show an “if-then-else” scenario, which makes them extremely weak in representing the dynamics of a system.**
- The aggregation relationship, and variations of it (such as composition, not discussed in this text), is unclear and has led to numerous debates and interpretations.
- Attempting code generation from a class diagram without adequate preparation can lead to confusion and errors.
- Inappropriate level of usage. A developer should use the advanced class diagram, with extra implementation classes and full signatures, at the solution level, whereas a business analyst should remain at the business entity level and only show <> classes in these diagrams

COMMON ERRORS IN BASIC CLASS DIAGRAM AND HOW TO RECTIFY THEM

Common Errors	Rectifying the errors	Examples
Using inheritance instead of association.	Ensure a semantic relationship with commonalities for inheritance; otherwise relate the two classes through association.	"Car is a vehicle" has meaningful commonalities—an inheritance relationship; but a car and a driver will be an association relationship
Paying too much attention to association vs. aggregation.	Start with an association relationship by default. Move it to aggregation only if the relationship is so close that objects from one class are tightly integrated with objects from another class	Room and walls are more tightly associated, hence aggregation.
Not adding multiplicities in the association	As much as possible, add multiplicities to association relationships.	When unsure, add "*" or "N" to indicate unknown multiplicities

Common Errors	Rectifying the errors	Examples
Adding multiplicities in inheritance relationship.	Inheritance implies one class is a type of another class. The object instantiated from this inheritance relationship is a single object. Therefore, multiplicities make no sense here.	Consider “car is a vehicle.” When this class design is instantiated, the single object that is created has the definition of both a car and a vehicle. Therefore, there is no multiplicity (number of objects of one class in relationship with number of objects of another class) in a car—vehicle inheritance relationship.

END