CMPS310

Lecture 5

Object Oriented Design Principles
Domain Model
and
Design Class Diagram

Topics:

- ***Developing Domain model using O-O design principles***
- ***Constructing Design Class Diagram***
- ***Class relationships***

Object Thinking

- Objects are instances of classes
- At the *conceptual level*, an object is a set of responsibilities.
- At the *specification level*, an object is a set of methods (behaviors)
- At the *implementation level*, an object is code and data and computational interactions between them.
- Identifying '*things*', 'types of things', their '*properties*', '*behaviour*' and '*relationships*' with other 'things' is critical and requires an approach called **Object Thinking.**
- A 'thing'/ 'object' may have physical existence or not
- An 'object' may only exist conceptually
- An object is an independent, entity, which
  - 'knows things' or 'stores things' (**properties** of objects)
  - 'does things' or encapsulates services (**behaviours** of objects)
  - 'collaborates with other objects' by exchanging messages (**relationships** among objects)

Tasks of Object Oriented Design

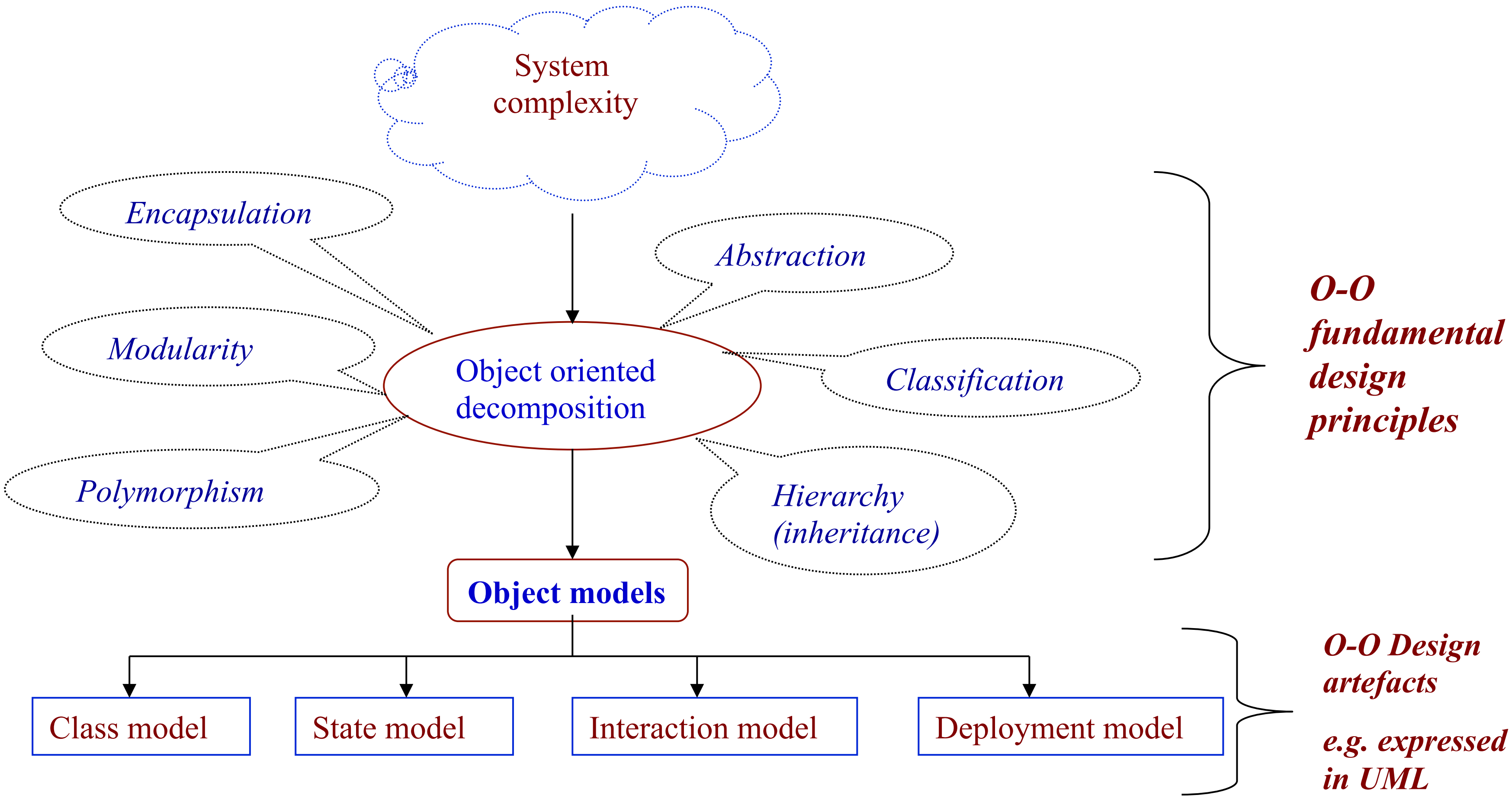
- To **identify objects** and **classes** from the problem domain –the core of the system
- To **recognise properties and behaviour** of the objects which are essential for the system by ignoring the inessential properties
- To **establish relationships** among the identified objects
- To **express all the above** design decision in a cohesive way using the appropriate notation such as UML
- To **transform** the design decisions into executable codes – developing running program

Applying Object Oriented Fundamental Design Principles

- To **identify classes** from the problem domain
  - Use the following object-oriented design principles*:*
    - Object thinking: Representing data in real world objects
    - Abstraction: Identifying objects, their properties and behaviour
    - Classification: Recognizing sameness of objects, their properties, behaviour
    - Modularity: Break into smaller pieces
- To **recognise properties and behaviour** of the objects by ignoring the inessential properties.
  - Use the following object-oriented design principles*:*
    - Encapsulation: Hiding details of properties and behaviour of objects
    - Abstraction: Identifying objects, their properties and behaviour
- To **establish relationships** among the identified objects
  - we use the following object-oriented design principles*:*
    - Classification: Recognising sameness of objects, their properties, behaviour
    - Hierarchy: Establishing specialisation relationships of objects
    - Modularity: Break into smaller pieces

O-O Decomposition and Artifact

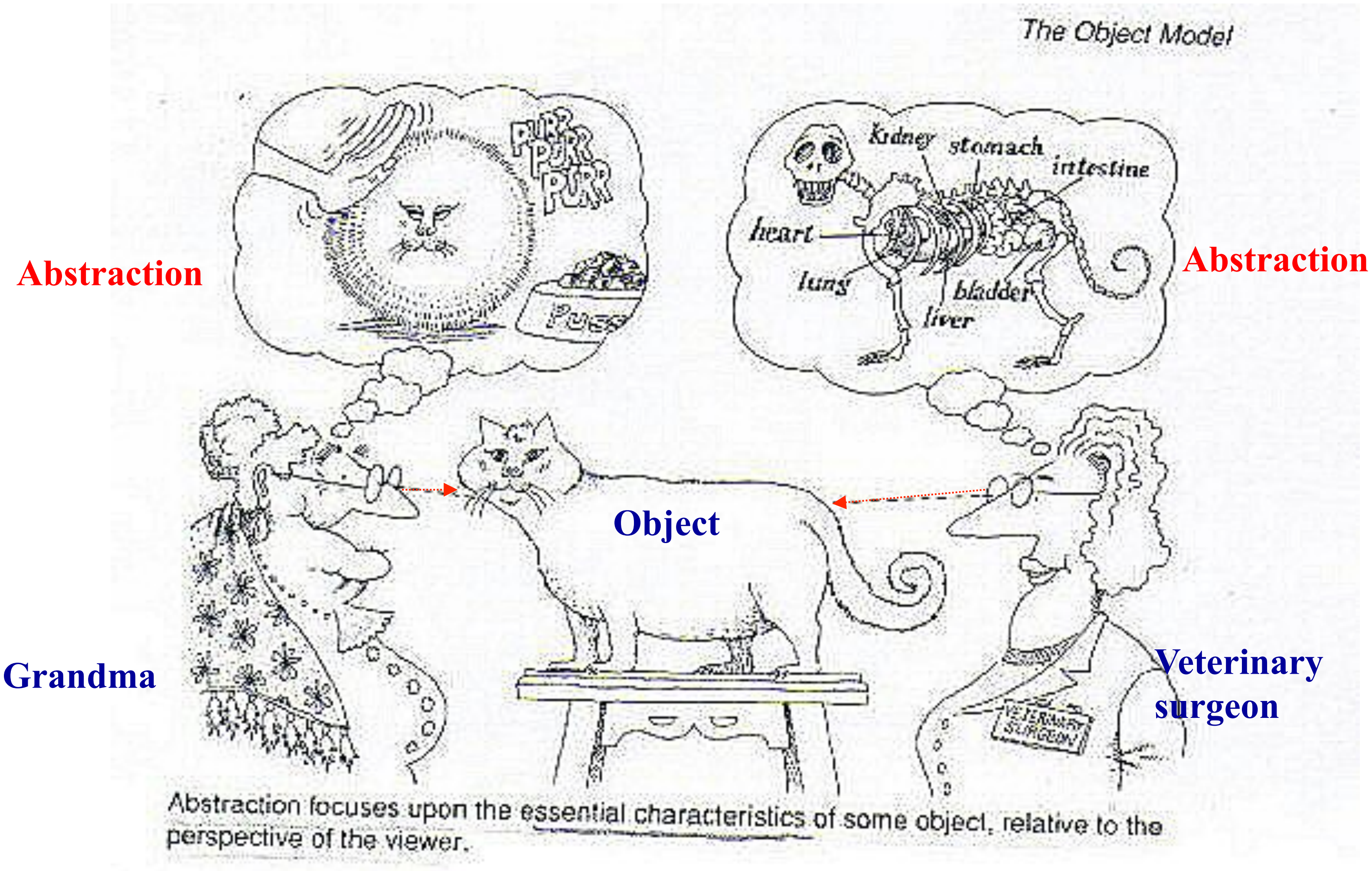
- Identification of classes and their associations require *object thinking* (O-O fundamental design principles)
- O-O Modeling is built upon well defined elements we collectively call the object model



Abstraction

- A model is an abstraction of something for the purpose of understanding it before building it
- Abstraction is a fundamental human capability that permits us to deal with complexity
- This is a design principle that is used to identify, recognize objects that are well suited to an application.
  - OO principles allows us to model a system using abstractions from the problem domain
- Abstraction allows us to manage complexity by creating a simplified representation of something
  - Concentrating on the essential characteristics

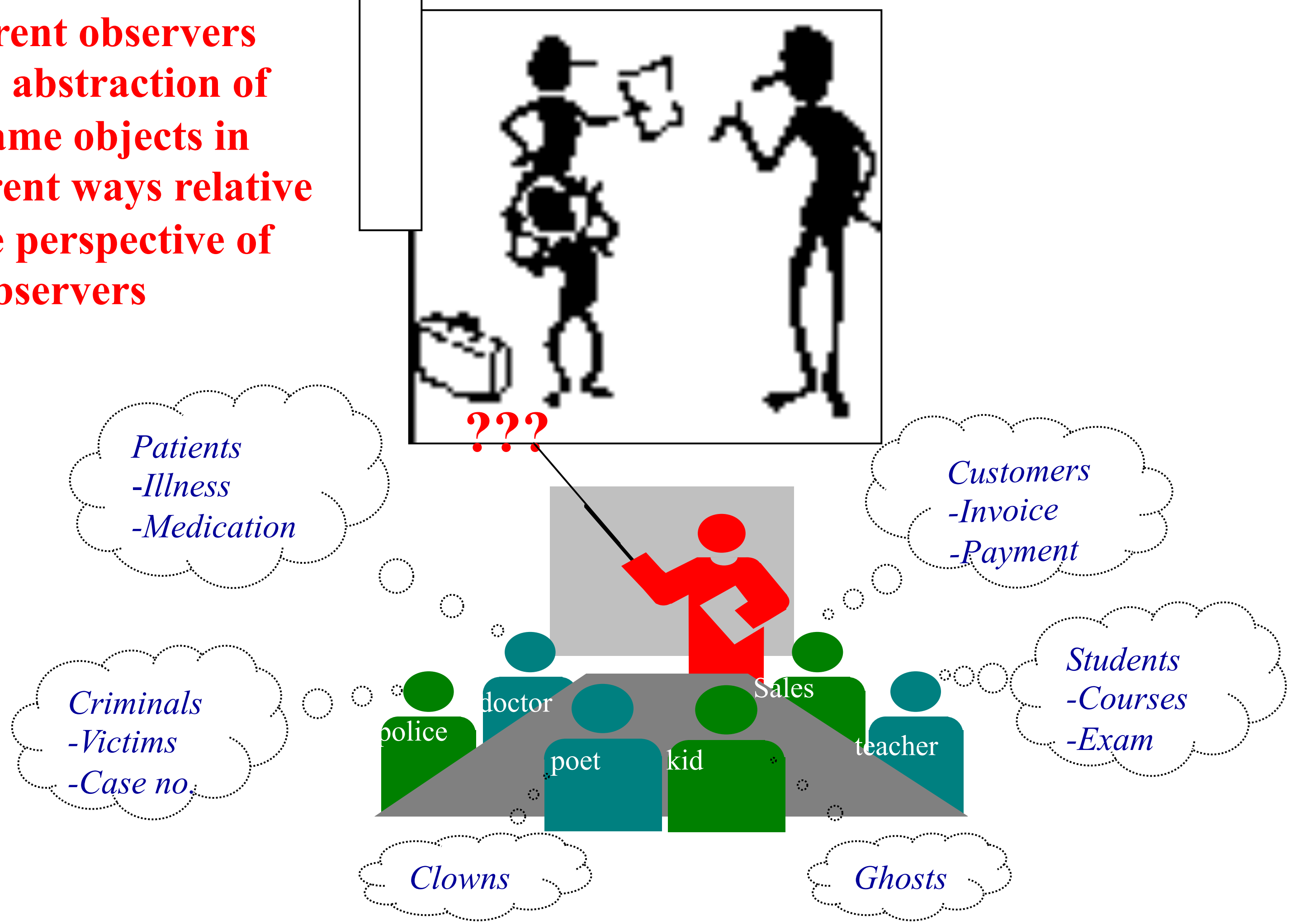
- Abstraction Example -
Same Object but Two Different Abstractions



- Source: Booch, G.: Object Oriented Analysis and Design with Applications, Addison-Wesley, 1993, 2nd Edition . Chapter 2.

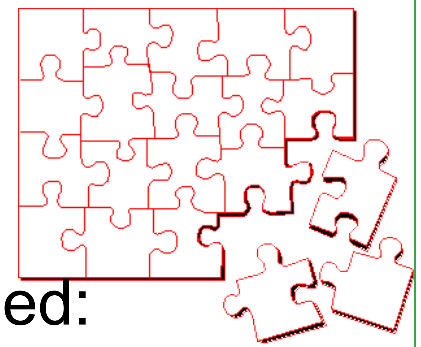
Various Abstractions

Different observers make abstraction of the same objects in different ways relative to the perspective of the observers



Modularity

- **To reduce complexity, we need to break a program into smaller pieces**
  - Facilitate the design, implementation, operation and maintenance of large programs
  - Permits reuse of logic
  - Ease **maintainability** and understandability



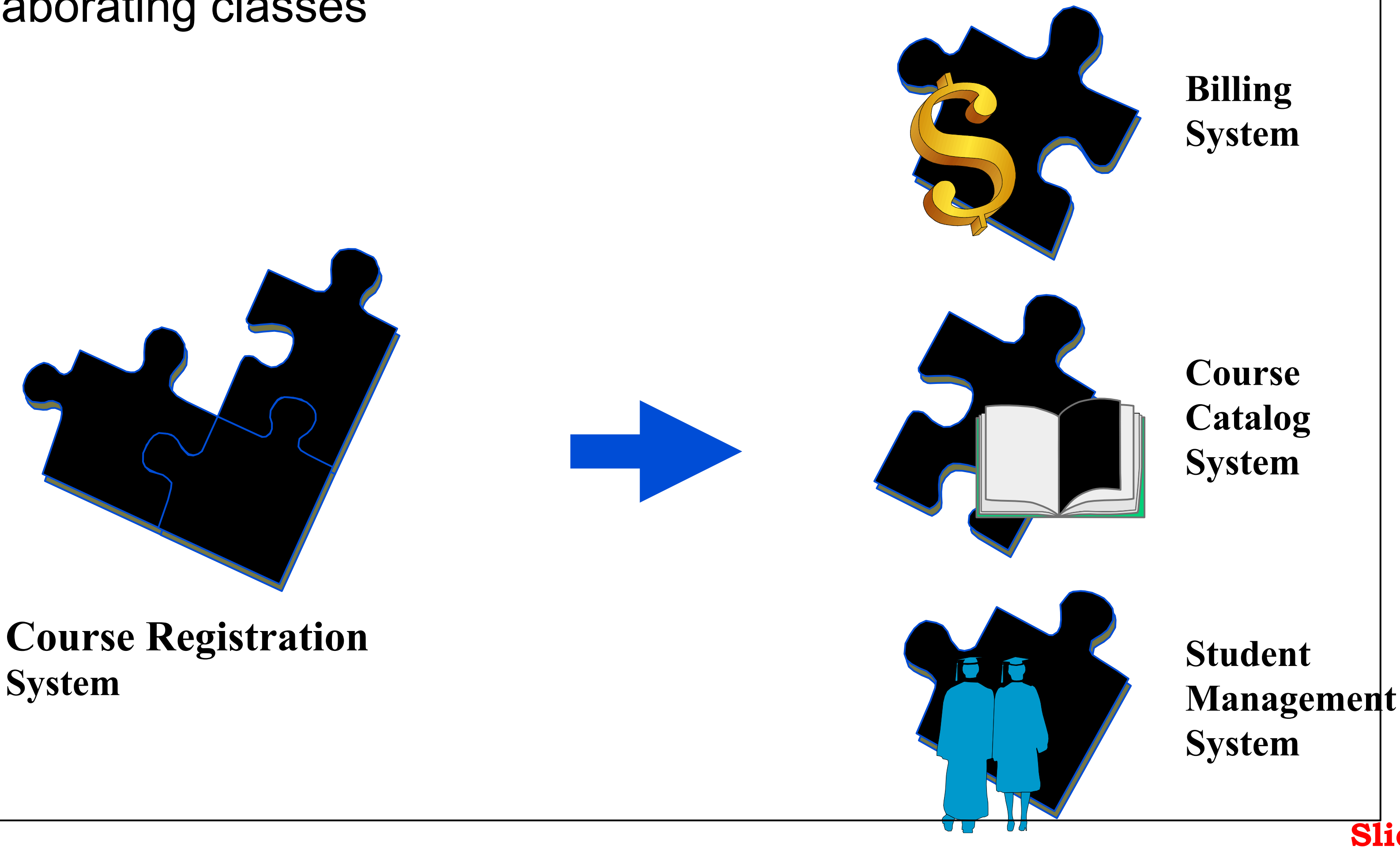
- Object-oriented decomposition is widely used:

=> We think of a program as a set of autonomous objects

$\{O_0, O_1, \ldots. O_n\}$ that collaborate to fulfill the requirements

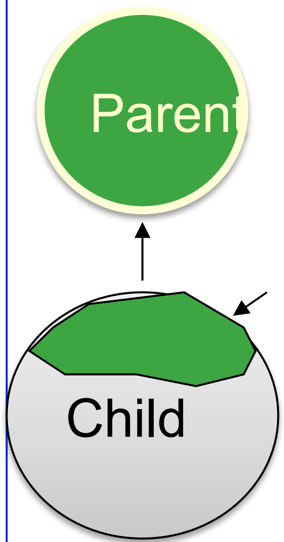
Example: Modularity

- For example, break complex systems into smaller components.
- Each component is composed of set of collaborating classes



Inheritance (Hierarchy)

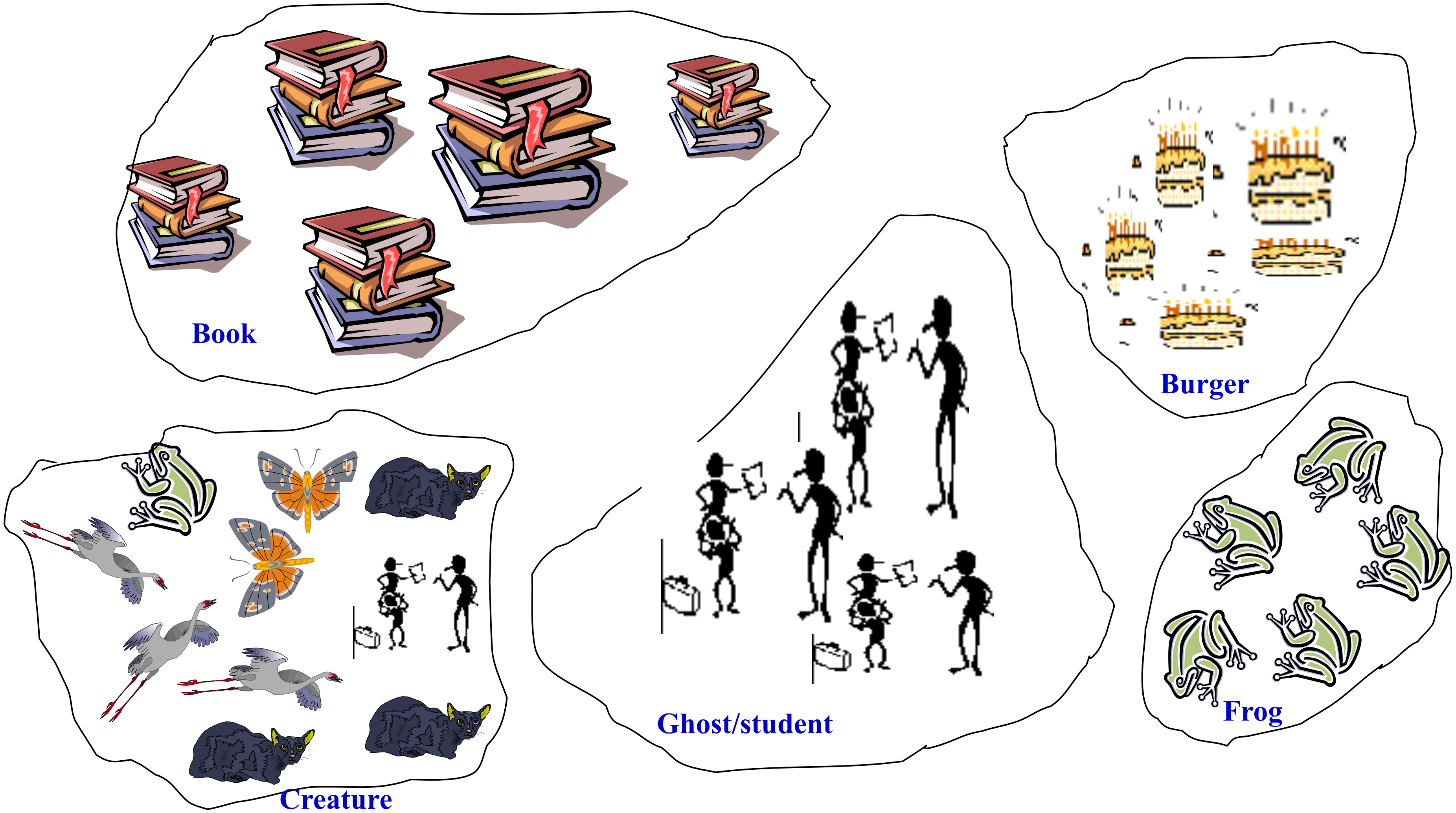
- Remember Generalization?
- Inheritance is the most important kind of hierarchy in O-O decomposition
- Inheritance organizes classes in inheritance hierarchies
  - A subclass inherits its parent's attributes, methods, and relationships.
- Inheritance is the sharing of properties or features among classes based on a hierarchical relationship
- Inheritance represents a hierarchy of classes.



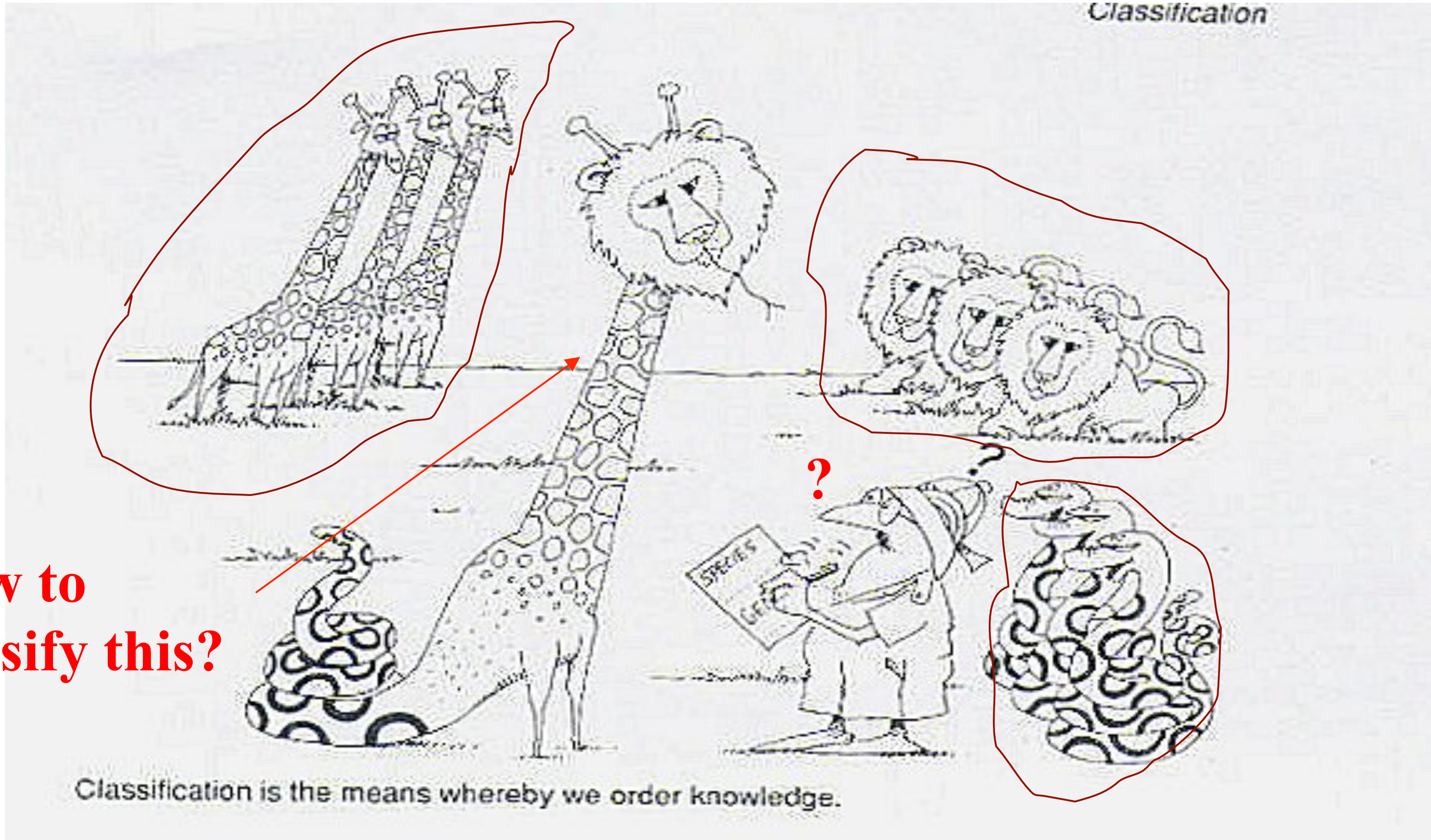
Classification

- Classification is the means whereby we order knowledge
- Recognizing the sameness among things allows us to identify the commonality within key abstractions
- Classification means that objects with the same properties and behaviour are grouped into a class/object type
- Each class describes infinite set of individual objects
- Classify is highly dependent upon
  - the reason for the classification (why do we do classification), and
  - the criteria used for the classification (how do we do classification)

Classification



Classification Example

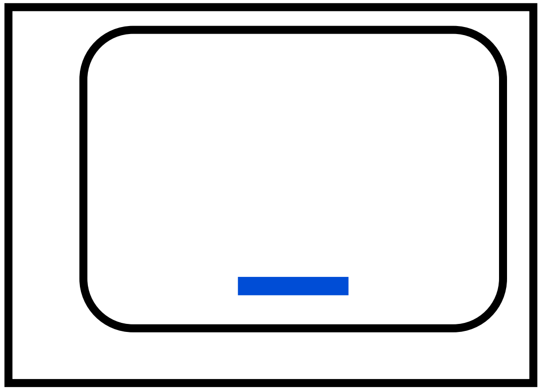


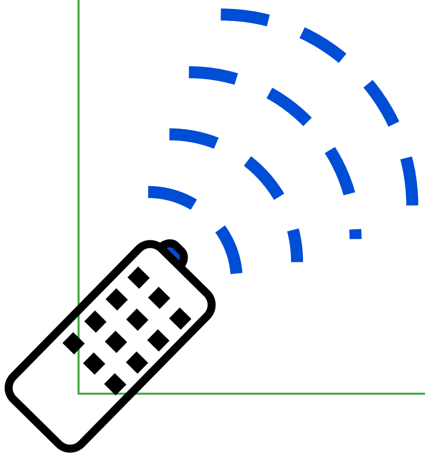
How to classify this?

- Source: Booch, G.: Object Oriented Analysis and Design with Applications, Addison-Wesley, 1993, 2nd Edition. Chapter 4.

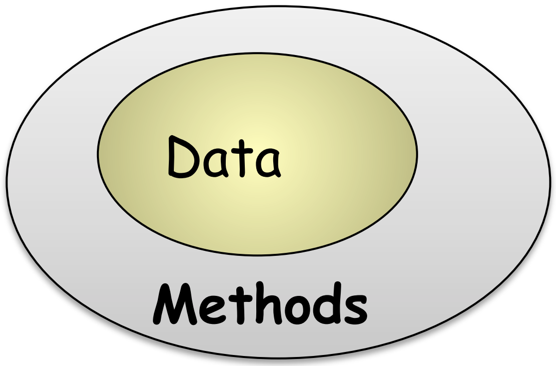
Encapsulation

- Combining the data and methods in the same entity
- Hiding implementation from clients.
  - Clients access the object via public interface
- It prevents others from seeing the inside view of an object —-also known as **information hiding**



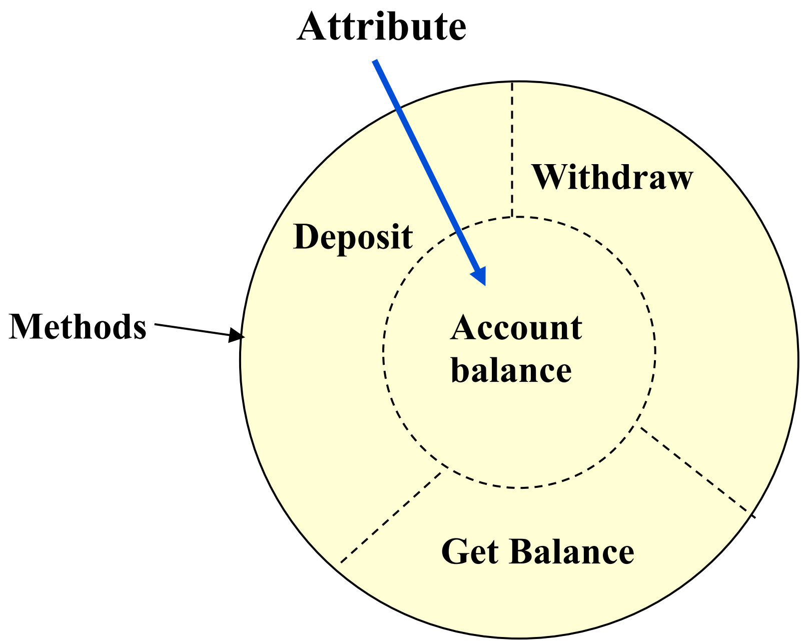


Improves the resiliency of the system, i.e. its ability to adapt to change



Encapsulation - Example

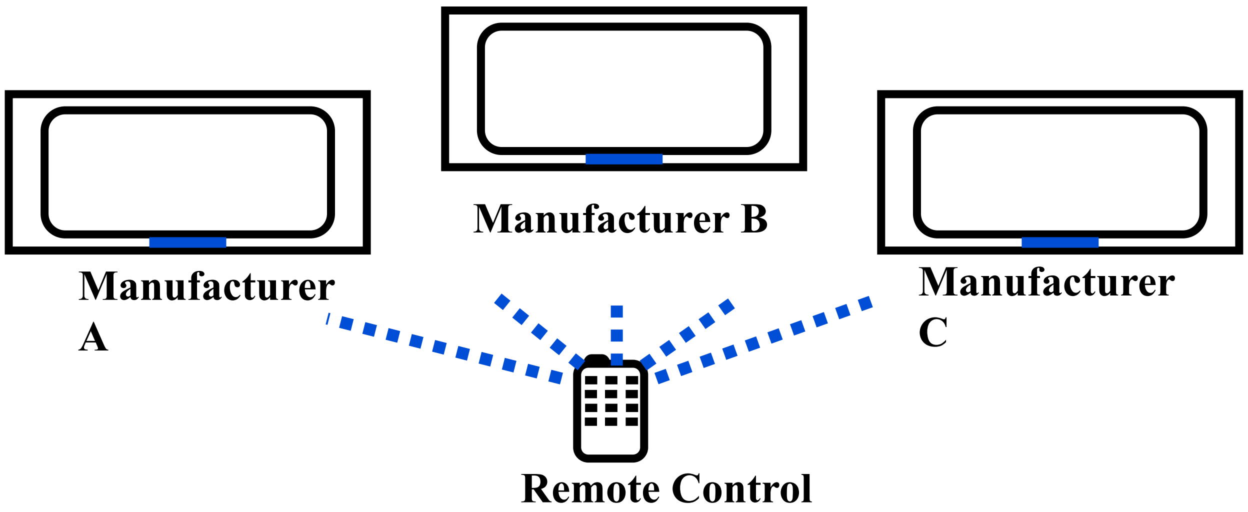
```
class Account {
  private String accountName;
  private double accountBalance;

  public  withdraw();
  public  deposit();
  public  getBalance();
  } // Class Account
```



Bank Account Object

Polymorphism

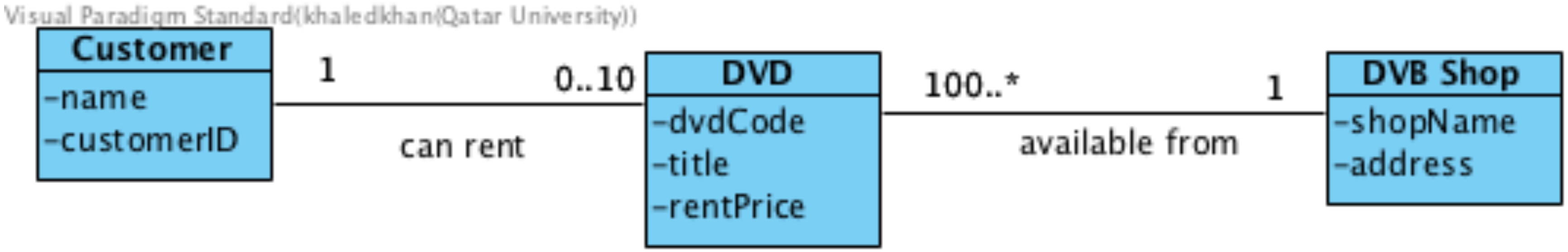
- The ability to hide many different implementations behind a single interface:
  - The capability of a method to do different things based on the object that it is acting upon.
  - Overloading and overriding are types of polymorphism.



Domain Model Elements

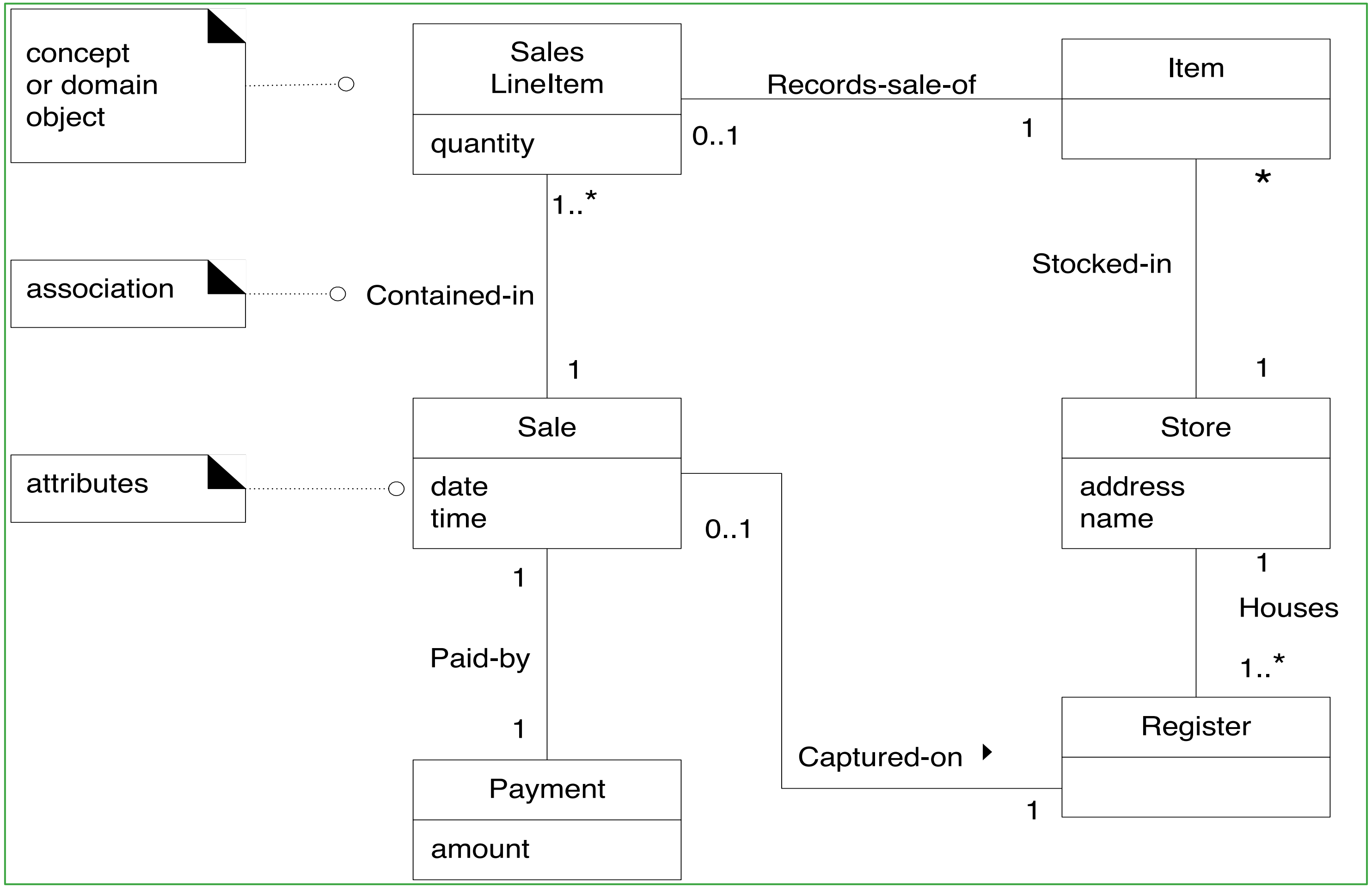
- A *Domain Model* visualizes, using **UML class diagram** notation, or domain objects.
  - It is a kind of “visual dictionary” of concepts & their relationships
    - A concept is an idea, thing, or object
  - Represents real-world concepts, not software classes and their responsibilities
- In a domain model, we have **four** types of elements:
  - **Conceptual class** (or **domain object**): which identifies a business entity or concept (typically noun), e.g. shop, video CD, member, etc.
  - **Associations between conceptual classes:** which define relevant relationships, those that capture **business information that needs to be preserved**, e.g.
    - A shop has many video CDs,
      - shop, video CD are domain objects (concepts)
  - **Attributes:** which are logical data values of a domain object, e.g. each club member may have a **membership_Number**
    - **Membership_Number** is the attribute of the domain object member.
  - **Multiplicity:** The degree of relationship between two domains objects/concepts
    - A member borrows **many** video CDs. **One** video CD can be borrowd by only **one** member
      - Has and borrow make the association between domain objects
      - Many is the multiplicity. One is the multiplicity

EXAMPLE: Partial DVD Renting Store Domain Model



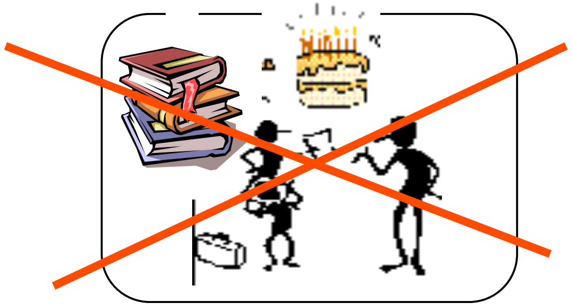
- Three conceptual class/domain concept:
  - **Customer**
  - **DVD**
  - **DVD Shop**
- Attributes:
  - **name, customerID;**
  - **dvdCode, title;**
  - **shopName, address**
- Two associations/relationships:
  - **can rent**
  - **available from**

Example: Partial Point-Of-Sale Domain Model

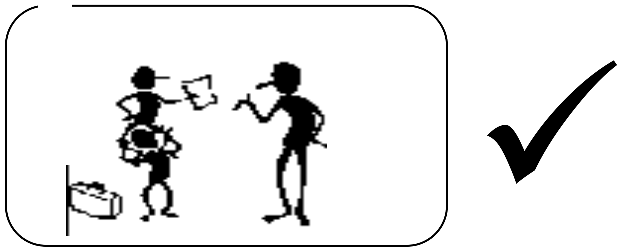


Object vs. Class

- A object must be uniquely identifiable and it must have state
  - My book, this pen, New York
- A class is a structure of similar objects, a single object is not identified
  - Pen, Book, City.
- An object is not a class, objects that share no common structure and behaviour cannot be grouped in a class;



Not A Class; a group of unrelated objects



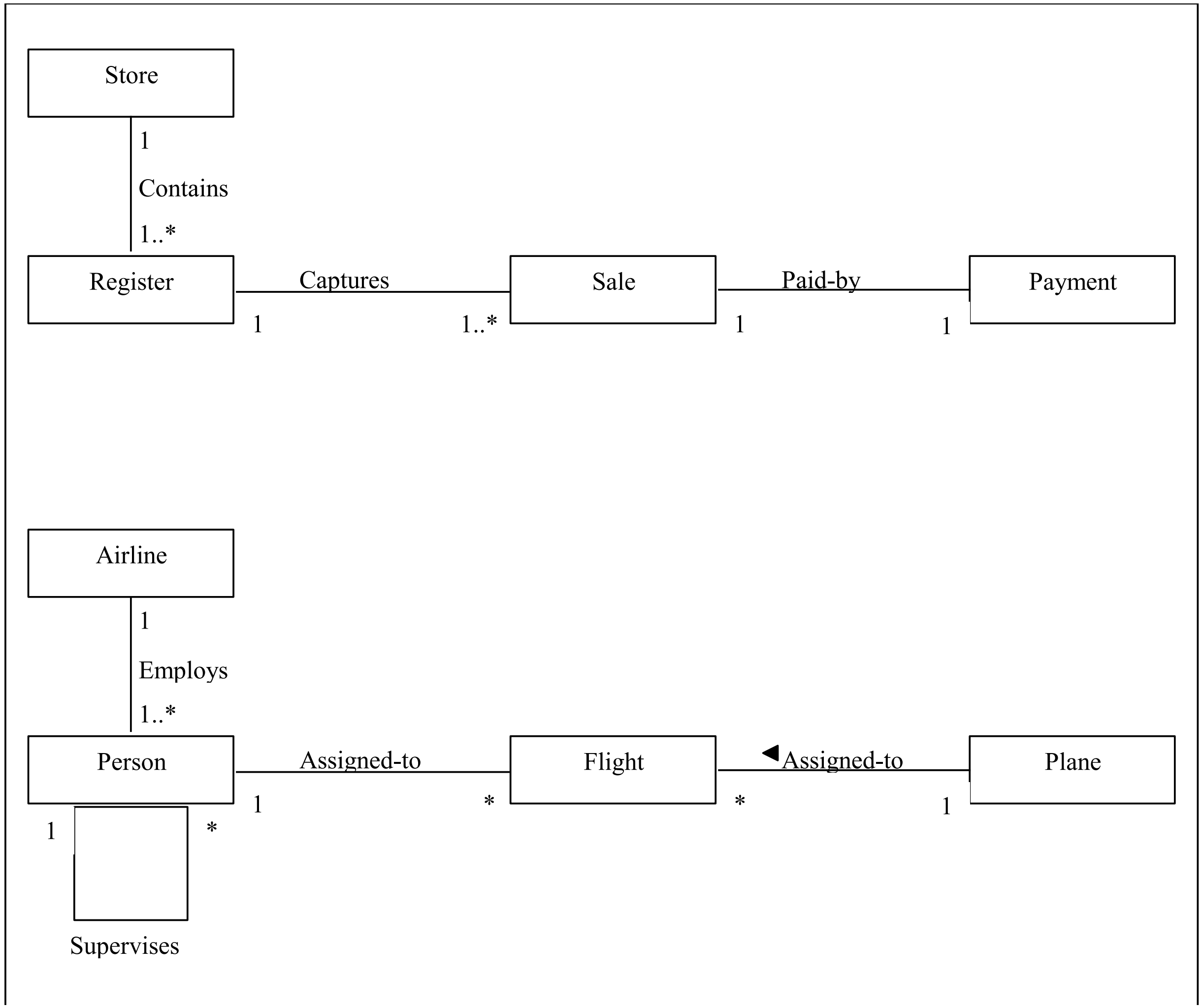
A Class

In some conventions and notations such as UML,

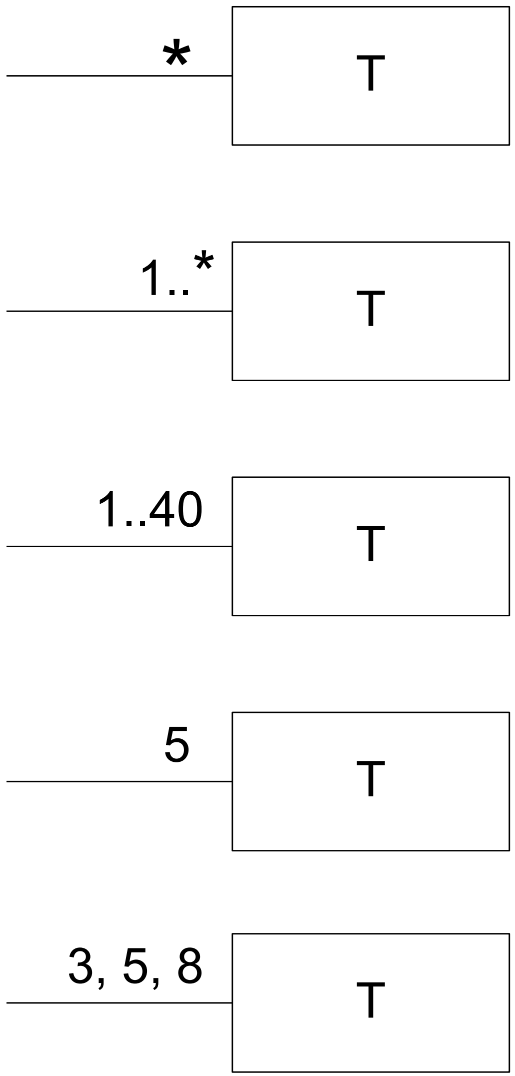
- **Properties are called attributes; age, date, name, marks**
- **Behaviours are called operations/methods; find, get, calculate, stop**

Associations

- Association is the relationship between domain concepts/classes
- Examples: **Captures; Paid-by; Assigned_to**
- Association can be recursive, that means an association can be related to the class itself; example: **Supervises**



Multiplicity



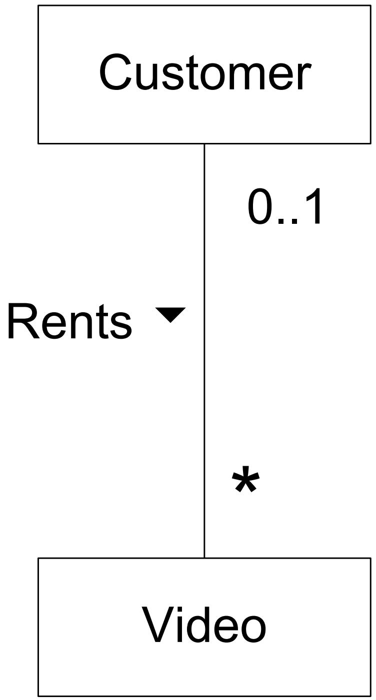
zero or more; "many"

one or more

one to forty

exactly five

exactly three, five or eight

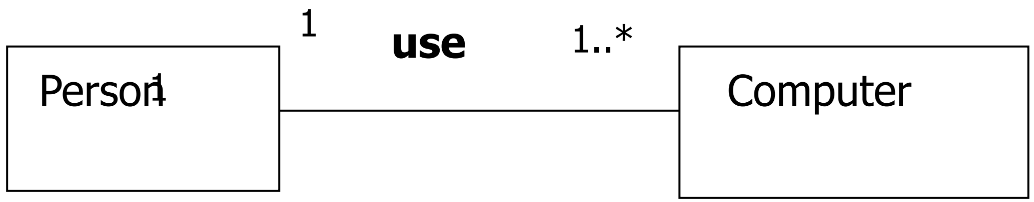


One instance of a Customer may be renting zero or more Videos.

One instance of a Video may be being rented by zero or one Customers.

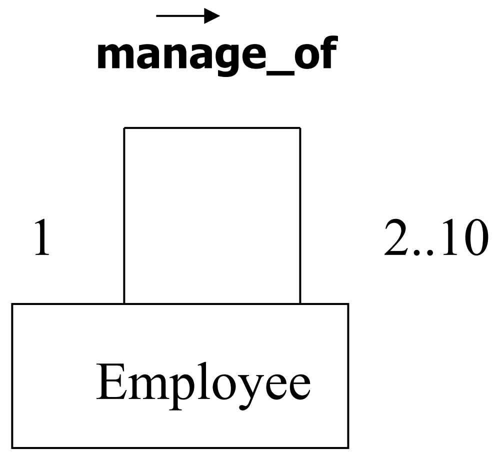
Normally, the multiplicity at a particular moment in time

Recursive Association

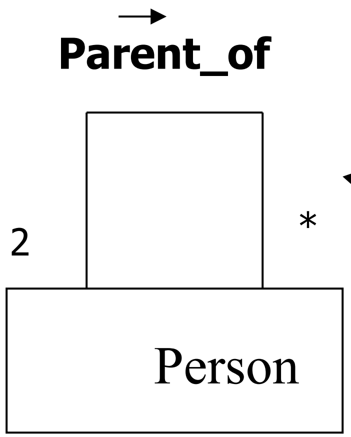


A person uses at least **one** computer or more.

A person may drive zero or many cars.

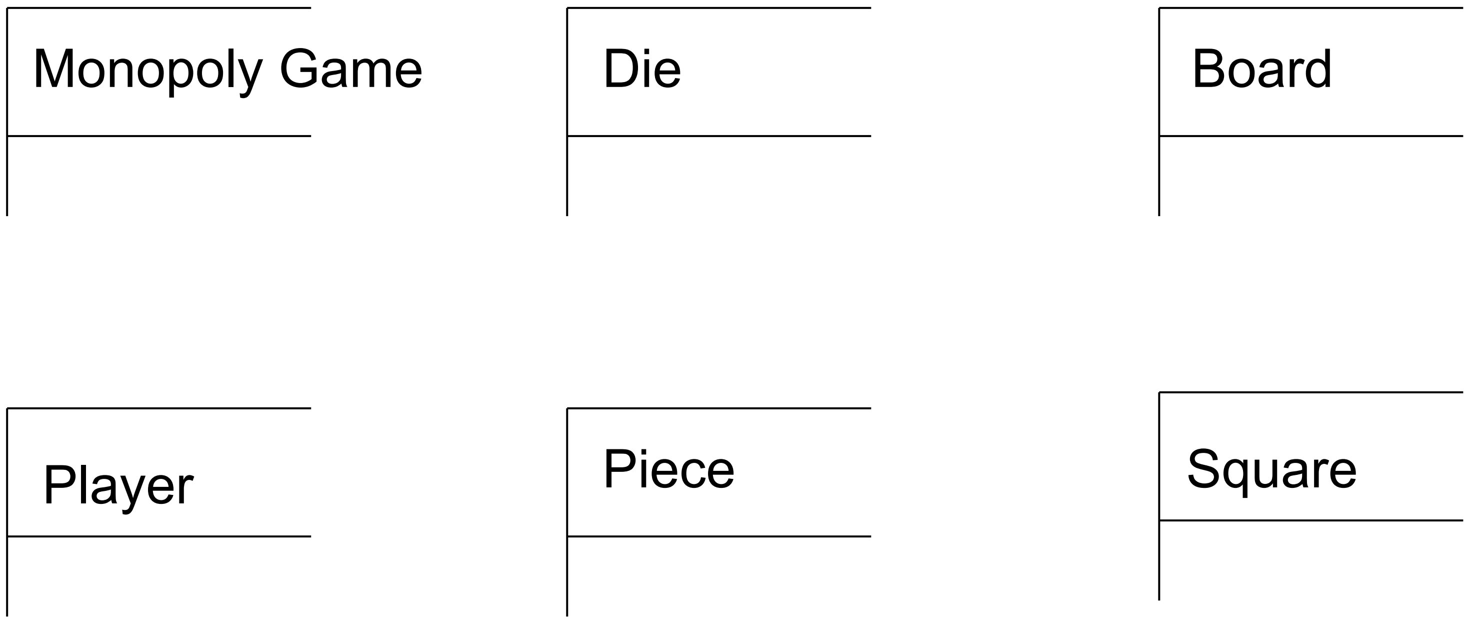


An employee manages at least 2 and maximum 10 employees. One employee is managed by only one employee

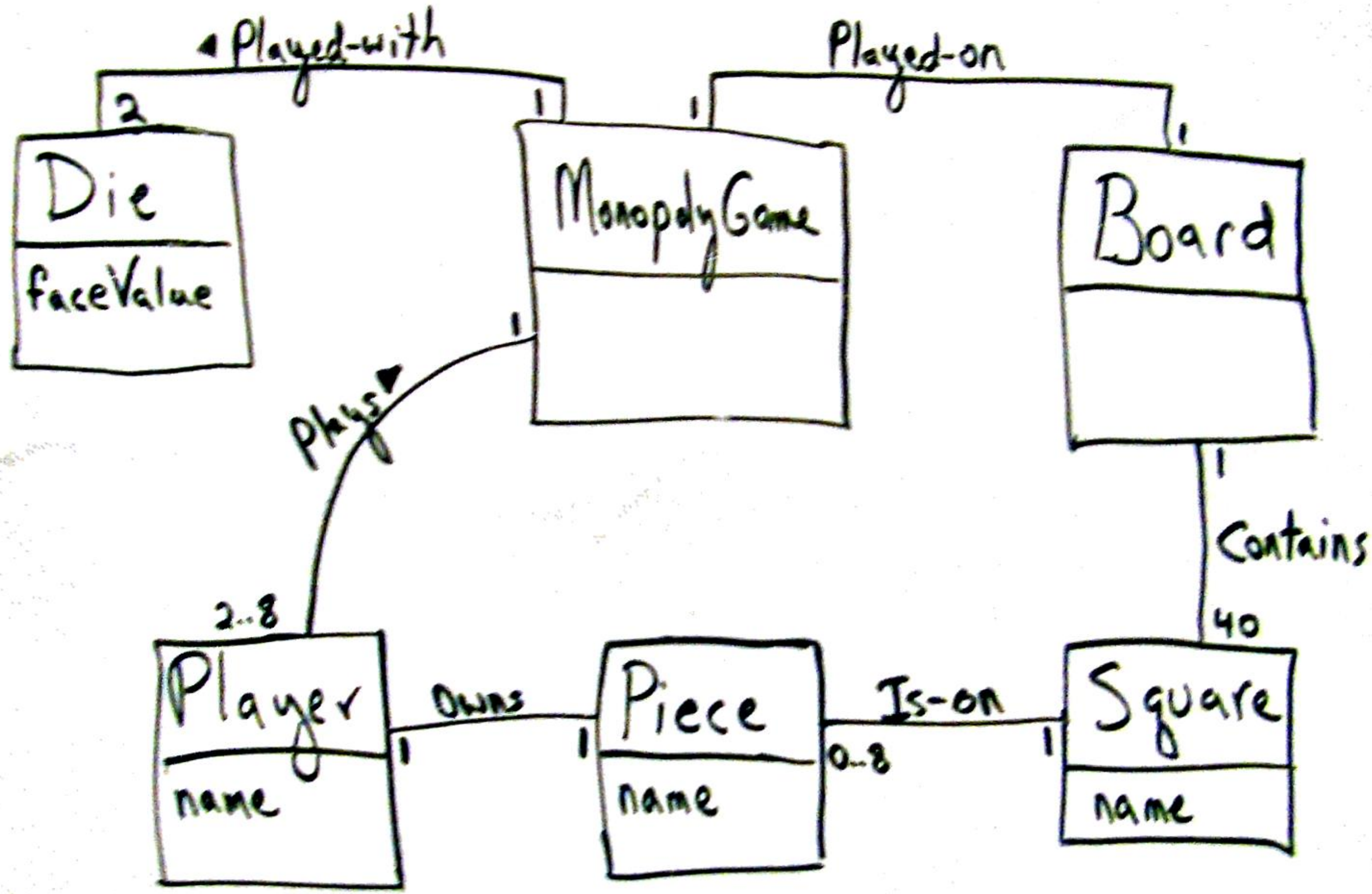


Recursive call: an object call other objects in the same class, but the role of the object is different

Example:
Monopoly Game Domain Model
(first identify concepts as classes)

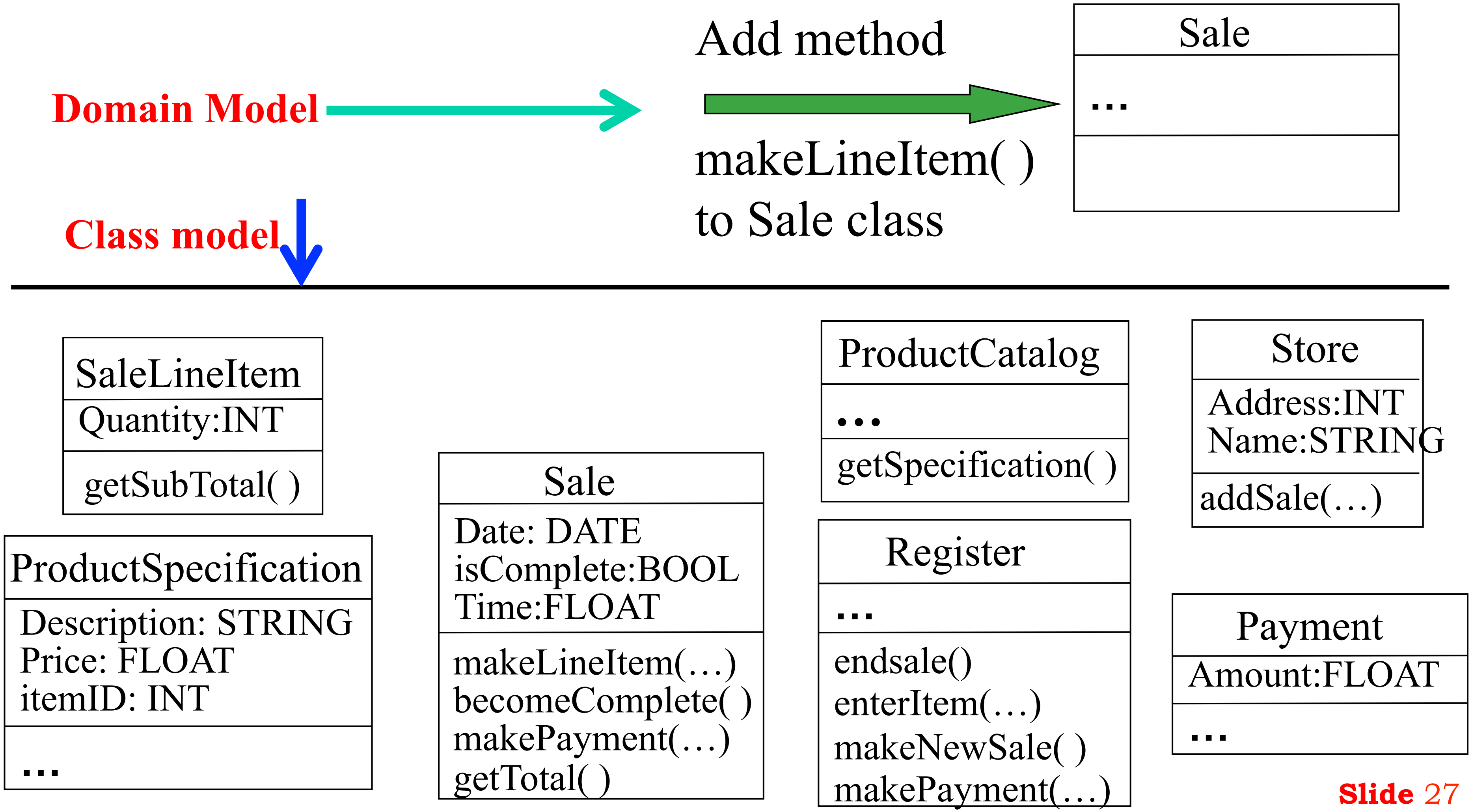


Monopoly Game Domain Model



Making Design Class Diagram from Domain Model

- *We add the methods of the identified classes*



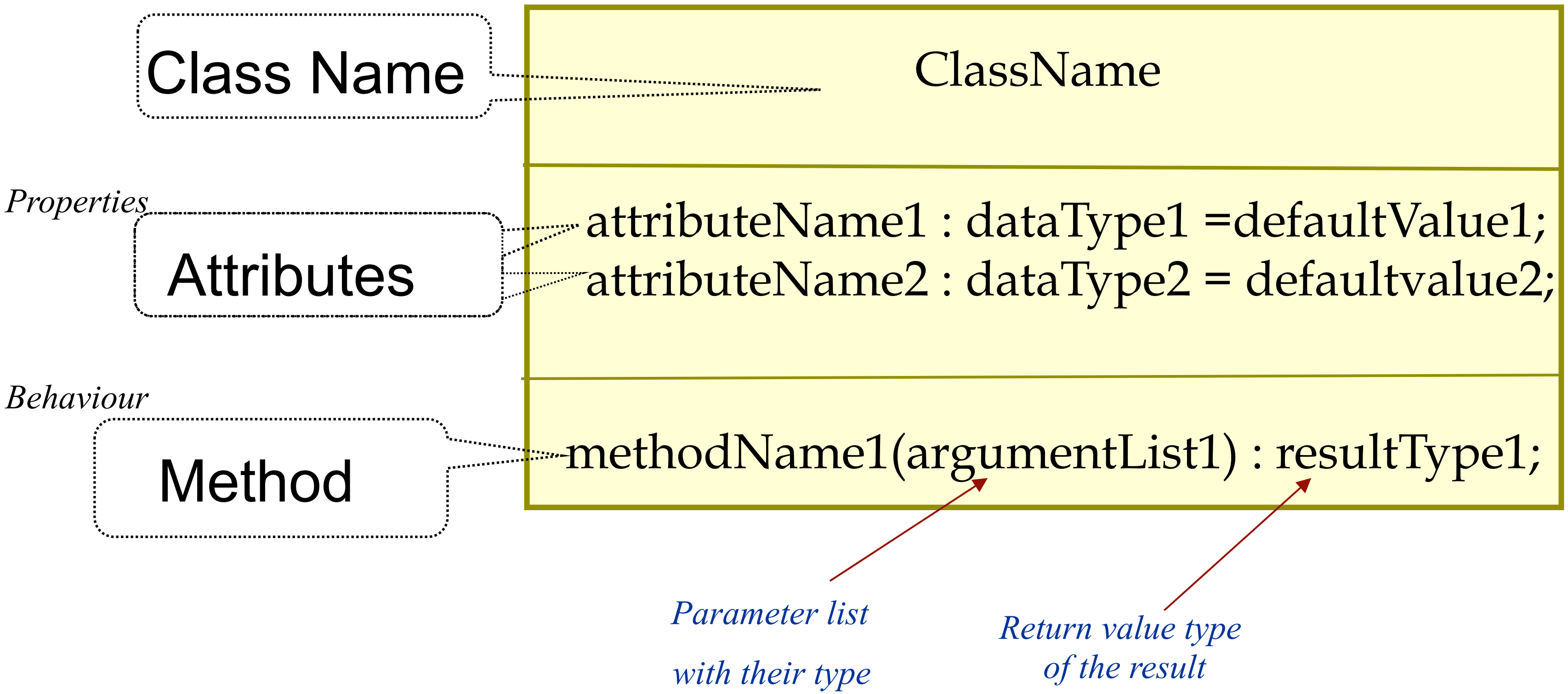
Design Class Diagram

- A class describes a group of objects with the same types of some or all
  - **Properties (attributes),**
  - **Behaviour (operations), -adding to the domain model**
  - **Kinds of relationships (associations), and**

- A class is **a set of** objects that share a common structure (properties) and a common behaviour (operations)

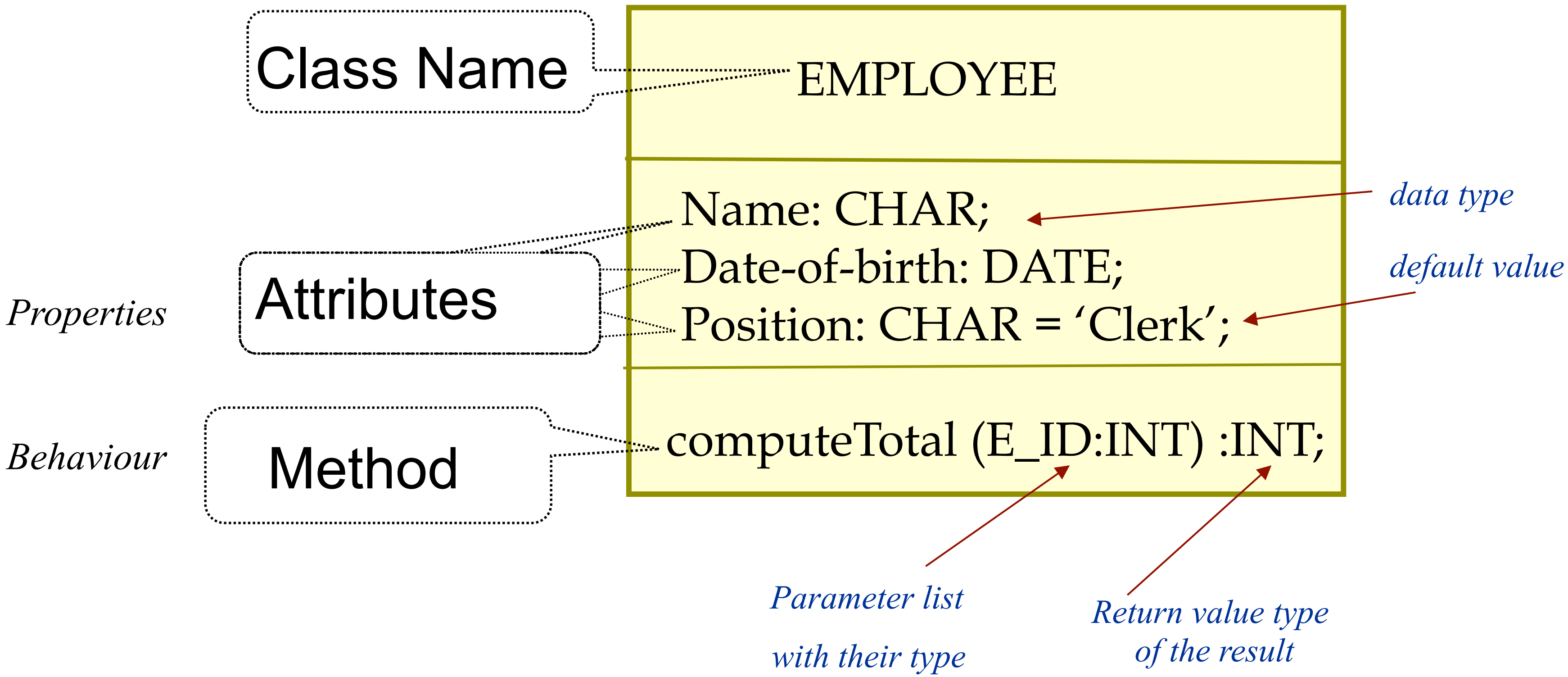
- If objects are the focus of O-O modelling, why we need class?
  - By **classifying** objects into classes, we abstract a problem
  - **Abstraction** gives modelling its power and ability to generalise a group of similar objects

Class Representation in UML



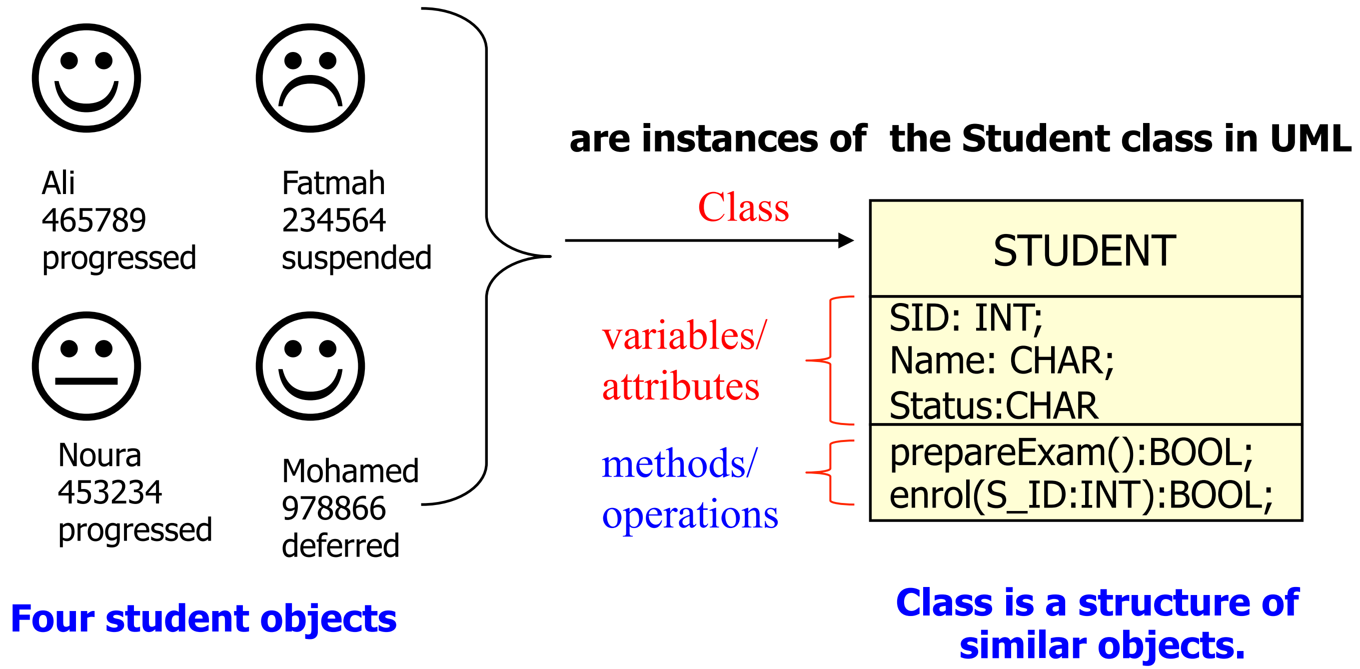
- An attribute should describe values, not objects
- Unlike objects, values lack identity. Types of values should be specified e.g., string. date, integer etc.

Example: Class in UML



Class and objects

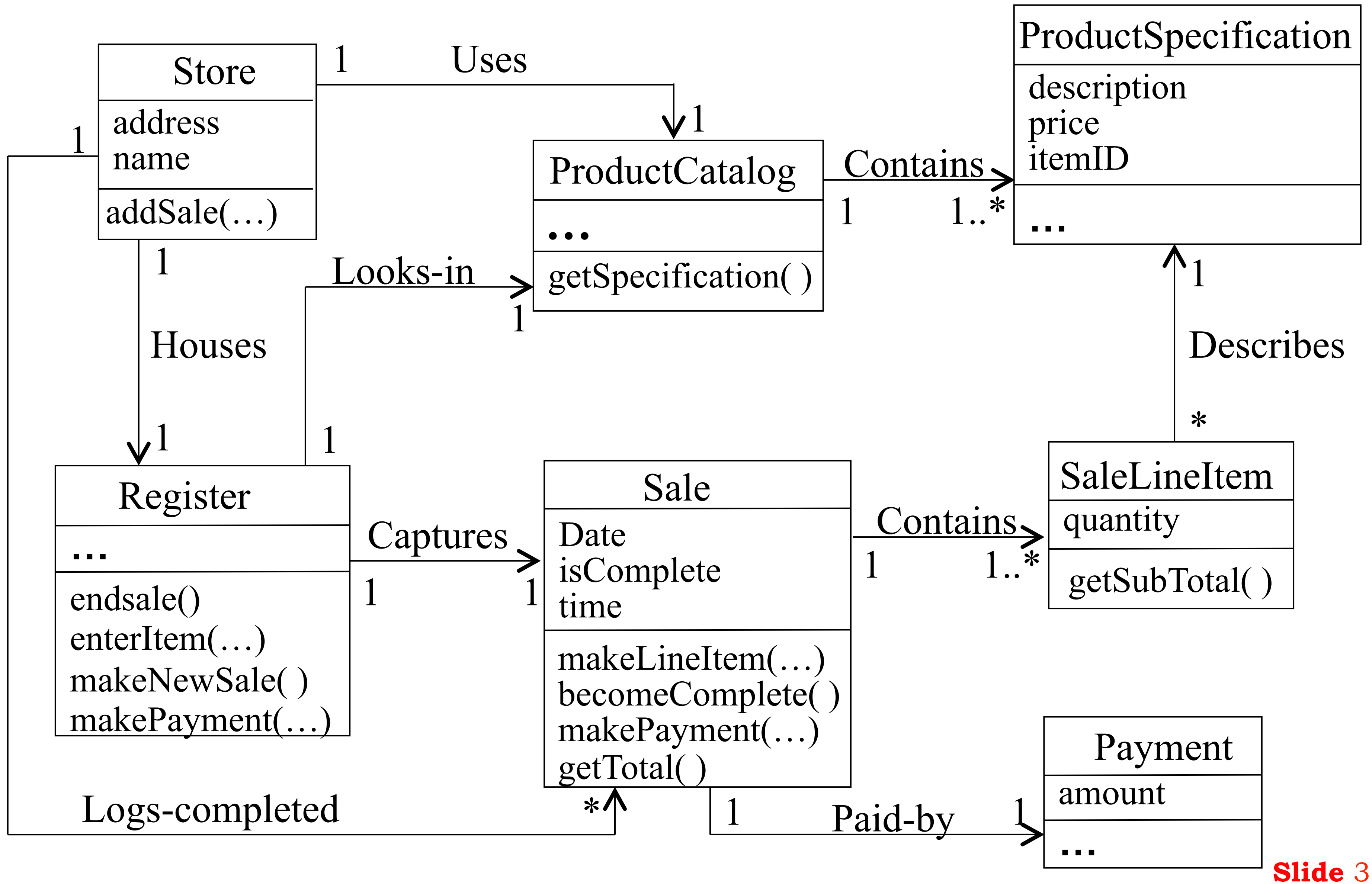
- A object must be uniquely identifiable and it must have state
  - my book, this pen, student Fatima,
- A class is a structure of similar objects, a single object is not identified
  - Pen, Book, City.
- An object is not a class, objects that share no common structure and behaviour cannot be grouped in a class;



Behaviour - Operation

- **Behaviour:**
  - Behaviours are the services (general functions) that an object (an instance of a class) performs (providing or receiving services) in a system.
  - Each object is responsible for some operations in the system it is in.
  - An operation is a function that may be applied to or by objects in a class
- **Operations/Methods**:
  - When behaviours are encoded in an O-O design notation such as in UML, they are referred to as *Methods*
  - Methods specify the way in which an object's data is manipulated
  - A method is the implementation of an operation for a class.
  - When an operation has methods on several classes, the methods all have the same **signature**
  - The signature is the *number* and *types of arguments* (parameters) and the *type of result values* (return values)
- **Examples of methods**:
  - In a class 'Employee'
    - a method can be "findSalaryRate"
    - a method can be "computeTotal"

Point of Sale Design Class Diagram



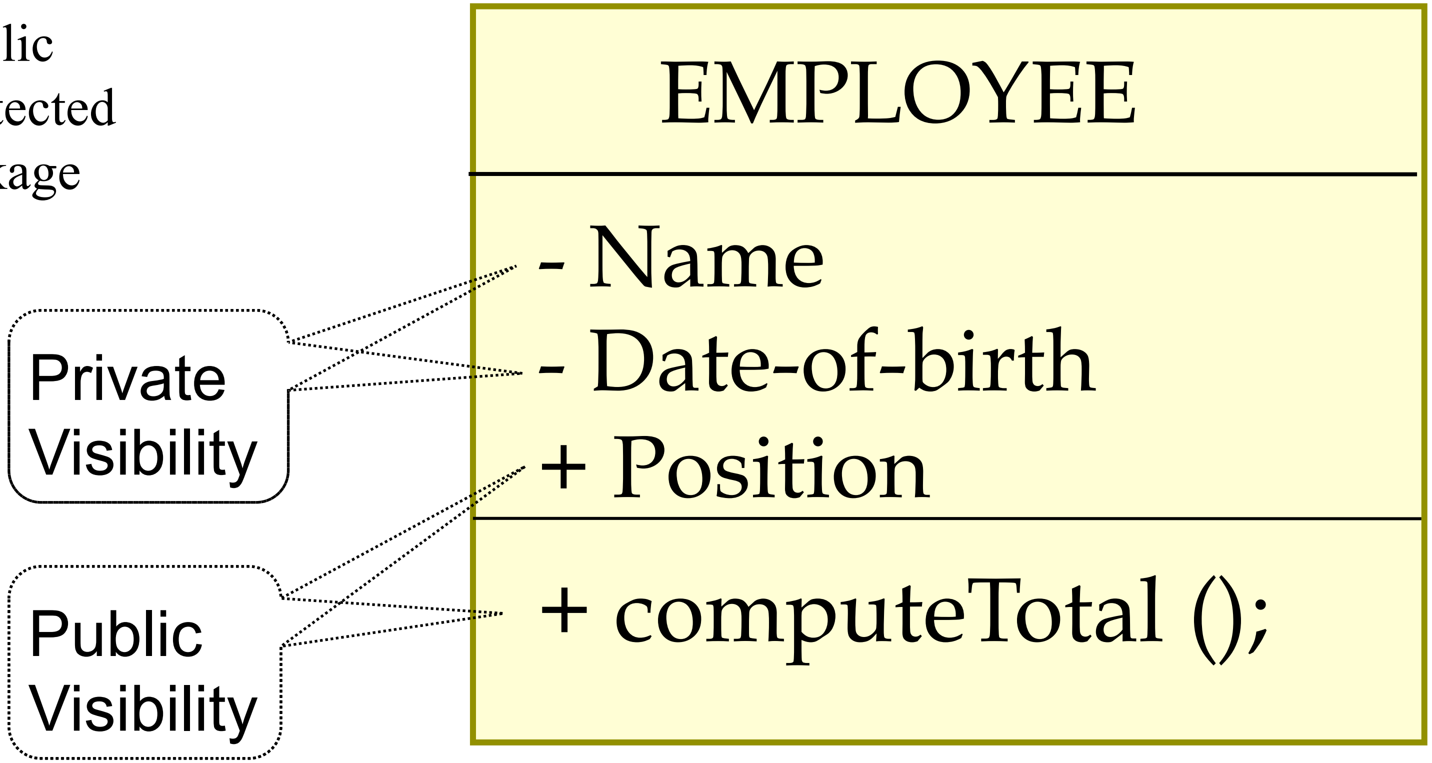
Visibility of Properties

- Visibility refers to the ability of a method to reference a feature from another class --possible values:
  - ***Public***: Any method can freely access public features
  - ***Protected***: Only methods of the containing class and its descendants via inheritance can access protected features
  - ***Private*****:** Only methods of the containing class can access private features
  - ***Package***: Methods of classes defined in the same package as the target class can access package features
- We must understand all public features to understand the capabilities of a class
- We may ignore private, protected and package features because they are merely an implementation issue

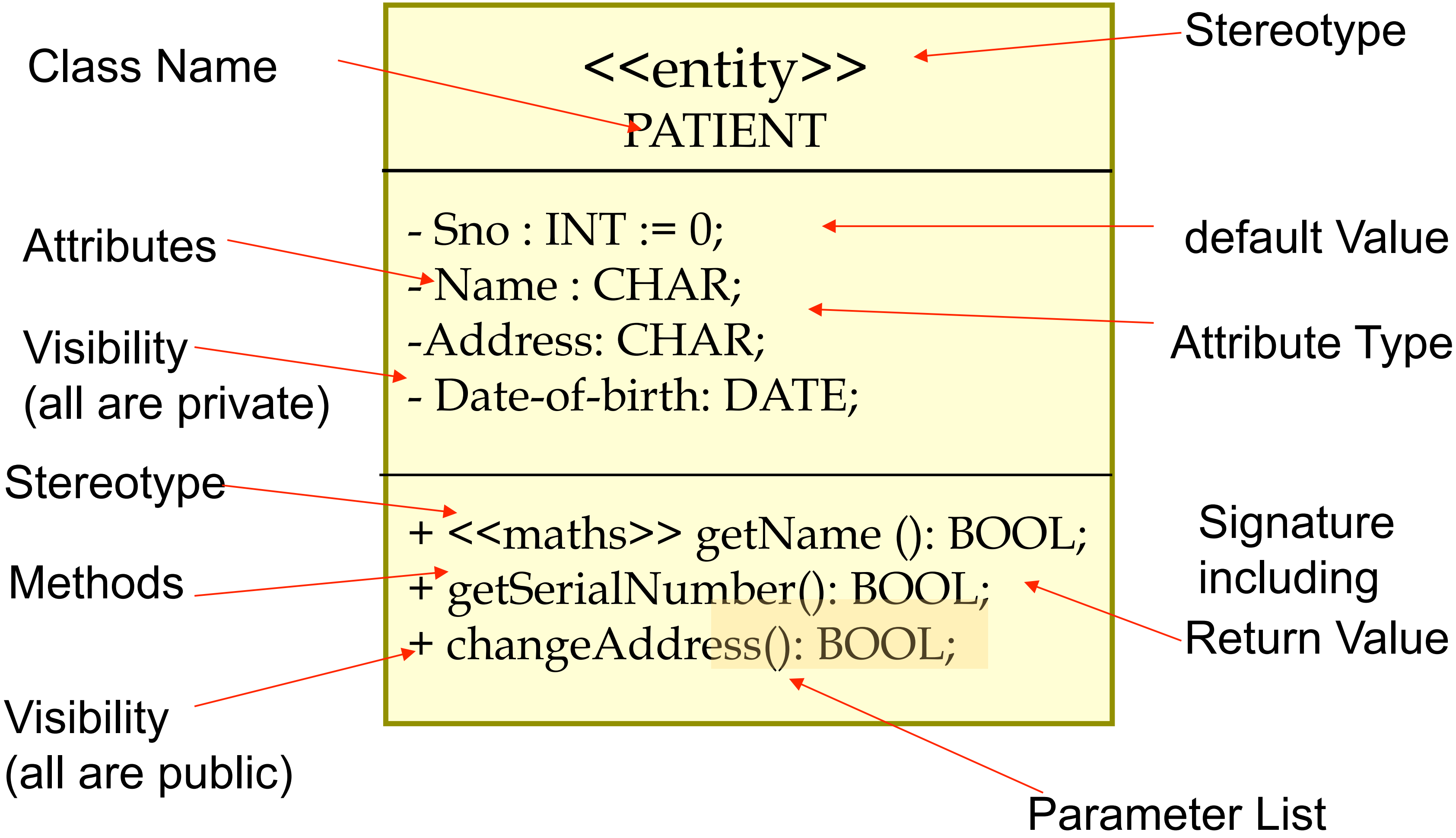
Visibility of Properties in UML

UML legends for visibility:

- \- Private
- \+ Public
- # Protected
- ~ Package



Detailed Class Definition in UML



Sourse: (Unhelkar 2005)

Some Criteria for Refining Classes

- **Redundant classes**: if two classes express the same concept, we use one of them which is most descriptive: e.g., Customer, client, user
- **Irrelevant classes**: If a class has little or nothing to do with the problem, eliminate it. The class could be important in another class: e.g., cost
- **Vague classes**: A class should be specific, not to be too broad in scope or ill-defined boundaries: e.g., system, security
- **Attributes**: Names that particularly describe individual objects, e.g., name, birth date
- **Operations**: If a name describes an operation that is applied to objects and not manipulated in its own right, e.g., checking passport
- **Objects/actors**: The name of a class should reflect its intrinsic nature and not an object or a actor that it plays in an association, e.g., Student Asma, her car.
- **Implementation constructs**: Features that are too implementation specific, e.g., Communication Line

Exercise: Refinement of Classes

- During the analysis phase, the following 28 candidate classes have been extracted from our knowledge of a flight reservation system

Reservation

Receipt

Transaction data

Booking number

Security

System

Communications Line

Cost

Software

Network

Amount

User

Cashier

Access

Transaction log

Server

Travel agency

Customer

Schedule

Qatar airways

Ticket

Airlines

Airport

Account

Flight number

Manager Ahmed

Reserve flight

Verify schedule

Solution: Refinement of Classes

During the design phase, the following have been identified according to the following criteria: **redundant classes, irrelevant classes, objects/.actors, vague classes, attributes, operations, and implementation constructs.**

Bad classes

Vague
- System
- Security
- Network

Attributes
- Transaction data
- Booking number
- Amount
- Flight number

Objects/ actors
- Qatar airways
- Manager Ahmed
- Cashier

Operation
- Verify schedule
- Reserve flight

Implementation constructs
- Communications Line
- Software
- Access
- Servers

Irrelevant
- Cost

Redundant
- User
- Transaction log

Refined classes

- Customer
- Ticket
- Receipt
- Schedule
- Airport
- Reservation
- Travel agency
- Airlines
- Account

Actor vs. Class in Class Diagram

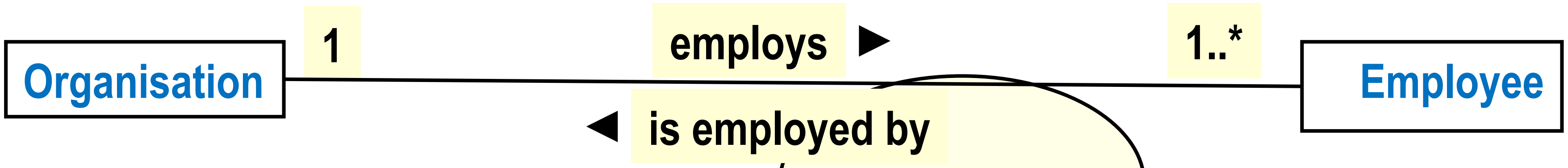
- An actor in a use case diagram can only be defined as a class in the class diagram if the information of the actor is manipulated/used within the system
- An actor **cannot** be an object of the system if it is not manipulated/saved/used within the system.
  - In that case, the actor is just a user, not an object
- Example
  - A student of Qatar university is an actor and also an object of the Qatar University Web based system
    - Why?
  - A visitor of QU Web based system is only an actor, not an object
    - Why?

Class Relationships

- Classes do not exist by themselves, but exist in relationships with other classes.
- Three basic kinds of relationships:
  - **Generalisation**
    - Denoting ‘a kind of’ relationship and capturing **inheritance** properties through hierarchy
    - A car is a kind of vehicle
    - A car is a specialised subclass of the more general class, vehicle
  - **Association**
    - Denotes some semantic connection among otherwise unrelated classes
    - Persons and cars are largely independent classes, but cars are driven by persons
  - **Aggregation**
    - Denotes ‘a part of’’ relationship
    - A fuel tank is not a kind of a vehicle, it is a part of a vehicle
  - **Composition**
    - Much stronger version of aggregation.

- **Classification** helps us to identify generalisation, aggregation and association among classes
- **Classification** helps us to split a large class into several specialised classes, or create one larger generalised class by uniting smaller specialised classes
- **Classification** may even discover previously unrecognised commonality, and create a new class
- **Abstraction** is also used to establish generalisation relationships among classes
- **Hierarchy of classes** can be used to make generalisation relationships among classes

Association : UML Notation and Typical Implementation




```
class Organisation
{
   List<Employee> employees;
   . . . . .
}

class Employee
{
   Organisation qatar_university;
   . . . . .
}
```

Aggregation

- Aggregation : (hollow diamond).
  Parts may ***exist independent of the whole***

e.g. **Employees may exist independent of the team**.

- Aggregation represents a relation “contains”, “is a part of”, “whole-part” relation.
  - Part instances can be added to and removed from the aggregate

Composition

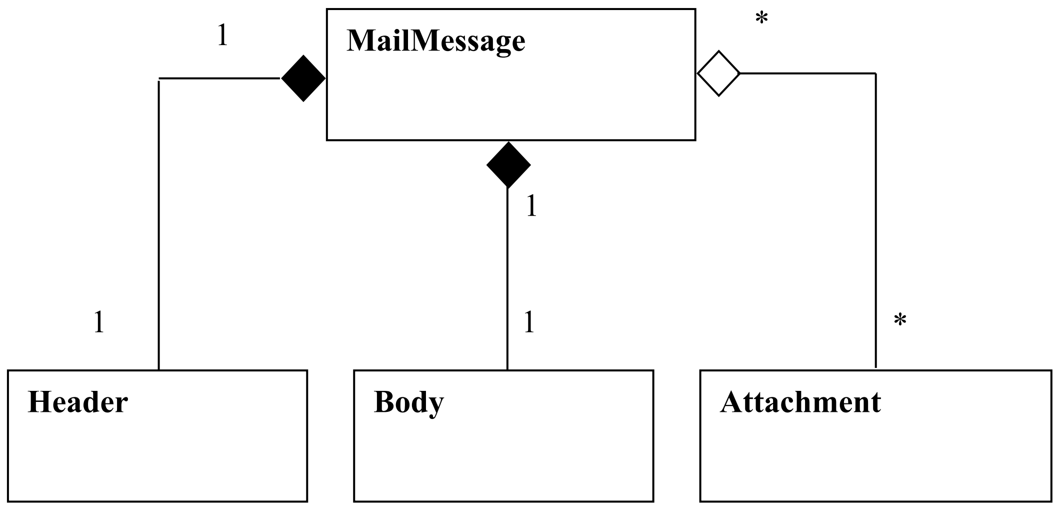
Composition : (filled diamond)
Every part may belong to only one whole, and If the whole is deleted, so are the parts

- Stronger than an aggregate
- Often involves a physical relationship between the whole and the parts, not just conceptual
- the part objects are created, live, and die together with the whole: **the life cycle of the 'part' is controlled by the 'whole'.** Part cannot exist independent of the whole.

e.g. Each building has rooms that can not be shared with other building!

Aggregation vs. Composition Example 1

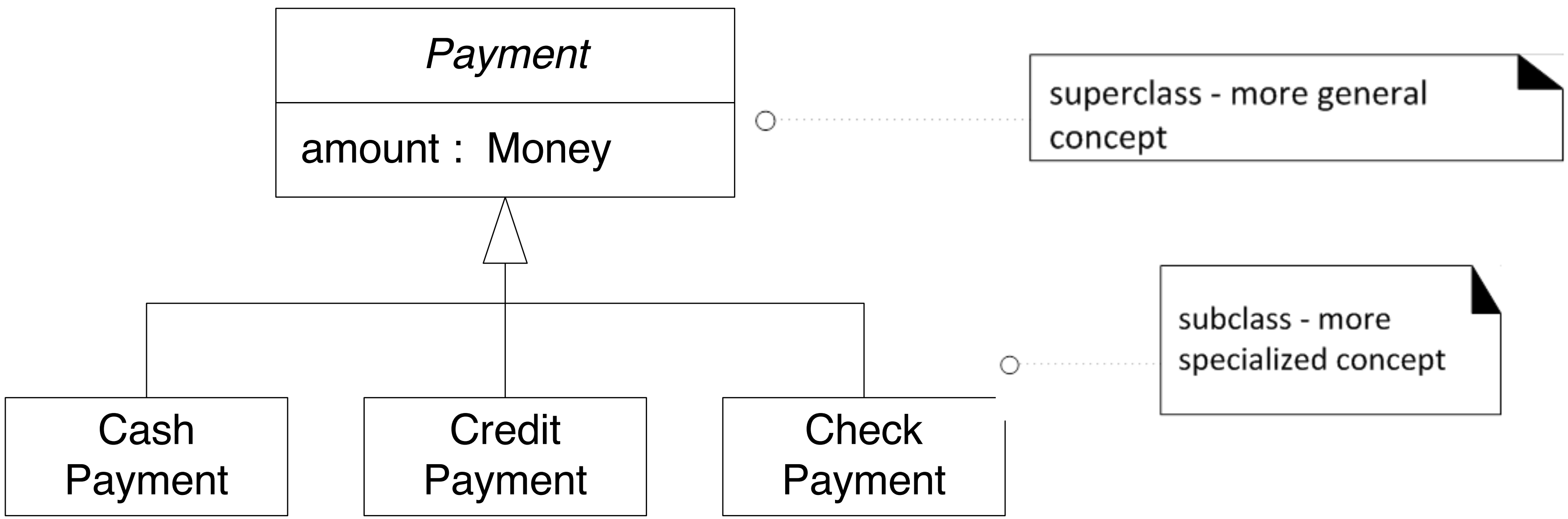
We could model the mail message example using composition and aggregation.



- When a MailMessage object is destroyed, so are the Header object and the Body object.
- The attachment object(s) are not destroyed with the MailMessage object, but still exist on their own.

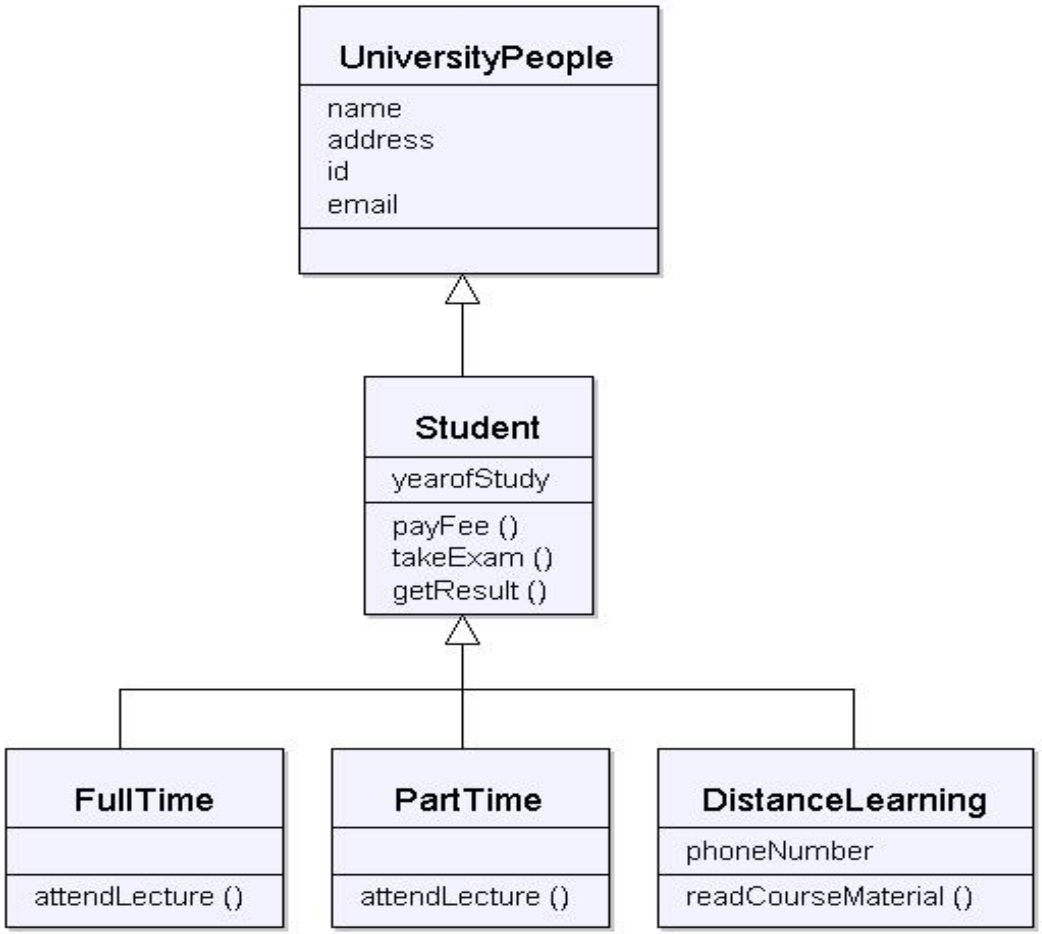
Generalization Example-1

- Generalization is a relationship between a general (super class) and a specific class (sub class).
- The specific class called the subclass **inherits** from the general class, called the superclass.
- Public and protected properties (attributes) and behaviors (operations) are inherited.
- It represents “is a” relationship among classes and objects.
- Represented by a line with an hollow arrow head pointing to the superclass at the superclass end.



Generalization Example-2

- Consider the following classes: UniversityPeople, Student, FullTime, PartTime and Distance Learning student. Draw a UML class diagram. Add properties and operations to the classes.



References

- Booch, G.: Object-Oriented Analysis and Design with Applications, Addison-Wesley, 1993, 2nd Edition.
- Blaha, M. and Rumbaugh, J.: Object-Oriented Modelling and Design with UML. Pearson Prentice-Hall, 2005. ISBN: 0-13-196859-9. (chapter 3,4)