CMPS411

Lecture 5

Object Oriented Analysis and Design

Domain Model and Design Class Diagram

Topics:

- ***Developing Domain model***
- ***Constructing Design Class Diagram***
- ***Class relationships***

Object Thinking

- 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, asynchronous 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)
- Object-oriented thinking allows us to model our system using abstractions from the problem domain

Why Object Thinking

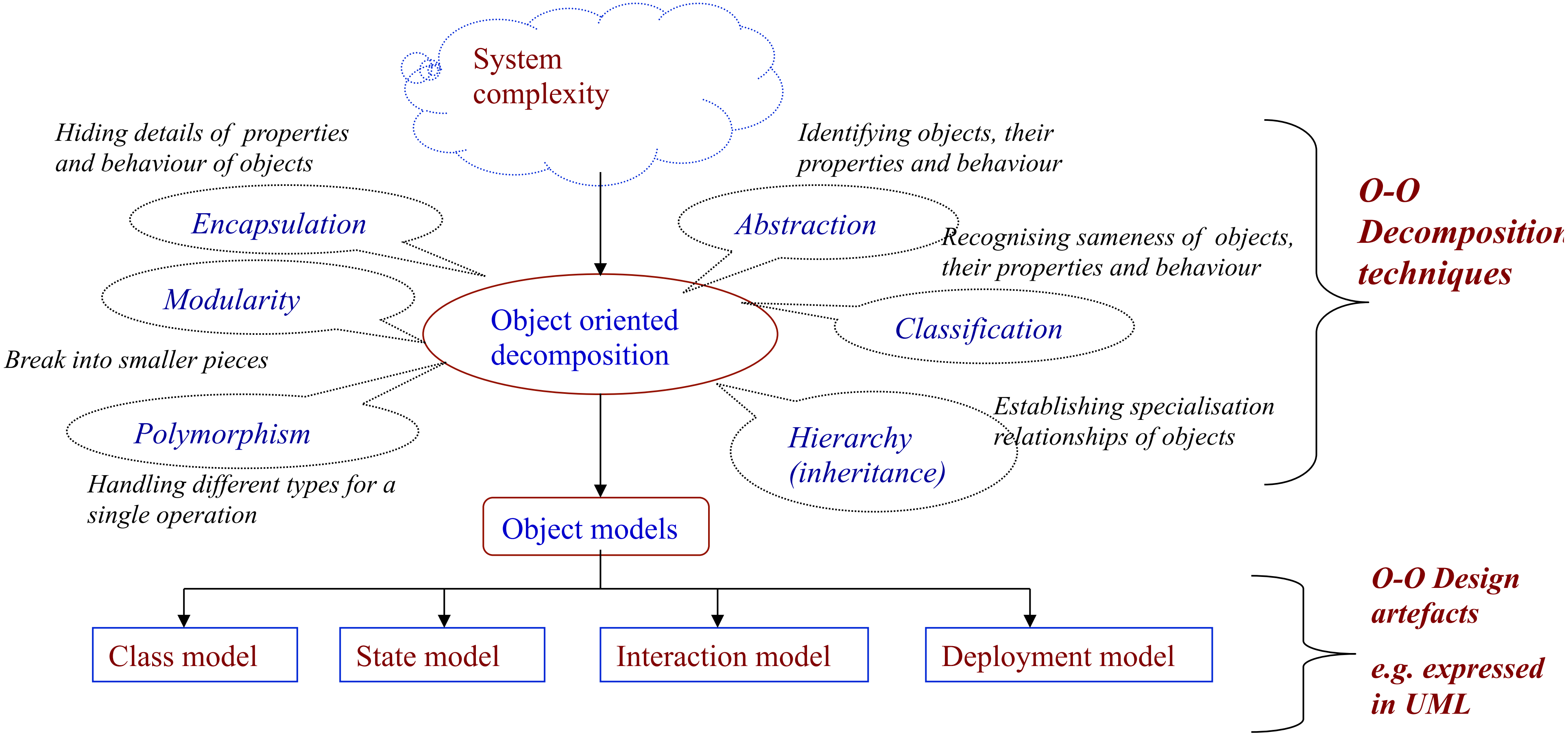
- To **identify classes** from the problem domain –the core of the system
  - *Techniques:*
    - Abstraction
    - Classification
    - Modularity

- To **recognise properties and behaviour** of the objects which are essential for the system by ignoring the inessential properties
  - *Techniques:*
    - Encapsulation
    - Abstraction

- To **establish relationships** among the identified objects
  - *Techniques:*
    - Classification
    - Hierarchy
    - Modularity.

O-O Decomposition and Artifact

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


System complexity
Hiding details of properties and behaviour of objects
Encapsulation
Modularity
Break into smaller pieces
Polymorphism
Handling different types for a single operation
Identifying objects, their properties and behaviour
Abstraction
Recognising sameness of objects, their properties and behaviour
Classification
Hierarchy (inheritance)
Establishing specialisation relationships of objects
Object oriented decomposition
Object models
Class model
State model
Interaction model
Deployment model
O-O Decomposition techniques
O-O Design artefacts
e.g. expressed in UML

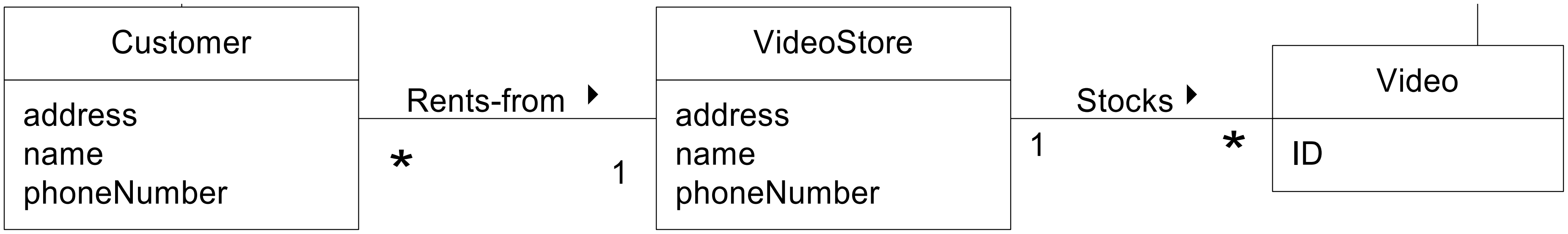
Domain Model and UML

- A *Domain Model* visualizes, using **UML class diagram** notation, noteworthy concepts 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
    - Does NOT show object's operations
- It helps us in understanding the domain: its concepts, terminology, and the relationships
- It provides inspiration for later creation of software design classes, to reduce "representational gap."
- Complexity of most software projects is understanding the domain objects.

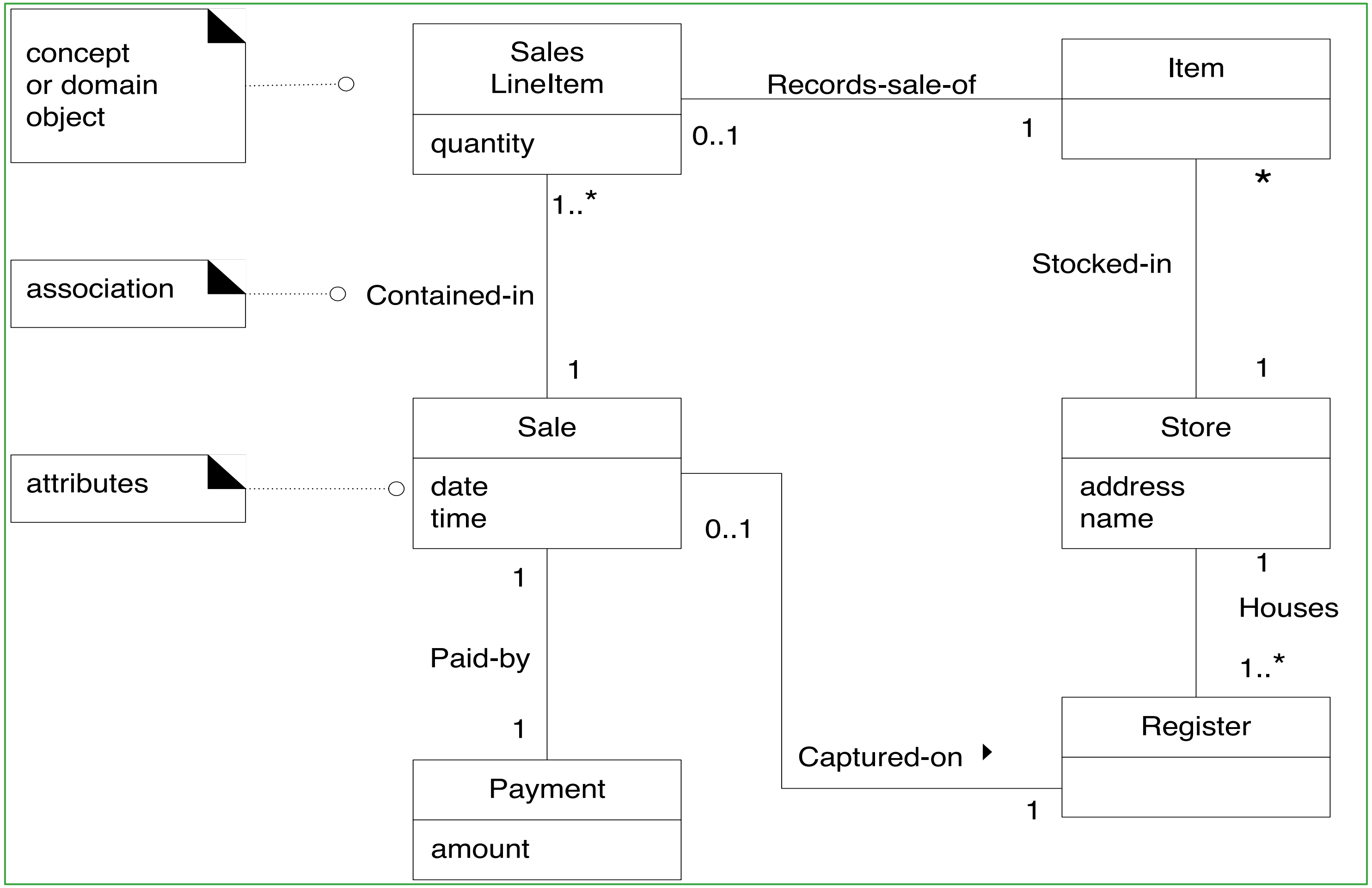
Domain Model Elements

- 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 Video Store Domain Model



Example: Partial Point-Of-Sale Domain Model

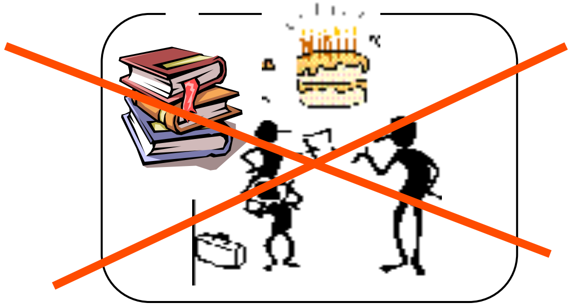


Identifying and Organizing Objects

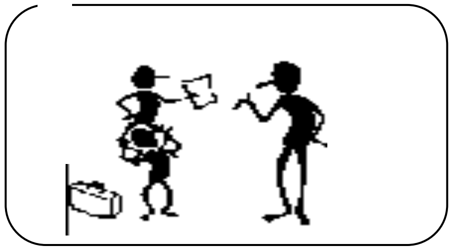
- **Abstraction** is often used to recognize the objects and their properties
  - Abstraction is a fundamental human capability that permits us to deal with complexity
  - Abstraction allows us to manage complexity by creating a simplified representation of something
  - Concentrating on the essential characteristics of objects/classes
  - The essential characteristics of an object that distinguish it from all other kinds of objects
  - This also assists to identify properties (attributes) and behaviors of the classes relevant to the application system
- **Classification** is used to group identified objects that have common properties or exhibit a common behavior (sameness/similarities of objects)
  - Classification means that objects with the same data structure and behavior are grouped into a class
  - Classification must have a criteria –reason for the classification
  - Classify objects in terms of presence or absence of a particular related property
    - Color of a car may be important for the inventory control system used by a car dealer
    - But, the color is not at all relevant to the traffic control software system
- **Modularity** helps us to reduce complexity, we need to break a program into smaller pieces

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**

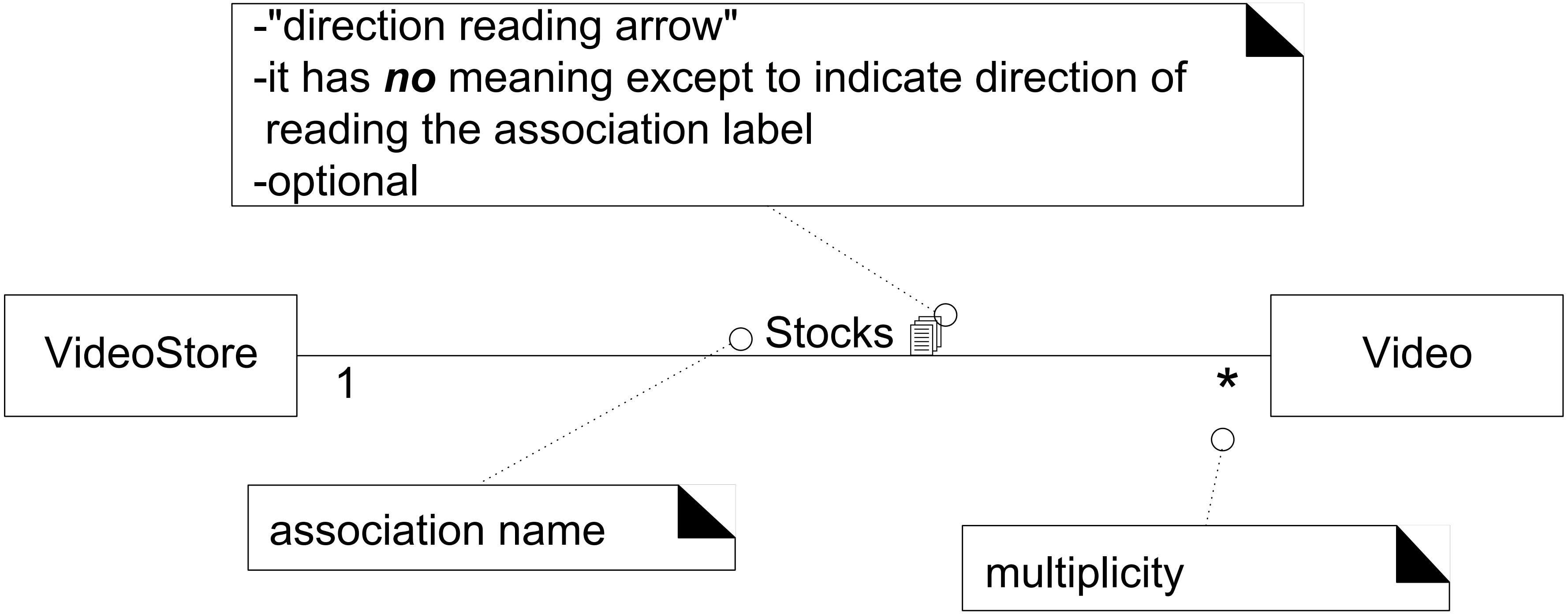
Attributes

- A logical data value of an object that needs to be remembered
    - Some attributes are derived from other attributes
- Show only “simple” relatively primitive types as attributes: number, text, date, time, Boolean…
- Connections to other concepts are to be represented as associations, not attributes.
- Objects vs. attributes
    - Attributes are ‘simple’ data types e.g. number, text
    - Concepts that are described by simple attributes are objects e.g., A Store has an address, phone number, etc.

  => Rule: If we do not think of a thing as a number or text in the real world, then it is probably a conceptual class.

Associations

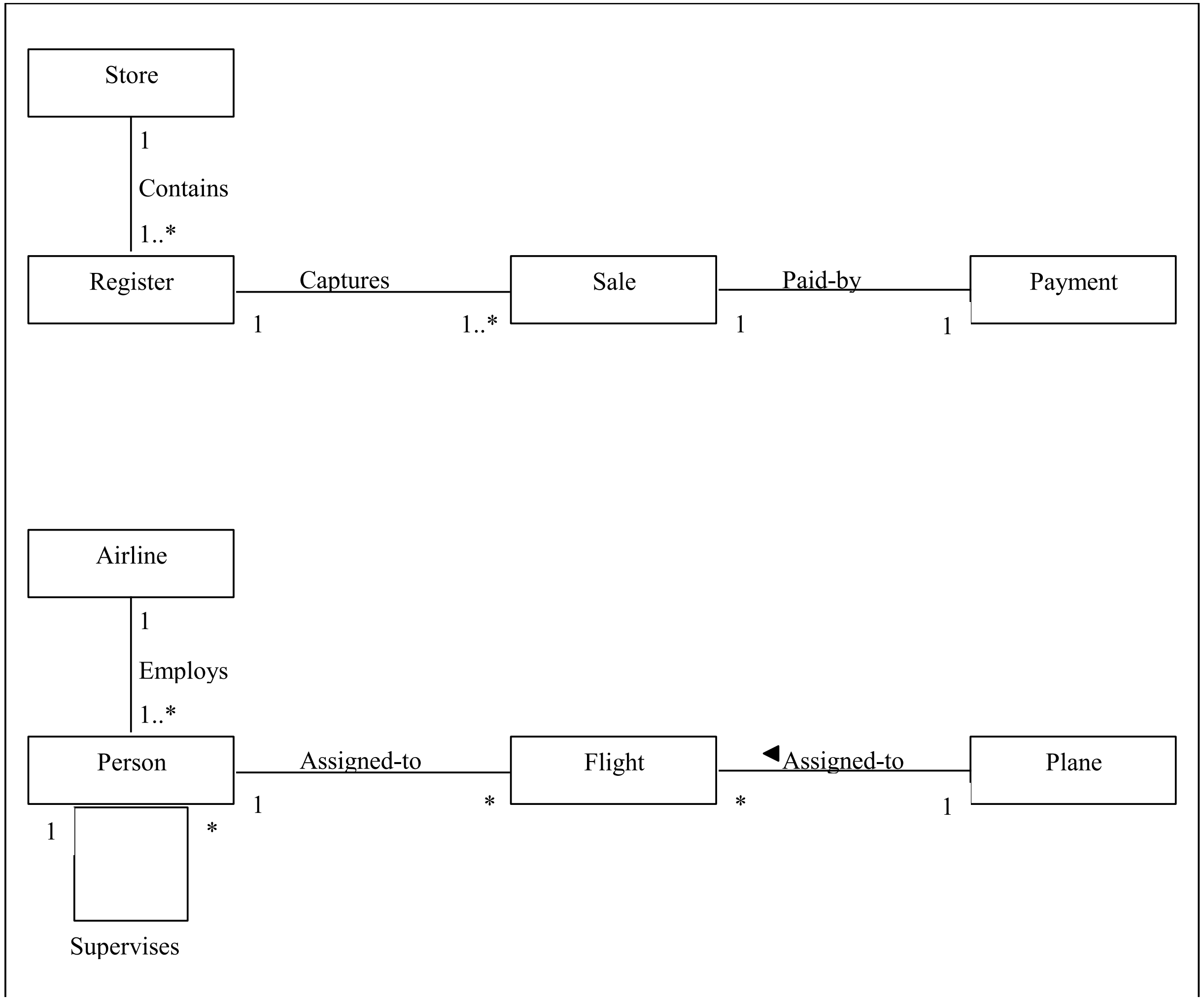
- Association = A meaningful relationship between objects
  - for which knowledge of the relationship needs to be preserved for some duration (i.e., “need-to-know” associations)



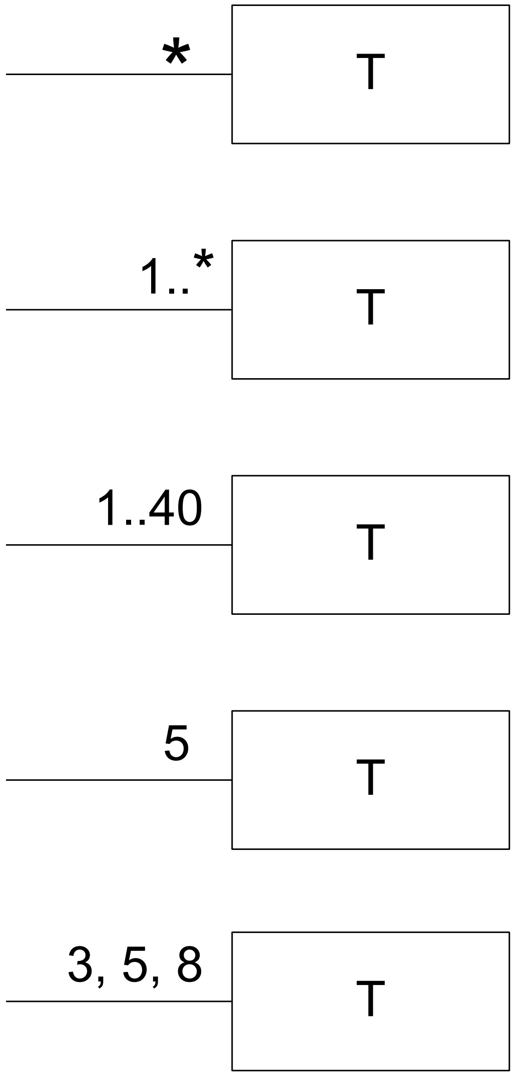
-"direction reading arrow"
-it has ***no*** meaning except to indicate direction of reading the association label
-optional
VideoStore
Stocks
Video
1
*
association name
multiplicity

Associations Names

- Name an association using a **VerbPhrase**
- Names should start with a capital letter.
- Legal formats are: Paid-by or PaidBy



Multiplicity



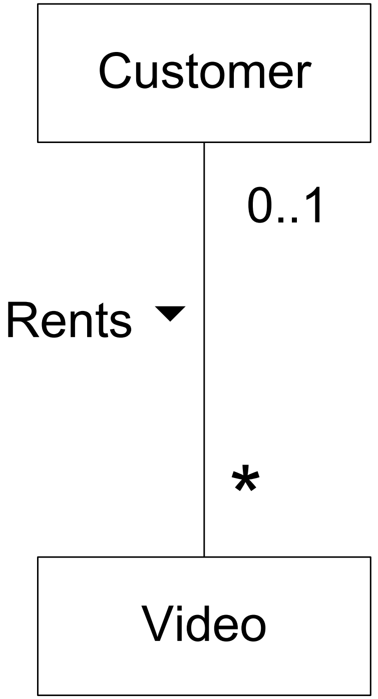
zero or more;
"many"

one or more

one to forty

exactly five

exactly three,
five or eight

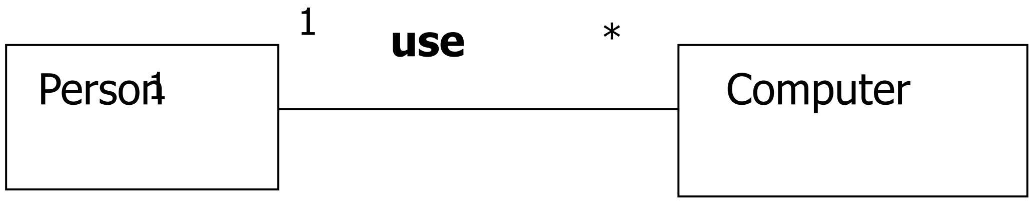


One instance of a Customer may be renting zero or more Videos.

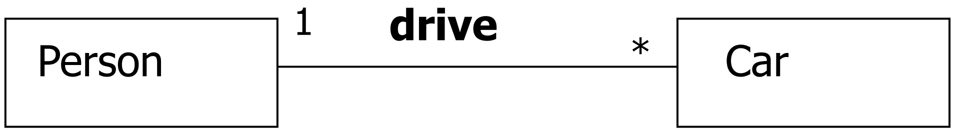
One instance of a Video may be being rented by zero or one Customers.

Normally, the multiplicity at a particular moment in time

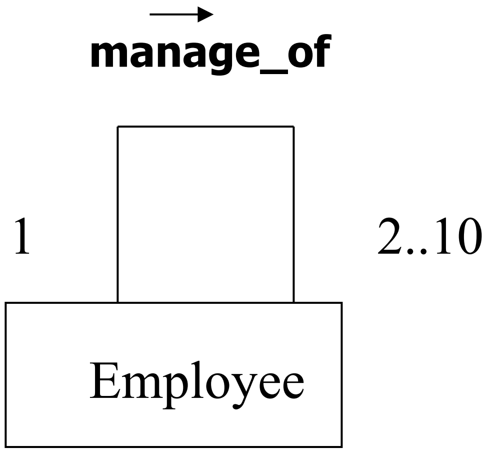
Recursive Association



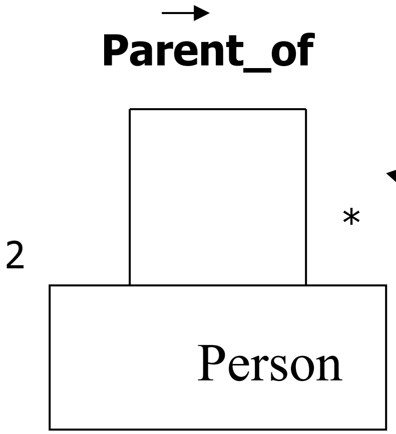
A person uses **a** computer.



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

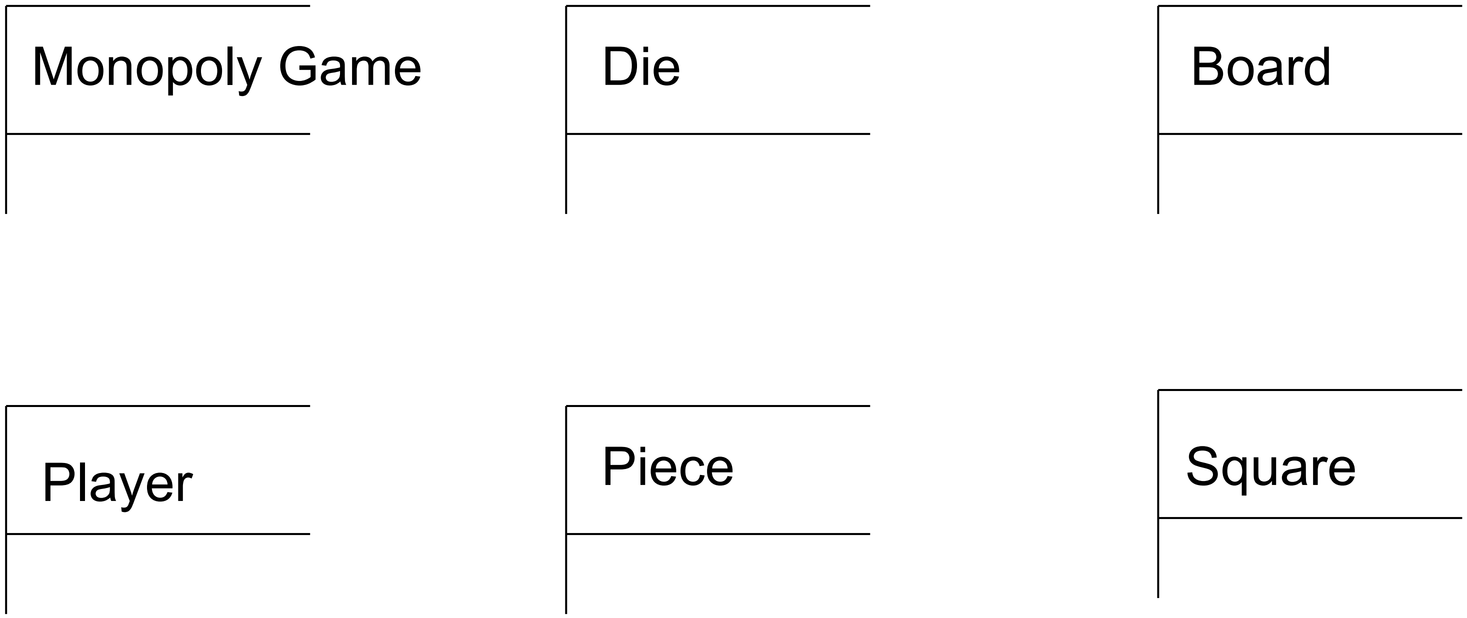


Recursive call: an object call other objects in the same class, but the role of the object is different

Steps to Create a Domain Model

- Identify candidate conceptual classes
  - Find generic noun phrases
- Draw them in a UML domain model (class diagram)
- Add associations necessary to record
  the relationships that must be retained
- Add attributes necessary for information to be preserved
- Use existing names for things, the vocabulary of the domain

Monopoly Game Domain Model
(first identify concepts as classes)



Monopoly Game

Die

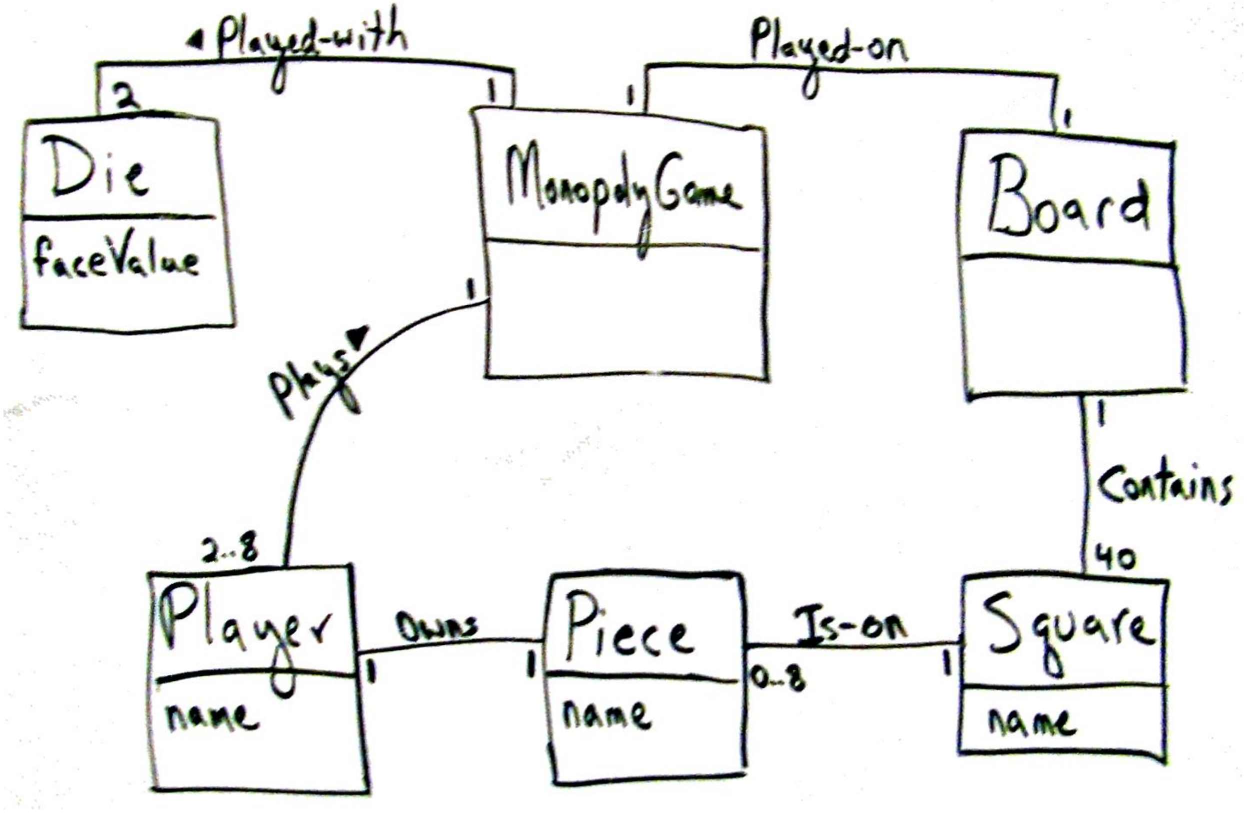
Board

Player

Piece

Square

Monopoly Game Domain Model



Exercise 1 - Noun Analysis

- A recruitment agency has decided to computerize their recruitment activities. The recruitment activities are only part of the business operation, not the entire business.
- **Use Case description:**
  - “When a new **client** contacts the **agency**, an **office-manager** assigns that **person** to a **consultant**, who will initially **interview** the **applicant** and fill out an **application form**. The **applicant** will then be matched to open **positions** at **companies**.”
- Given the use case description, make a list of the nouns used
- Analyze the list and determine what the candidate classes could be

Exercise 1 - Solution

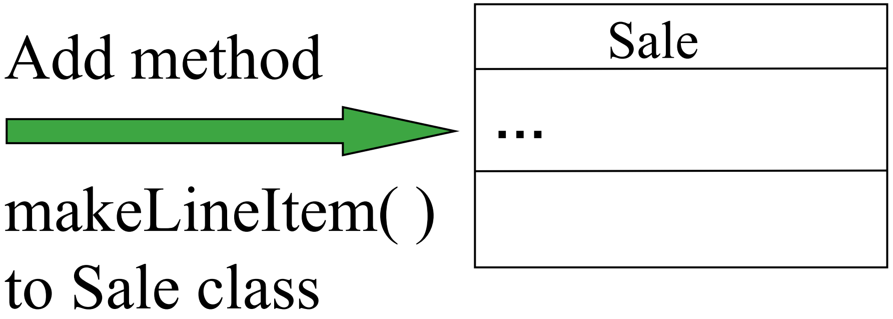
- Nouns
  - Client
  - Agency
  - Office-Manager
  - Person
  - Consultant
  - Applicant
  - Application Form
  - Company
  - Position
- Classes:
  - Applicant (client, person and applicant are the same)
  - Office-manager
  - Consultant
  - Application form
  - Company
  - Position
- Agency refers to the whole organization, its outside the system to be modelled.

Finding Associations

- “Fix” one concept class and consider the other ones in turn
- Are these two concepts related?
- If so, decide on the name of the association, the multiplicity…
  - Don’t just draw a line and leave the association unspecified
- Note that any missing associations will be discovered during the design

Making Design Class Diagram from Domain Model

- *We add the methods of the identified classes*

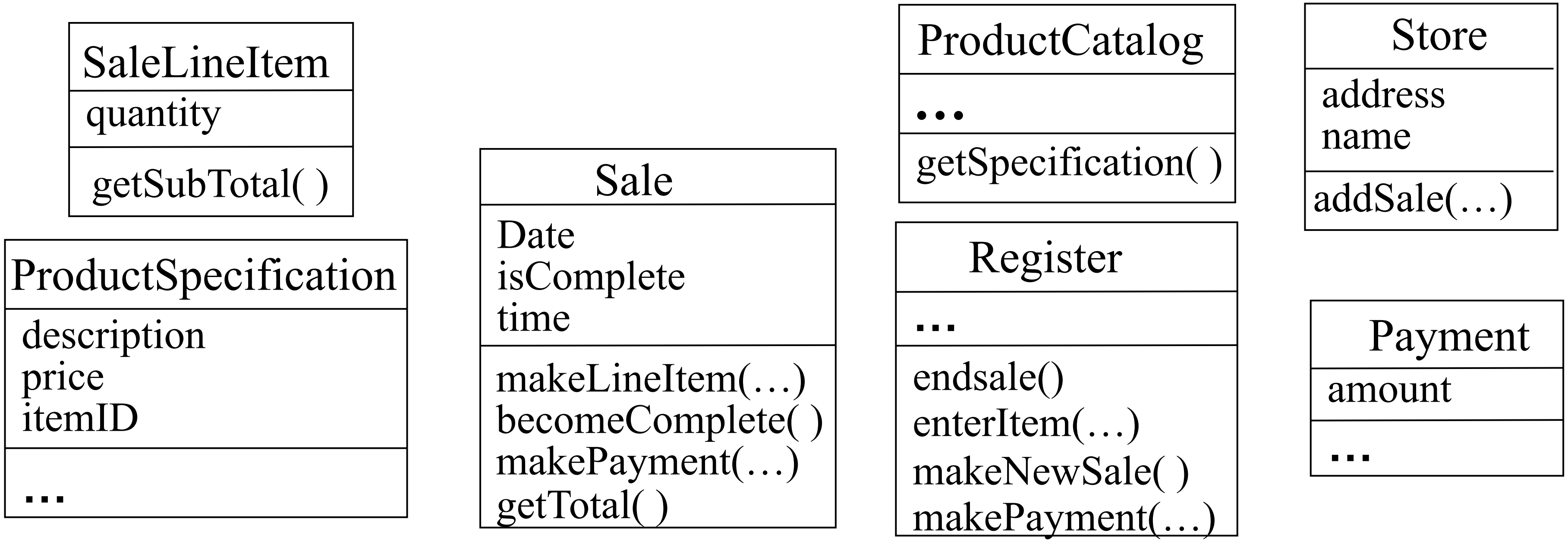


Add method

makeLineItem( )
to Sale class

Sale

…



SaleLineItem

quantity

getSubTotal( )

ProductSpecification

description
price
itemID

…

Sale

Date
isComplete
time

makeLineItem(…)
becomeComplete( )
makePayment(…)
getTotal( )

ProductCatalog

…

getSpecification( )

Register

…

endsale()
enterItem(…)
makeNewSale( )
makePayment(…)

Store

address
name

addSale(…)

Payment

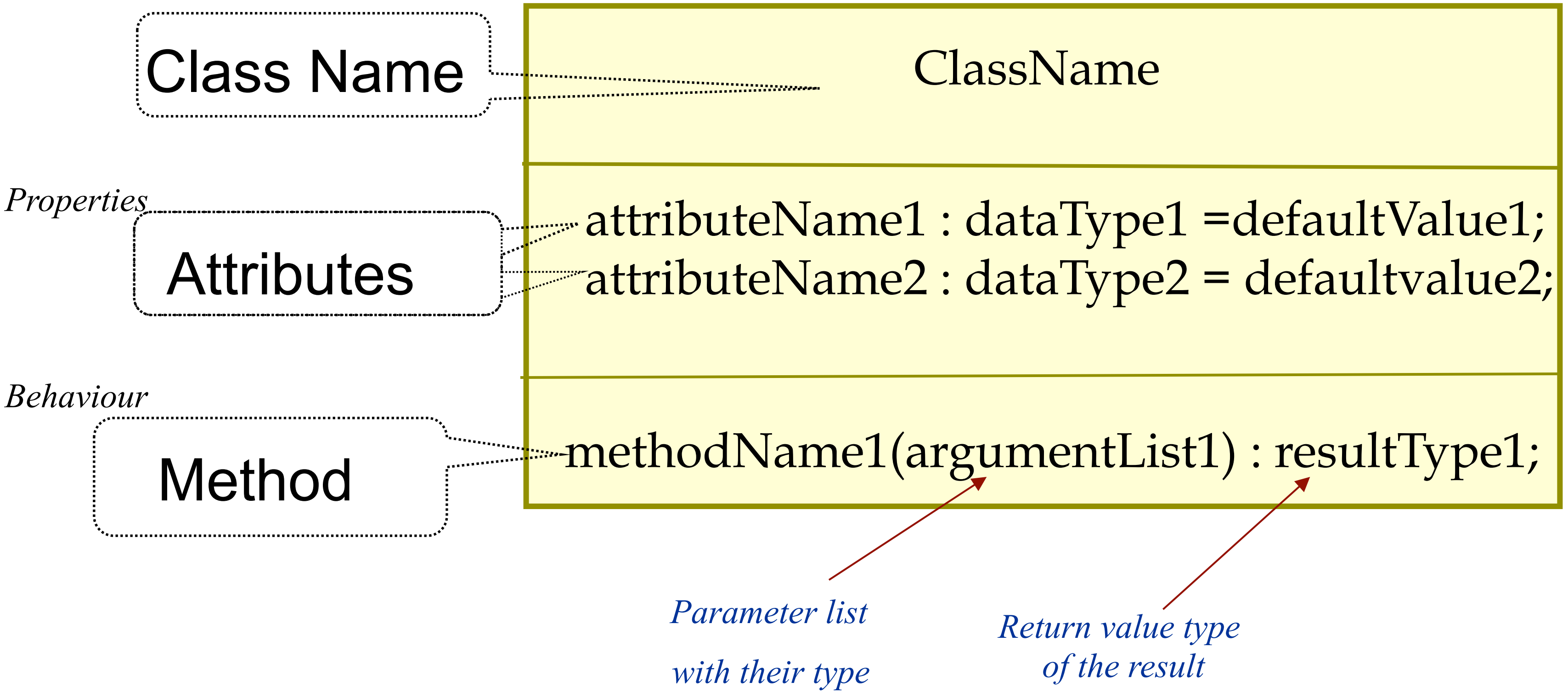
amount

…

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

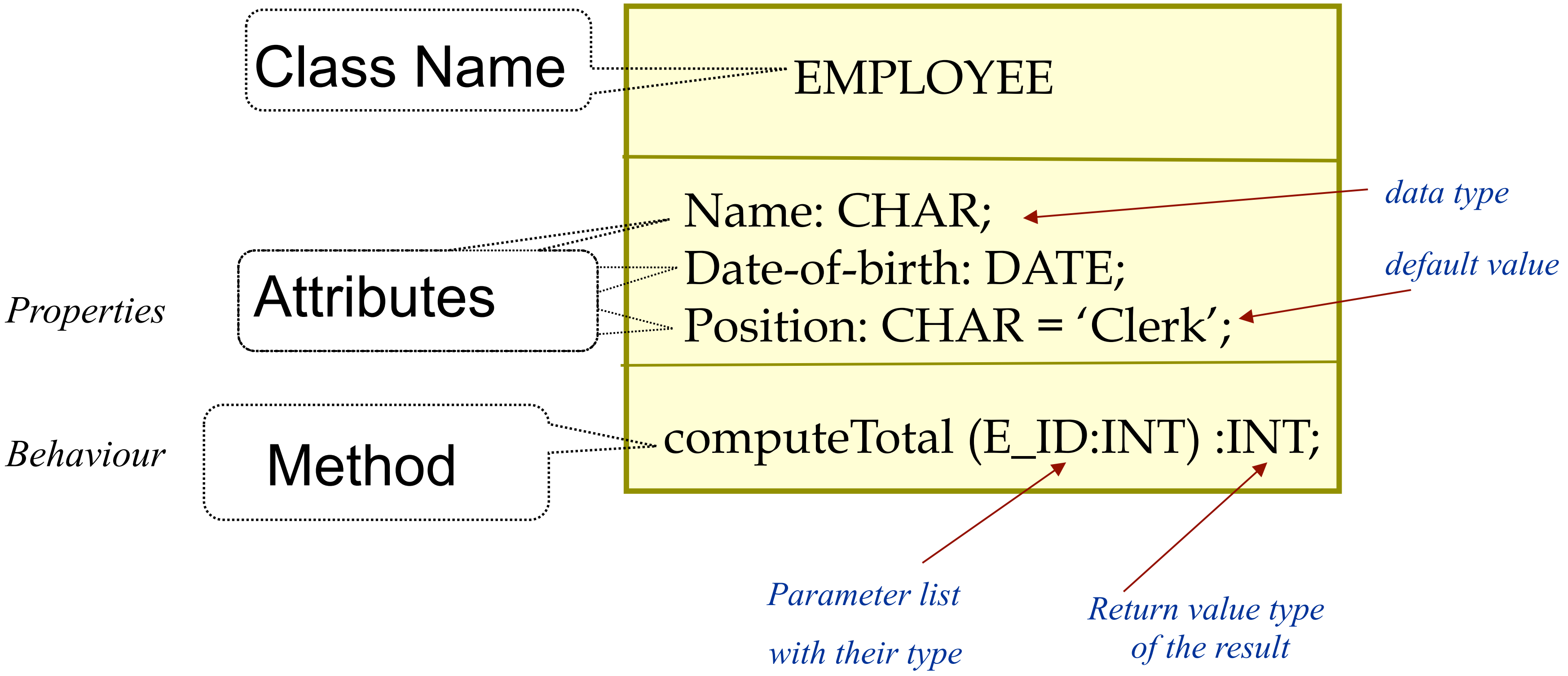


Class Name
ClassName
Properties
Attributes
attributeName1 : dataType1 =defaultValue1;
attributeName2 : dataType2 = defaultvalue2;
Behaviour
Method
methodName1(argumentList1) : resultType1;
Parameter list
with their type
Return value type
of the result

- An attribute should describe values, not objects

- Unlike objects, values lack identity. Types of values should be specified e.g., string. date, integer etc.

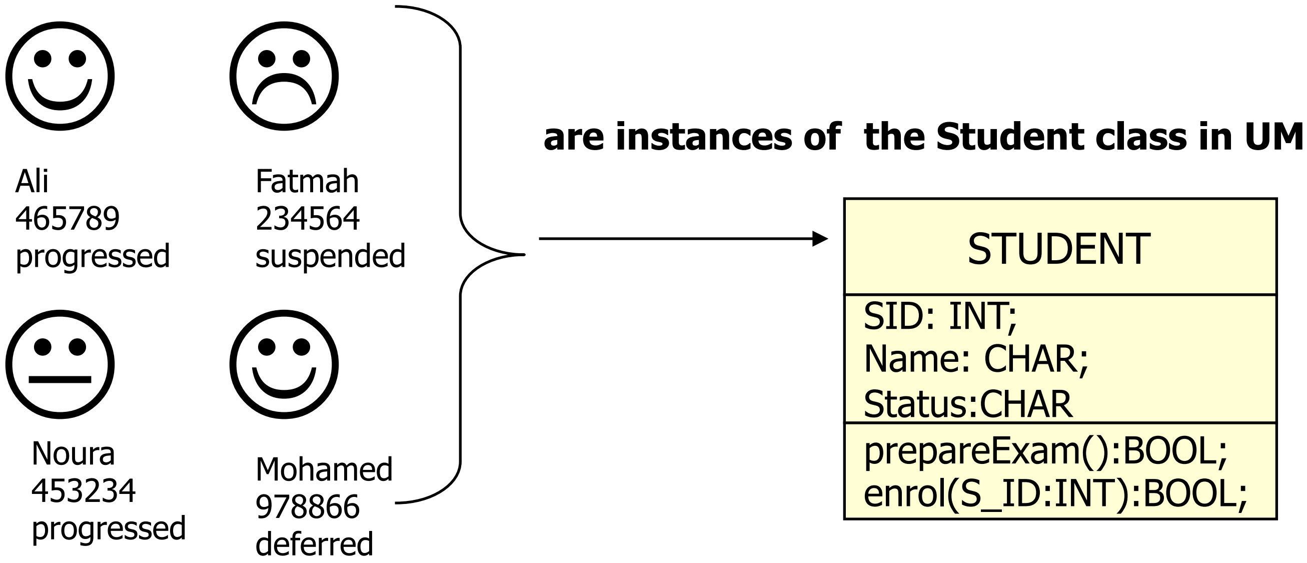
Example: Class in UML



Class and objects

- Objects are thus instantiated (created/defined) from classes
- Object is uniquely identifiable and has state
- Class is a structure of group of similar objects where an object cannot be identifiable

student objects



Ali
465789
progressed

Fatmah
234564
suspended

Noura
453234
progressed

Mohamed
978866
deferred

are instances of the Student class in UML

STUDENT

SID: INT;
Name: CHAR;
Status:CHAR

prepareExam():BOOL;
enrol(S_ID:INT):BOOL;

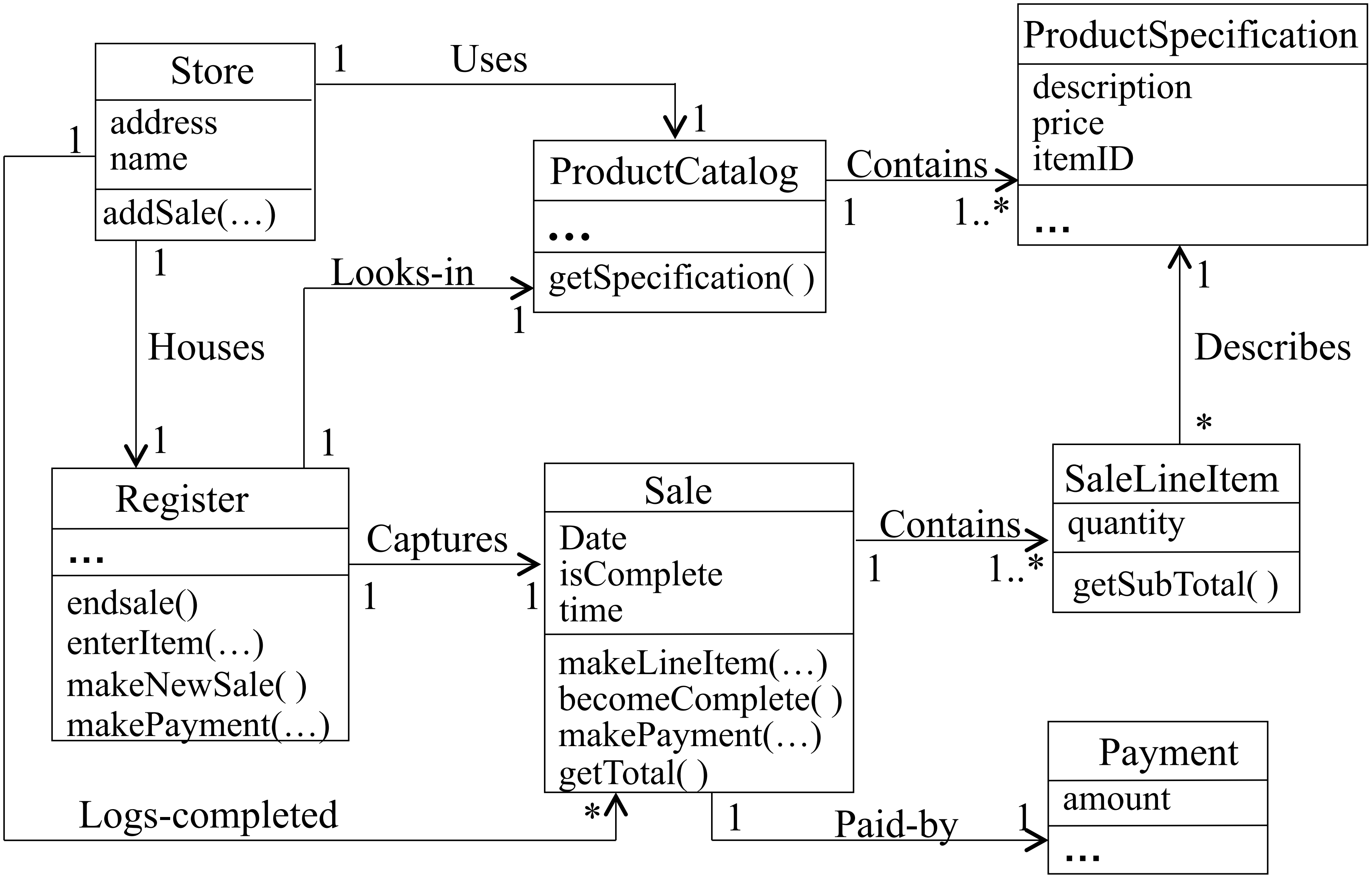
Object has specific information

Class has no specific information, only structure of similar objects.

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



Store
address
name
addSale(…)

ProductCatalog
…
getSpecification( )

ProductSpecification
description
price
itemID
…

Register
…
endsale()
enterItem(…)
makeNewSale( )
makePayment(…)

Sale
Date
isComplete
time
makeLineItem(…)
becomeComplete( )
makePayment(…)
getTotal( )

SaleLineItem
quantity
getSubTotal( )

Payment
amount
…

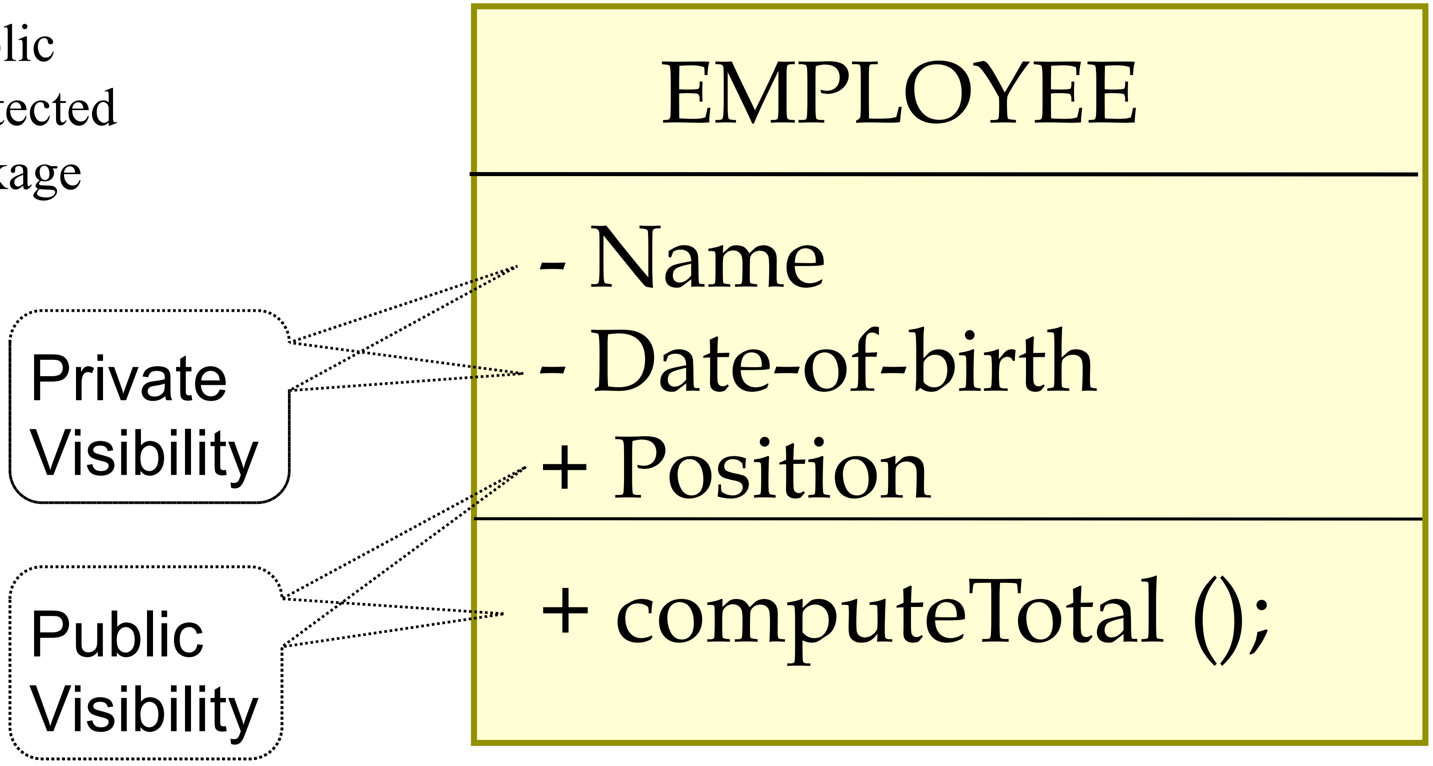
Visibility of Properties

- 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



EMPLOYEE

- Name

- Date-of-birth

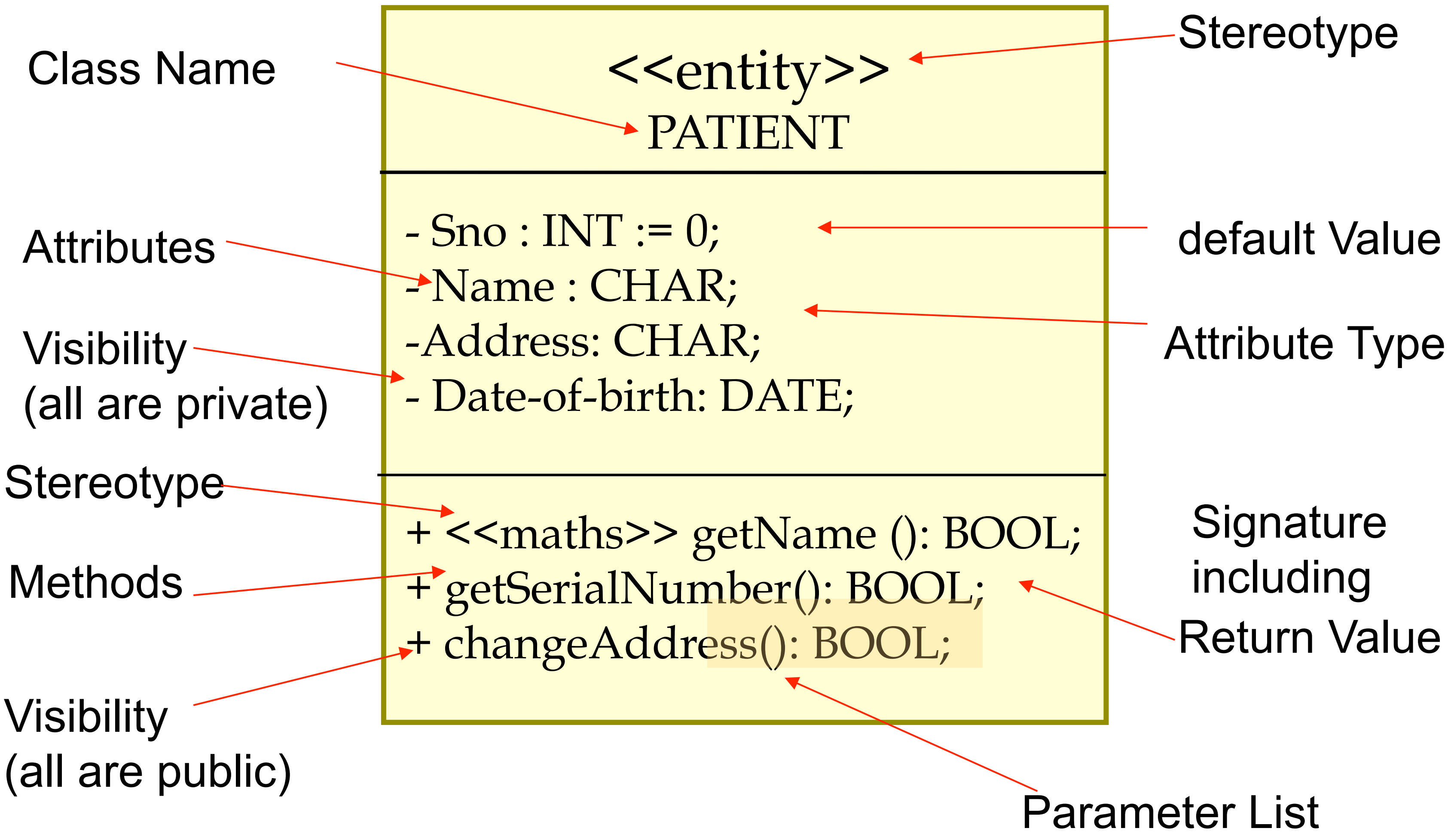
+ Position

+ computeTotal ();

Private Visibility

Public Visibility

Detailed Class Definition in UML



<<entity>>
PATIENT

- Sno : INT := 0;
- Name : CHAR;
-Address: CHAR;
- Date-of-birth: DATE;

+ <<maths>> getName (): BOOL;
+ getSerialNumber(): BOOL;
+ changeAddress(): BOOL;

Class Name

Stereotype

Attributes

default Value

Visibility
(all are private)

Attribute Type

Stereotype

Methods

Signature
including
Return Value

Visibility
(all are public)

Parameter List

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

Airport

Account

Flight number

Ticket

Airlines

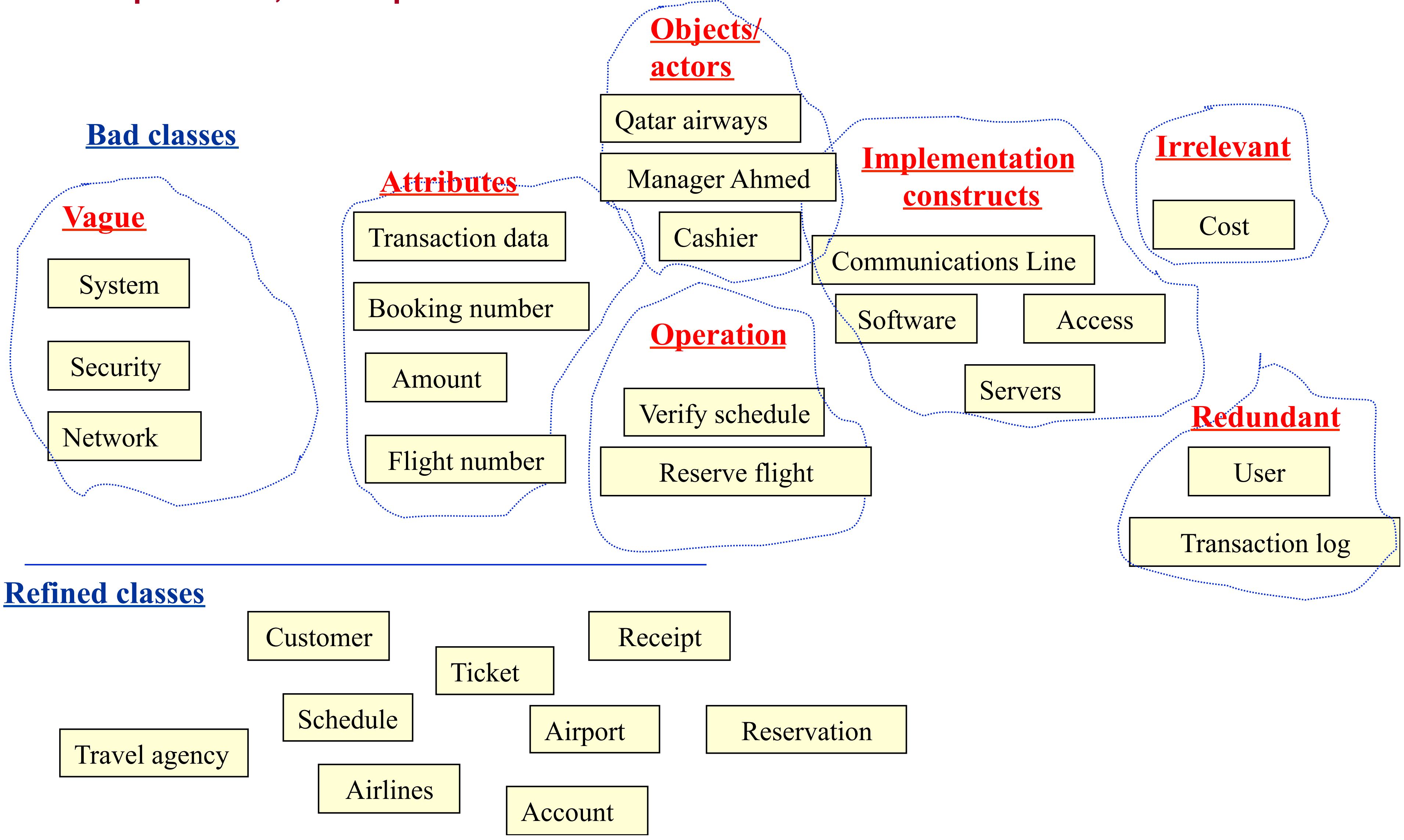
Manager Ahmed

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

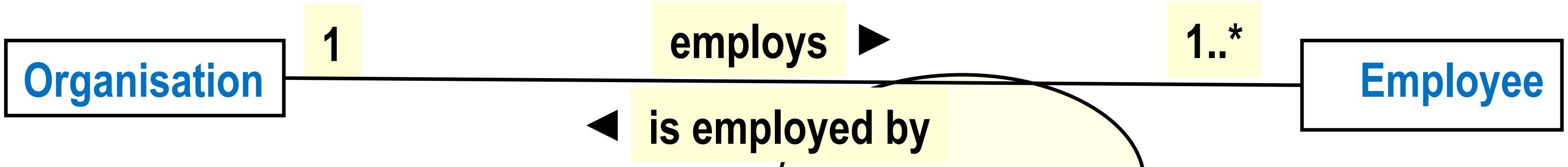
- An actor in a use case diagram can only be defined as a class in the class diagram if the information of the actor is manipulated/used within the system
- An actor **cannot** be an object of the system if it is not manipulated/ saved/used within the system.
  - In that case, the actor is just a user, 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



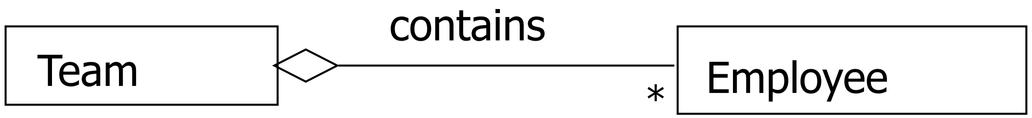
```
class Organisation
{
   List<Employee> employees;
   . . . . .
}

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**.



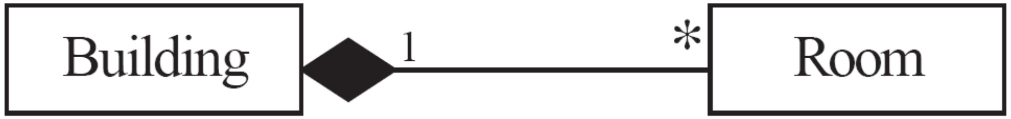
- Aggregation represents a relation “contains”, “is a part of”, “whole-part” relation.
  - Part instances can be added to and removed from the aggregate

Composition

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

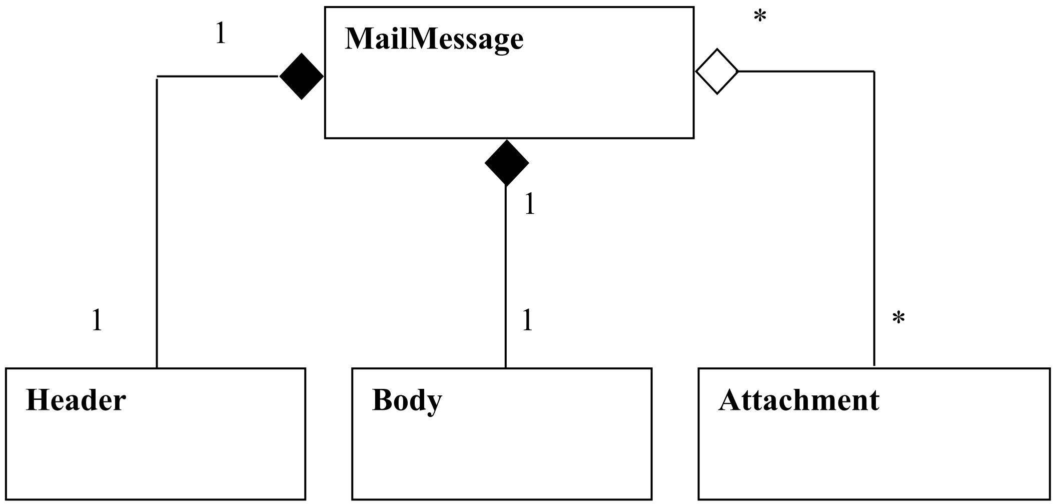
- Stronger than an aggregate
- Often involves a physical relationship between the whole and the parts, not just conceptual
- the part objects are created, live, and die together with the whole: **the life cycle of the ‘part’ is controlled by the ‘whole’.** Part cannot exist independent of the whole.

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



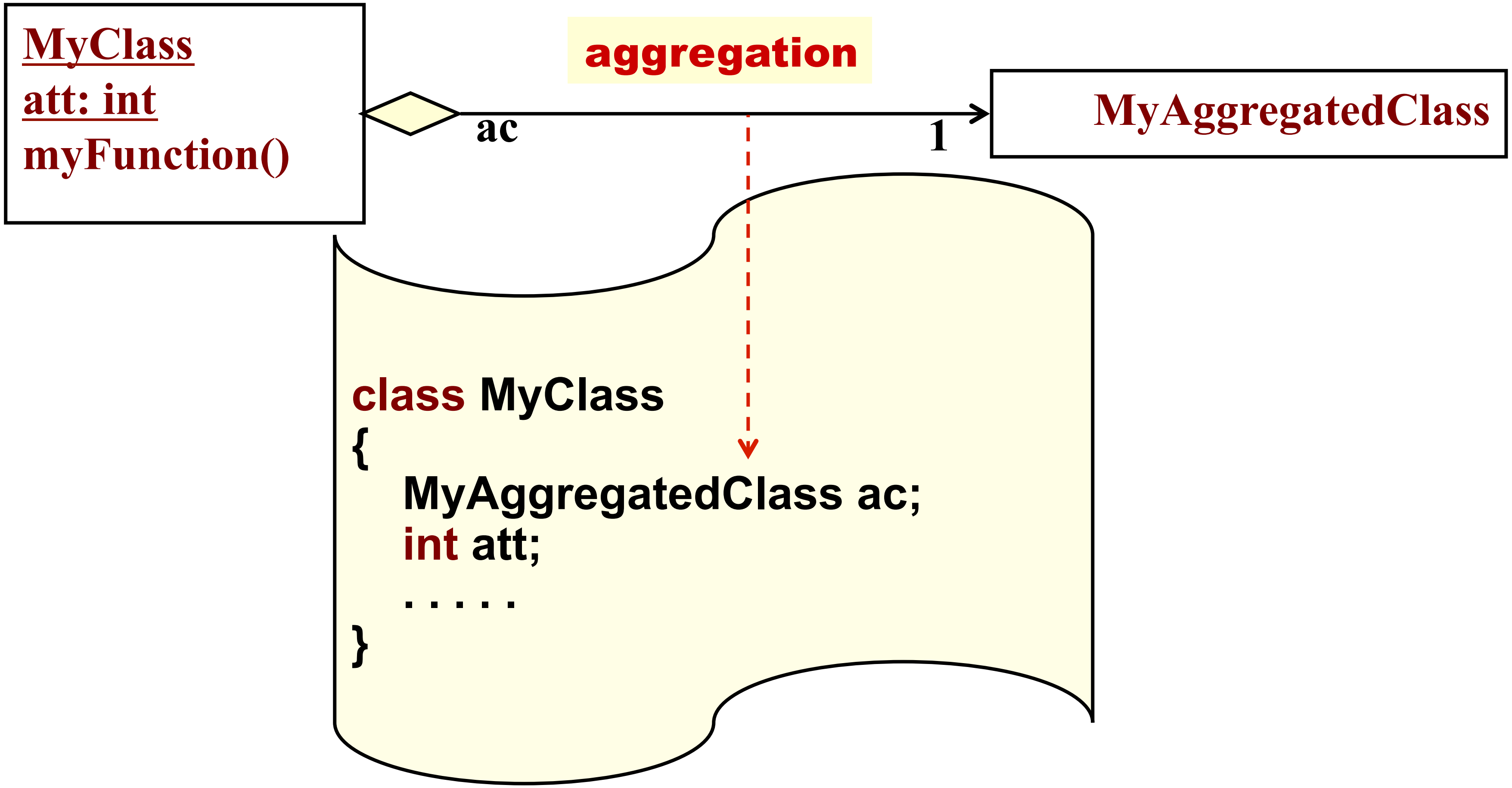
Aggregation vs. Composition Example 1

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



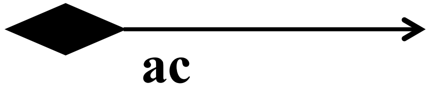
- When a MailMessage object is destroyed, so are the Header object and the Body object.
- The attachment object(s) are not destroyed with the MailMessage object, but still exist on their own.

Aggregation : UML Notation and Typical Implementation



```
class MyClass
{
   MyAggregatedClass ac;
   int att;
   . . . . .
}
```

composition

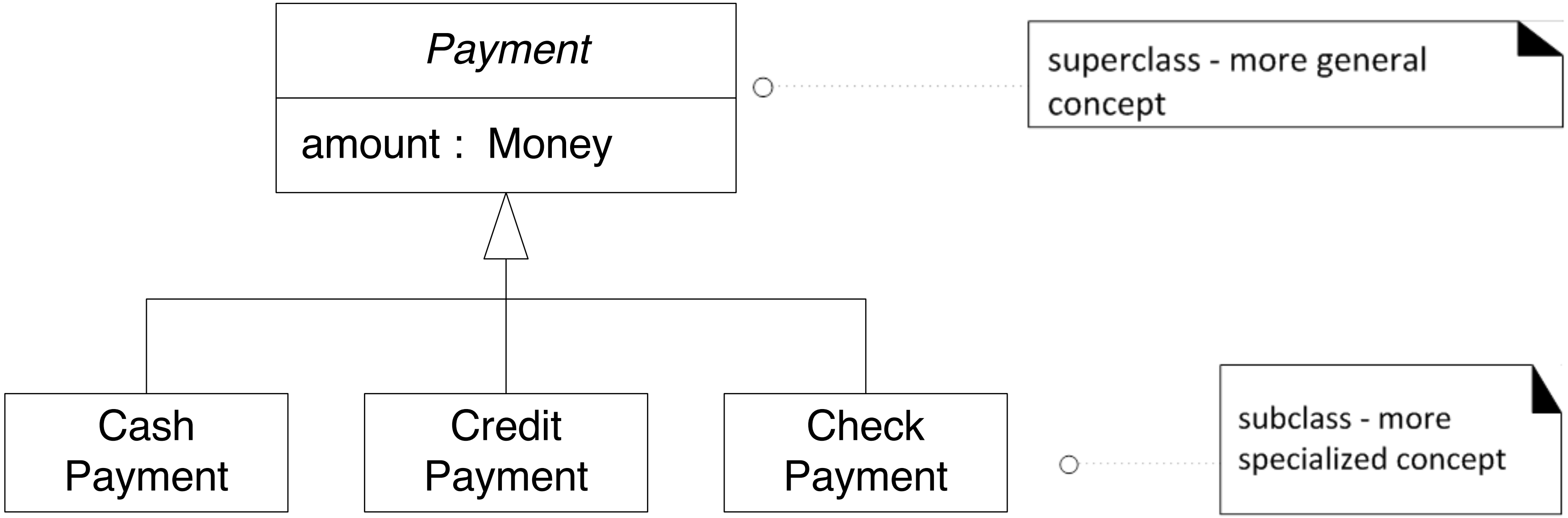


Composed object exists only in the scope of owner object

Generalization

- Generalization is a relationship between a general and a specific 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.

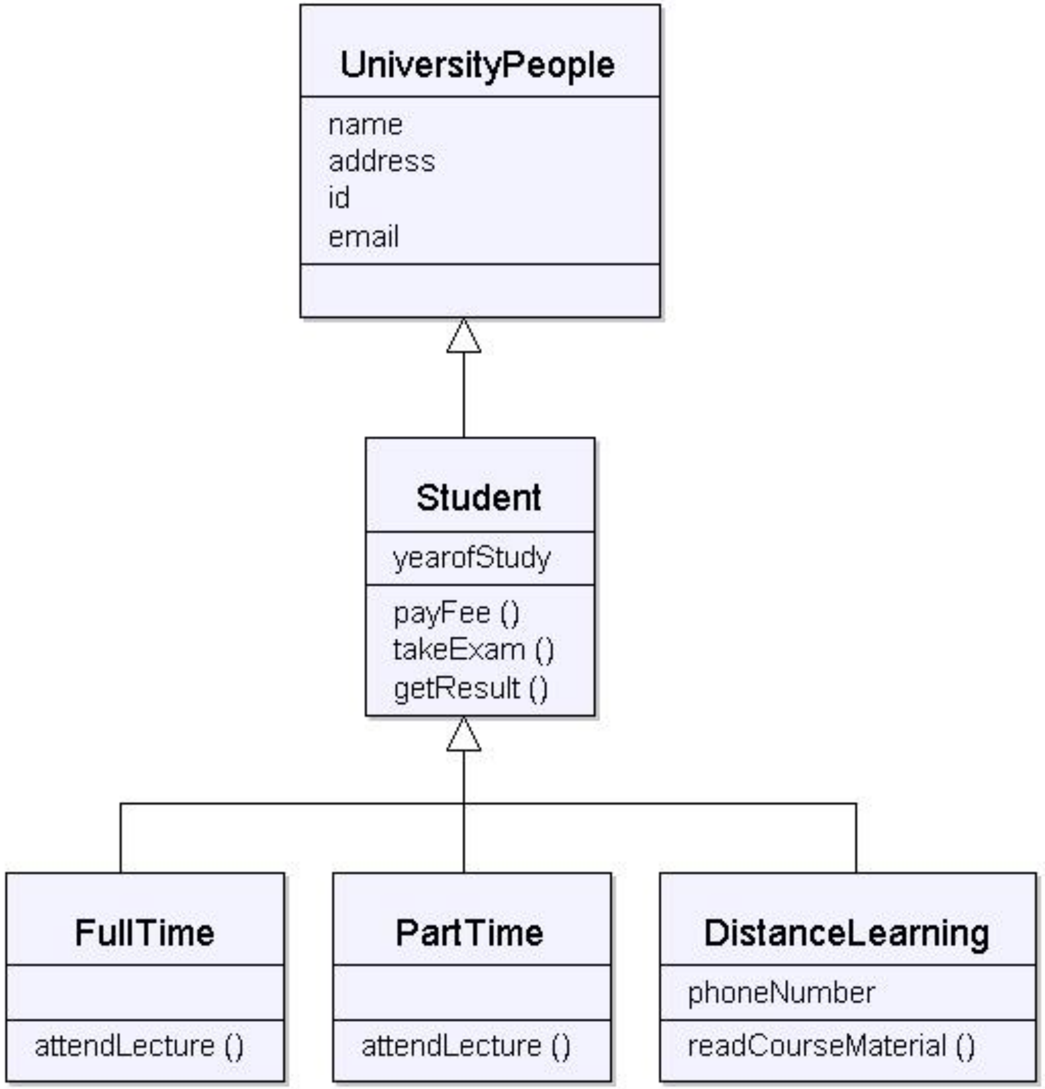
Generalization Example


Payment
amount : Money
Cash Payment
Credit Payment
Check Payment
superclass - more general concept
subclass - more specialized concept

Exercise 1 - Generalization

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

Exercise 1 - Solution



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)