UML

Class Diagram

What Is the UML?

- The UML is a language for
    - Visualizing
    - Specifying
    - Constructing
    - Documenting

  the artifacts of a software-intensive system.

- The Unified Modelling Language (UML) is an industry standard for object oriented design notation, supported by the Object Management Group (OMG).

UML class diagrams

- What is a UML class diagram?
  - **UML class diagram**: a picture of the classes in an OO system, their fields and methods, and connections between the classes that interact or inherit from each other
- What are some things that are not represented in a UML class diagram?
  - details of how the classes interact with each other
  - algorithmic details; how a particular behavior is implemented

In software development

UML is a:

- **design**: 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 will 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?

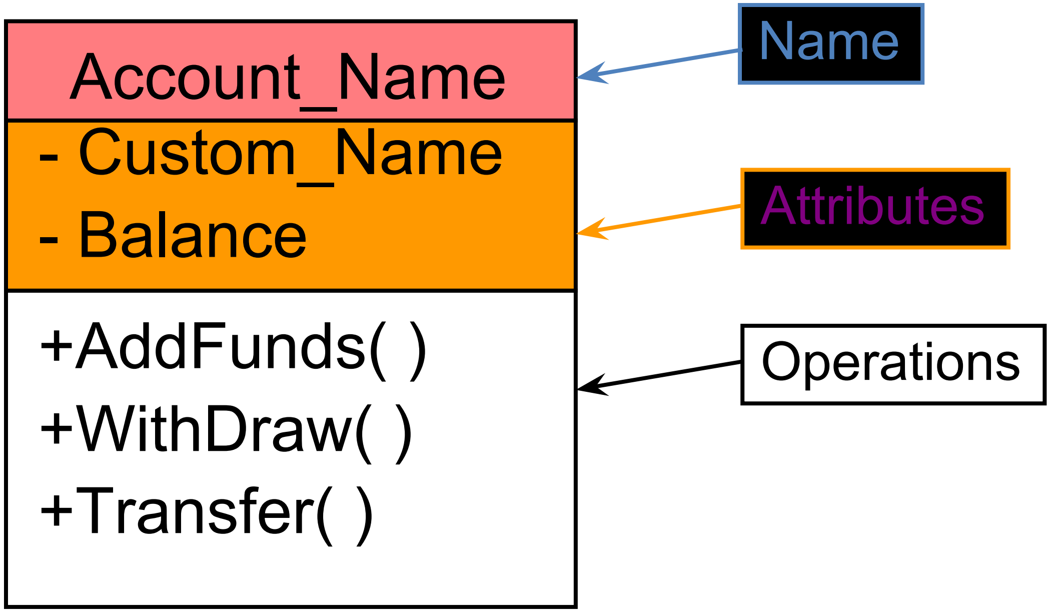
Class Diagrams

- Class diagram is basically a graphical representation of the static view of the system and represents different aspects of the application. So a collection of class diagrams represent the whole system.

- Each class is represented by a rectangle subdivided into three compartments
  - Name
  - Attributes
  - Operations
- Modifiers are used to indicate visibility of attributes and operations.
  - '+' is used to denote *Public* visibility (everyone)
  - '#' is used to denote *Protected* visibility (friends and derived)
  - '-' is used to denote *Private* visibility (no one)
- By default, attributes and operations are hidden.
- The last two compartments may be omitted to simplify the class diagrams

Cont..

- A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.



Account_Name

- Custom_Name

- Balance

+AddFunds( )

+WithDraw( )

+Transfer( )

Name

Attributes

Operations

Class Names

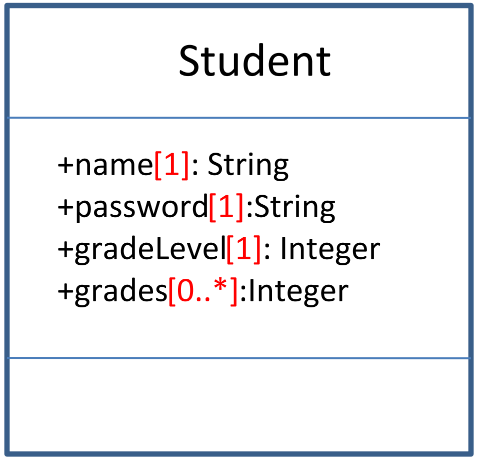
- The name should be a noun or noun phrase
- The name should be singular and description of each object in the class
- The name should be meaningful from a problem-domain perspective
  - “Student” is better than “Student Data” or “S-record” or any other implementation driven name

Attributes

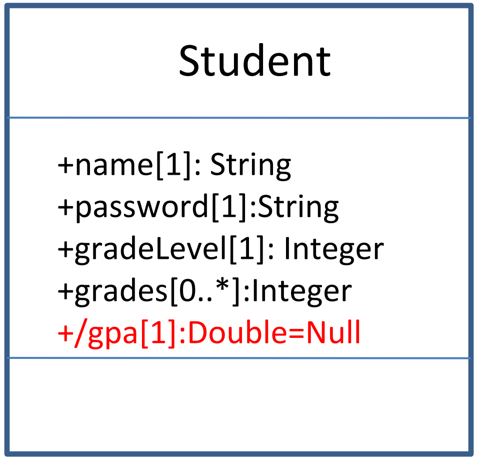
- Attributes represent characteristics or properties of classes
- They are place holders or slots that hold values
- The values they hold are other objects (or primitive types)
- Syntax :
  - **[*visibility*]** **name** [*multiplicity*] [:***type***] [=*initial-value*]

Cont..

- ✔*visibility*: public “+”, protected “#”, or private “-”
- ✔*name*: capitalize first letter of each word that makes up the name, except for the first
- ✔*multiplicity*: number, range, or sequence of number or ranges.
- ✔*type*: built-in type or any user-defined class
- ✔*initial-value*: any constant and user-defined object


Student

+name[1]: String
+password[1]:String
+gradeLevel[1]: Integer
+grades[0..*]:Integer


Student

+name[1]: String
+password[1]:String
+gradeLevel[1]: Integer
+grades[0..*]:Integer
+/gpa[1]:Double=Null

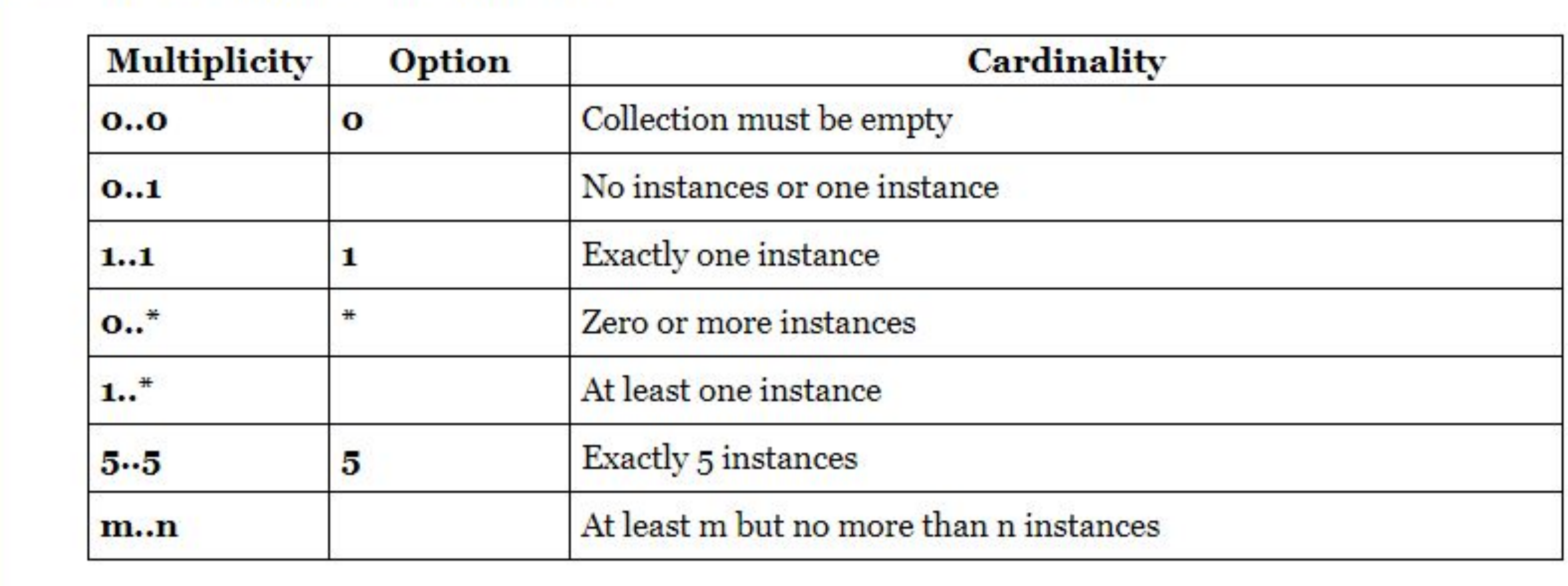
 A *derived* attribute is one that can be computed from other attributes, but doesn’t actually exist. For example, a Person’s age can be computed from his birth date. A derived attribute is designated by a preceding ‘/’ as in:

/ age : Date

Multiplicity

Multiplicity is a definition of **cardinality** - i.e. **number of elements**

Some typical example of multiplicity



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

Operation Syntax

- **Operation is a behavioural features of class.**
- Operation is invoked on an instance of the classes for which the operation is a feature
- [***visibility***] **name [(*parameter-list*)] [:*return-type*]**

  visibility: “+”, “#”, “-” optional by default private

  name: verb or verb phase, capitalize first letter of every word, except first

  parameter-list: coma separated list of parameters

  return-type: primitive type or user-defined type

Cont..

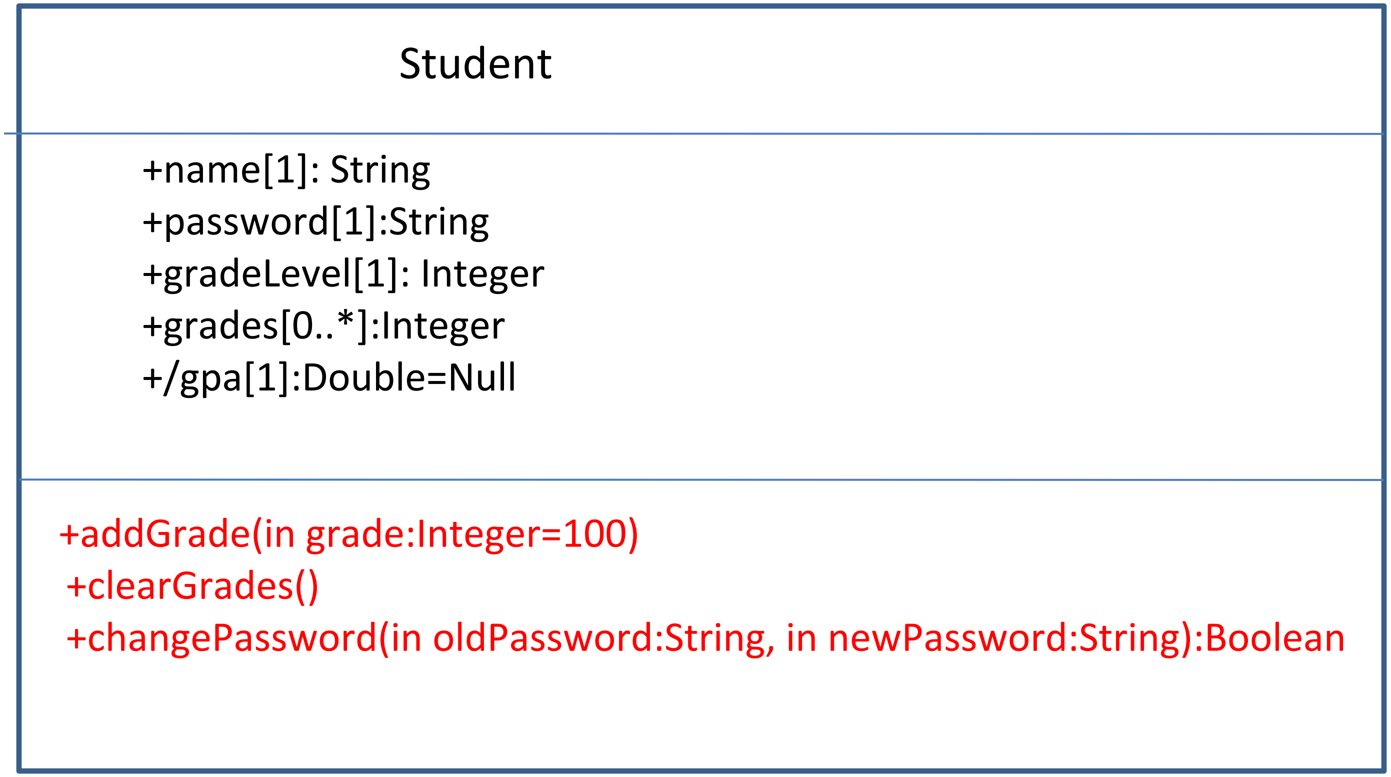
Direction of parameter is described as one of:

direction ::= 'in' | 'out' | 'inout' and defaults to 'in' if omitted.

in: passed by value

inout: passed by reference

out: value not passed into function rather the operation returns a value with in the parameter.

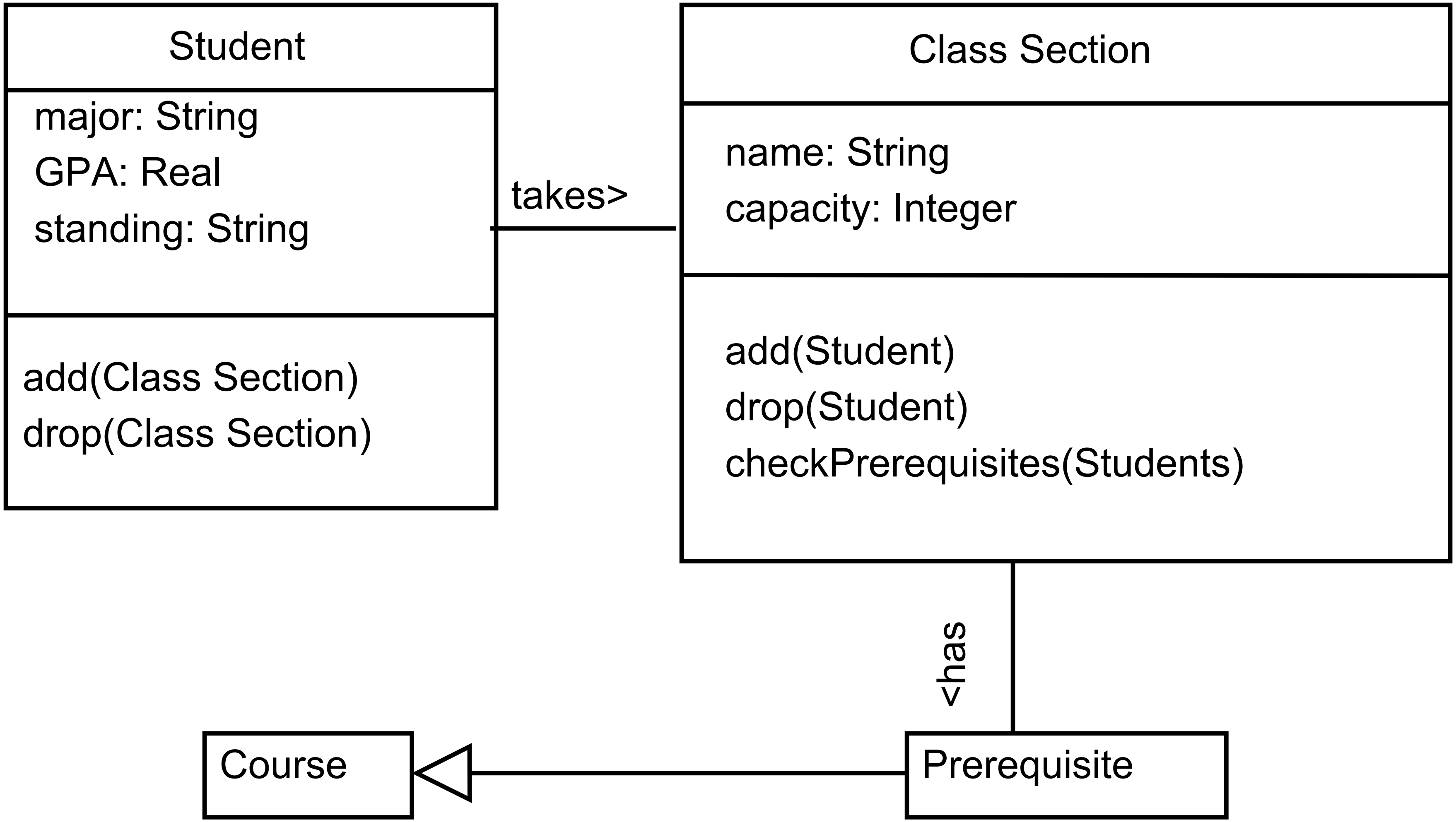


Student

+name[1]: String
+password[1]:String
+gradeLevel[1]: Integer
+grades[0..*]:Integer
+/gpa[1]:Double=Null

+addGrade(in grade:Integer=100)
+clearGrades()
+changePassword(in oldPassword:String, in newPassword:String):Boolean

Operations



Student

major: String
GPA: Real
standing: String

add(Class Section)
drop(Class Section)

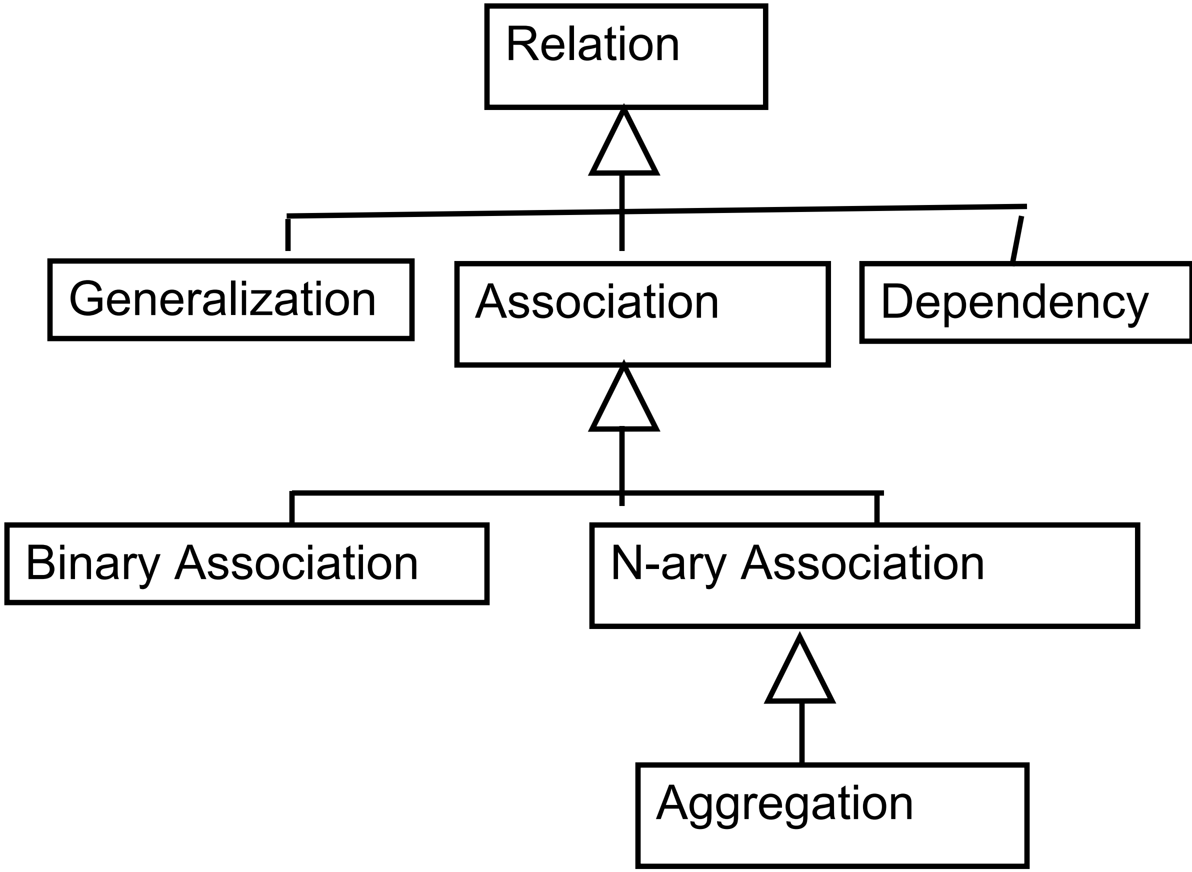
Class Section

name: String
capacity: Integer

add(Student)
drop(Student)
checkPrerequisites(Students)

Type of Relationships in Class Diagrams

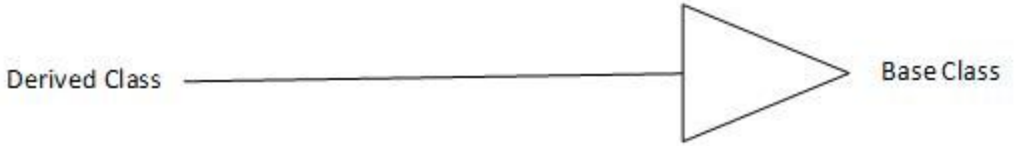
1. Generalization
2. Association
3. Composition
4. Aggregation
5. Realisation
6. Dependency



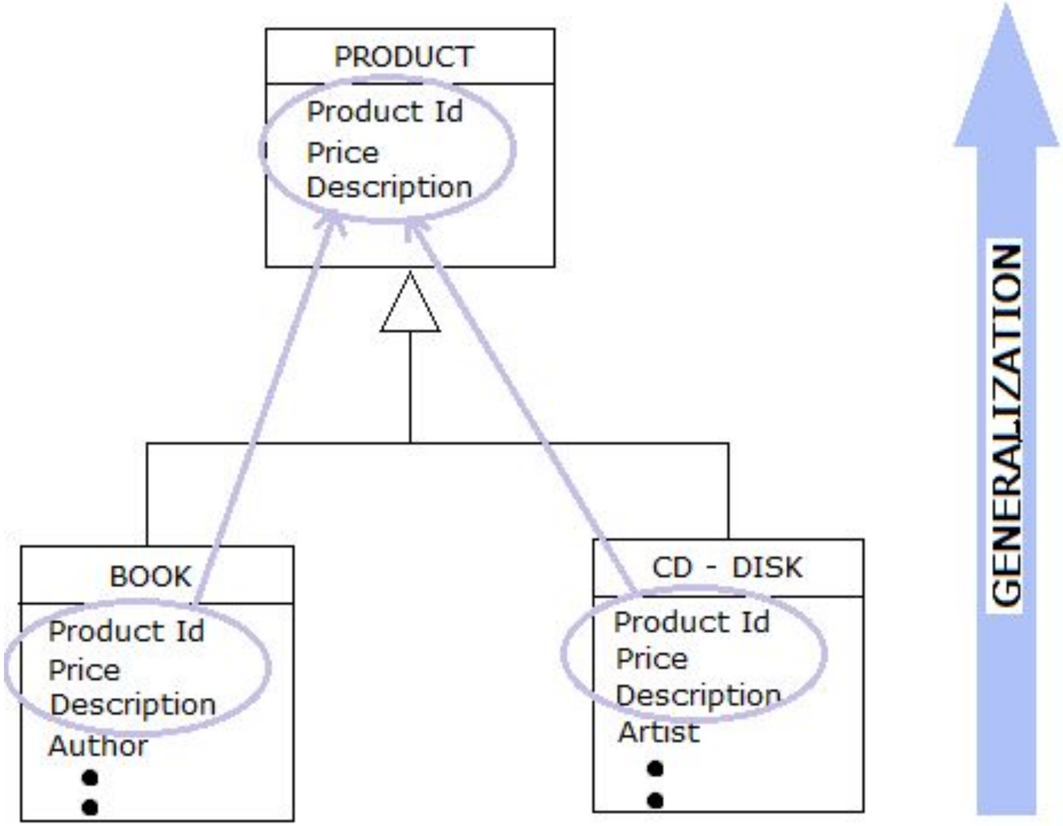
Generalization

- When we create a base/super class from two or more similar type of objects by extracting their all common characteristics(attributes and behavior) and combine them, then this process is known as Generalization.
- **generalization**: an inheritance relationship
  - inheritance between classes
  - interface implementation
- Indicates that objects of the specialized class (subclass) are substitutable for objects of the generalized class (super-class).

  "is –a-kind-of" relationship.

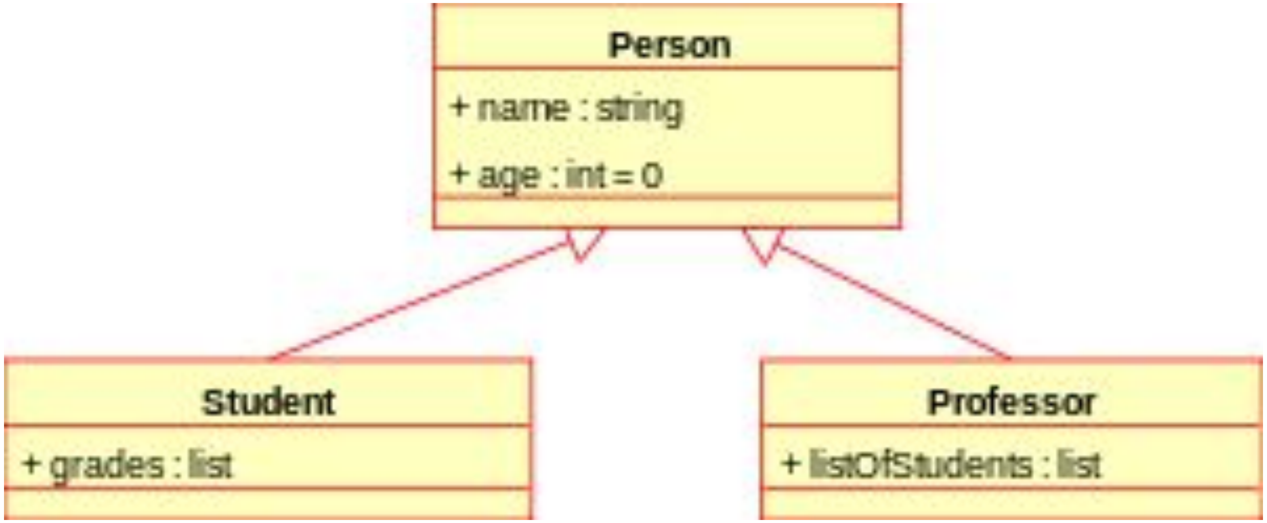


Cont..



Example : In the above figure the classes Book and Disk partially share the same attributes. During generalization, the shared characteristics such as Id, Price and description are combined and used to create a new super class Product. Book and Disk become sub classes of the class Product.

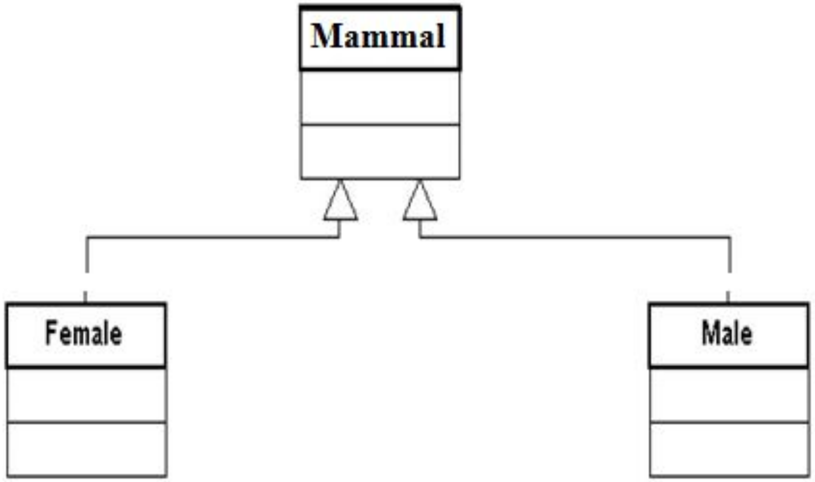
Cont..



Class diagram showing generalization between the super class *Person* and the two subclasses *Student* and *Professor*

Generalization

- A sub-class inherits from its super-class
  - Attributes
  - Operations
  - Relationships
- A sub-class may
  - Add attributes and operations
  - Add relationships
  - Refine (override) inherited operation



C++ Example for Generalization

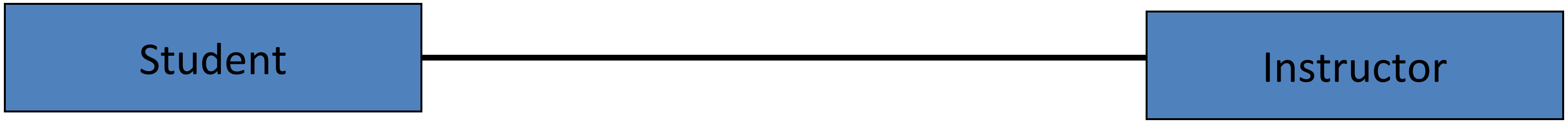
```
class 2DPoint {
    int x, y;
  };
  class 3DPpoint : 2DPoint {
     int z;
  };
```

Associations

- **Association** is a **relationship** between **classes** which is used to show that instances of classes could be either **linked** **to each other**
- A semantic relationship between two or more classes that specifies connections among their instances.
- A structural relationship, specifying that objects of one class are connected to objects of a second (possibly the same) class.
- Example: “An Employee works for a Company”
- **association**: a usage relationship
  - dependency
  - aggregation
  - composition

Association Relationships

- If two classes in a model need to communicate with each other, there must be link between them.
- An *association* denotes that link.



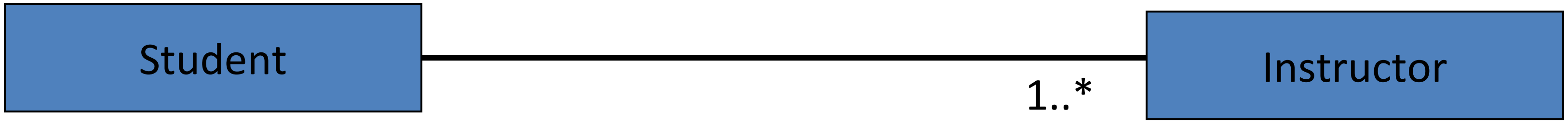
Association Relationships (Cont’d)

Multiplicity

- The number of instances of the class, next to which the multiplicity expression appears, that are referenced by a **single** instance of the class that is at the other end of the association path.

- Provides a lower and upper bound on the number of instances

- The example indicates that a *Student* has one or more *Instructors*:



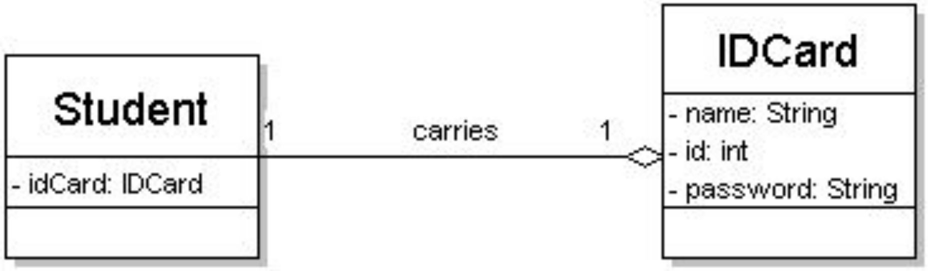
Student

1..*

Instructor

Multiplicity of associations

- one-to-one
  - each student must carry exactly one ID card



Student

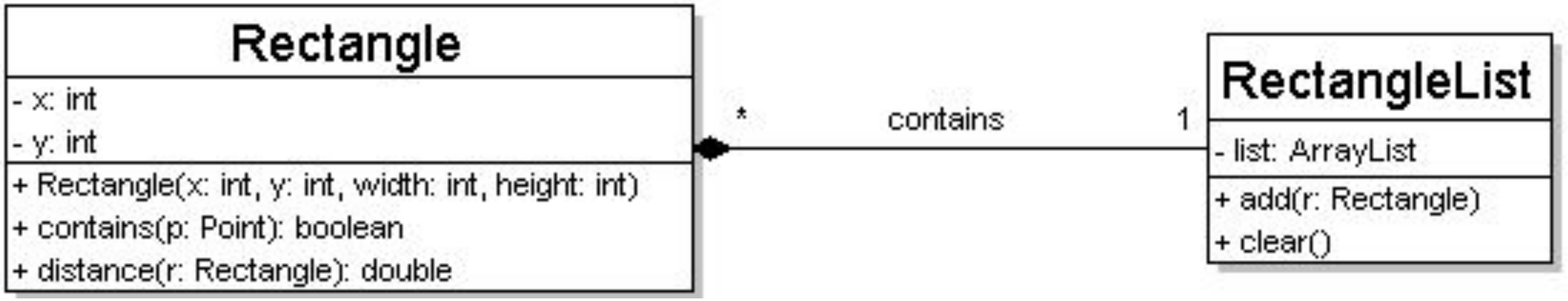
- idCard: IDCard

1 carries 1

IDCard

- name: String
- id: int
- password: String

- one-to-many
  - one rectangle list can contain many rectangles



Rectangle

- x: int
- y: int

+ Rectangle(x: int, y: int, width: int, height: int)
+ contains(p: Point): boolean
+ distance(r: Rectangle): double

* contains 1

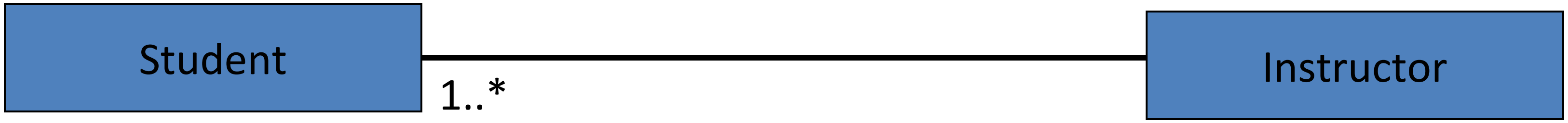
RectangleList

- list: ArrayList

+ add(r: Rectangle)
+ clear()

Association Relationships (Cont’d)

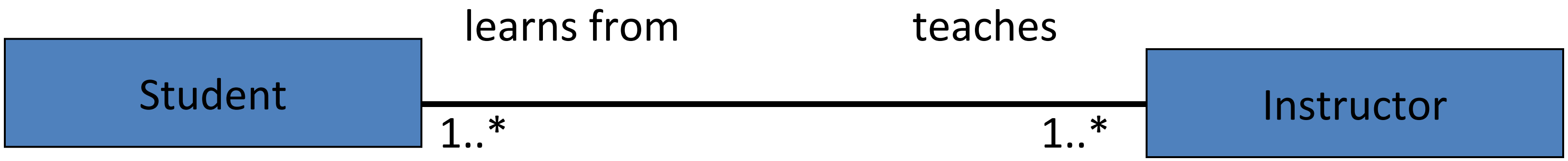
The example indicates that every *Instructor* has one or more *Students*:



Association Relationships (Cont'd)

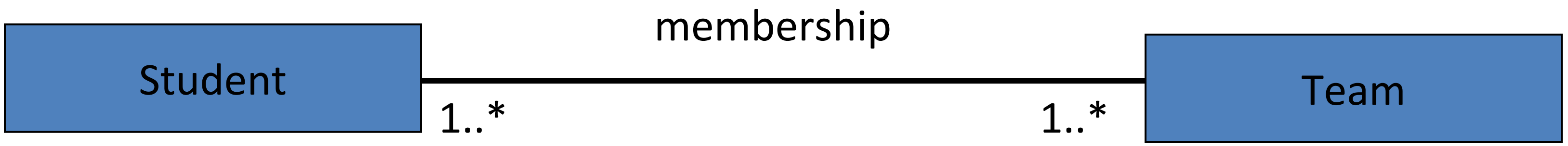
Association Role

- We can also indicate the behavior of an object in an association (*i.e.,* the *role* of an object) using *rolenames called association role*
  - A **role** is an end of an association where it connects to a class.
  - May be named to indicate the role played by the class attached to the end of the association path.
  - Usually a noun or noun phrase
- Multiplicity can also be added to the association role to indicate how many objects of one class relate to one object of another class.



Association Relationships (Cont'd)

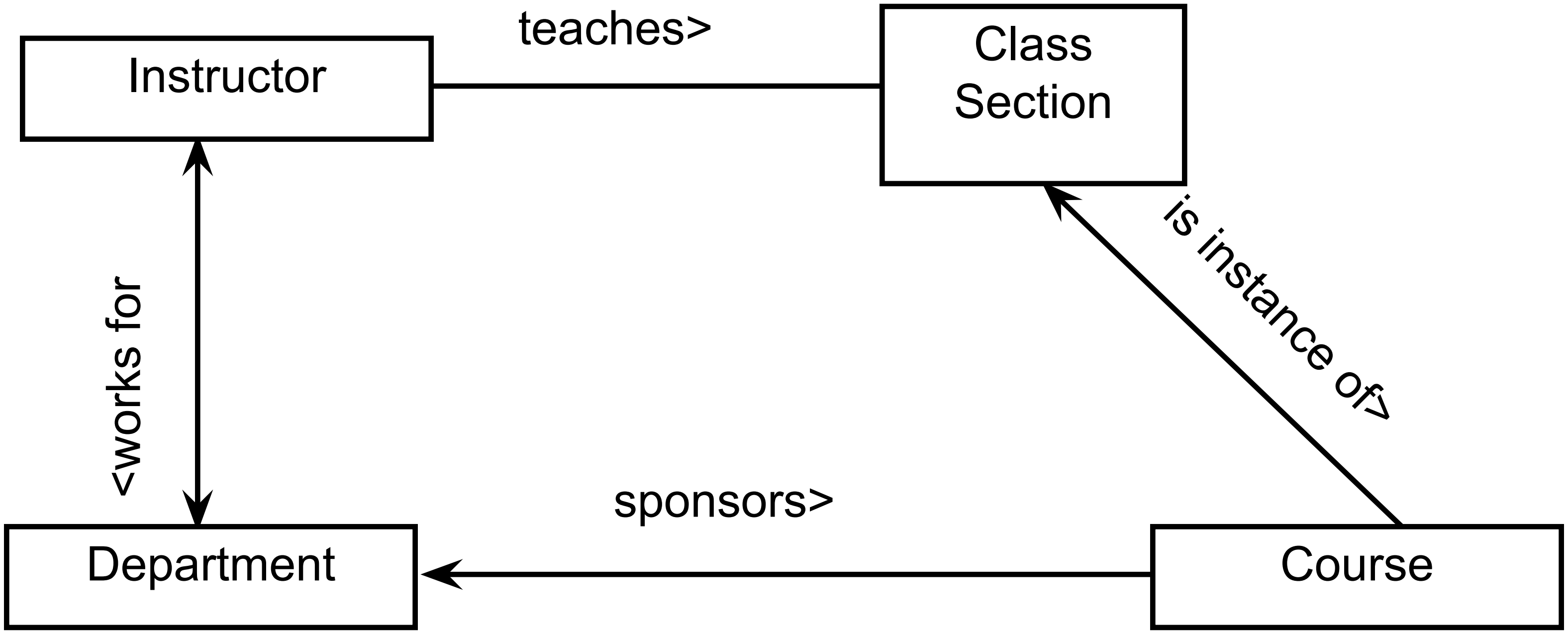
We can also name (label) the association.



Associations

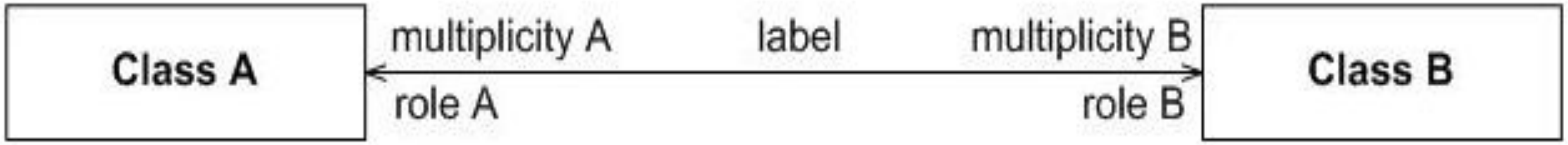
Navigation

- The navigation of associations can be
  - uni-directional
  - bi-directional
  - unspecified
- Navigation is specified by the arrow, not the label



In short Associations:

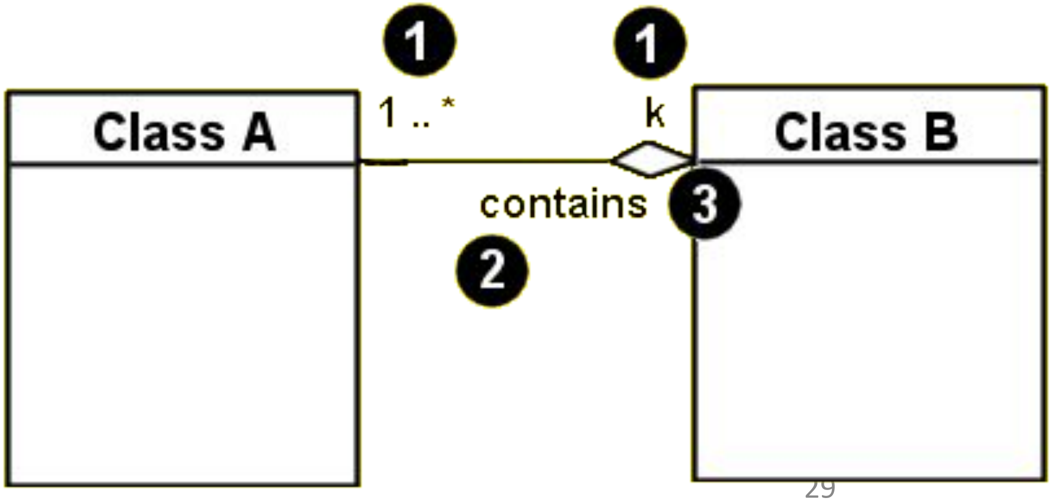
- Connect two classes
- Have an optional label
- Have multiplicities
- Are directional
- Have optional roles



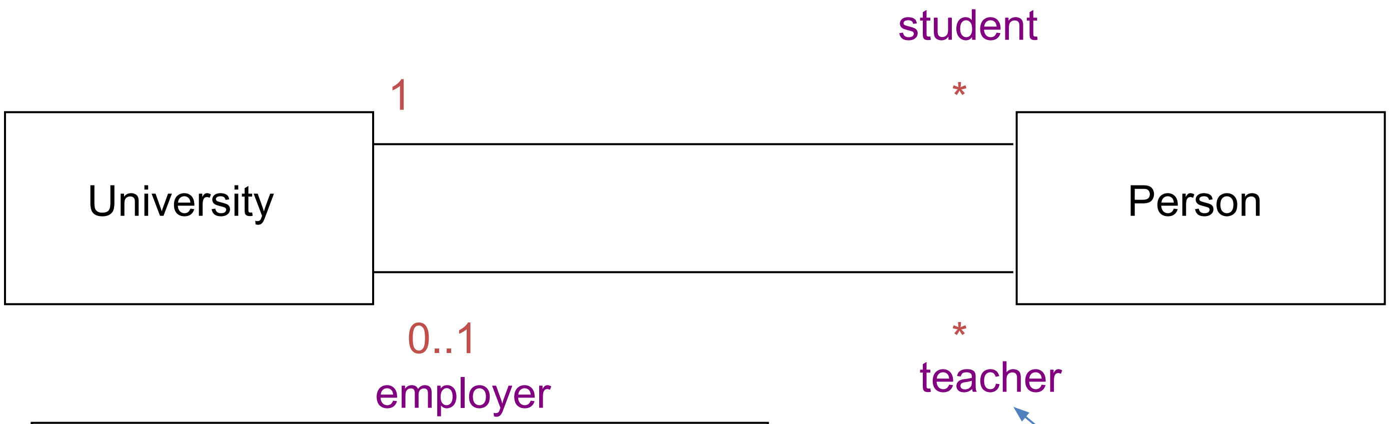
Associational relationships

- associational (usage) relationships

1. multiplicity (how many are used)
   - * ⇒ 0, 1, or more
   - 1 ⇒ 1 exactly
   - 2..4 ⇒ between 2 and 4, inclusive
   - 3..* ⇒ 3 or more
2. name (what relationship the objects have)
3. navigability (direction)



Associations



Multiplicity

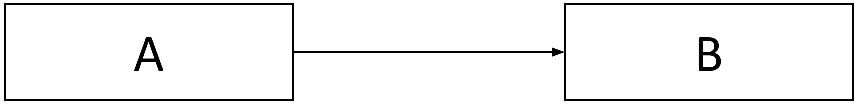
Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

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

C++ Example

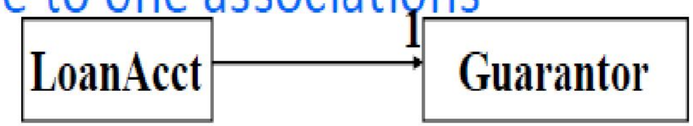
Association is typically implemented with a pointer or reference instance variable.



```
class A
{
  private:
    B* b;
};
```

Cont..

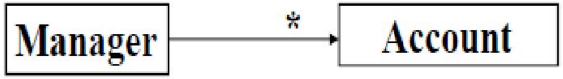
- One to one associations



```
Class loanAccount{
  guarantor * theguarantor;

  public:
  loanAccount(Guarantor g)
  {  theguarantor=g;}
};
```

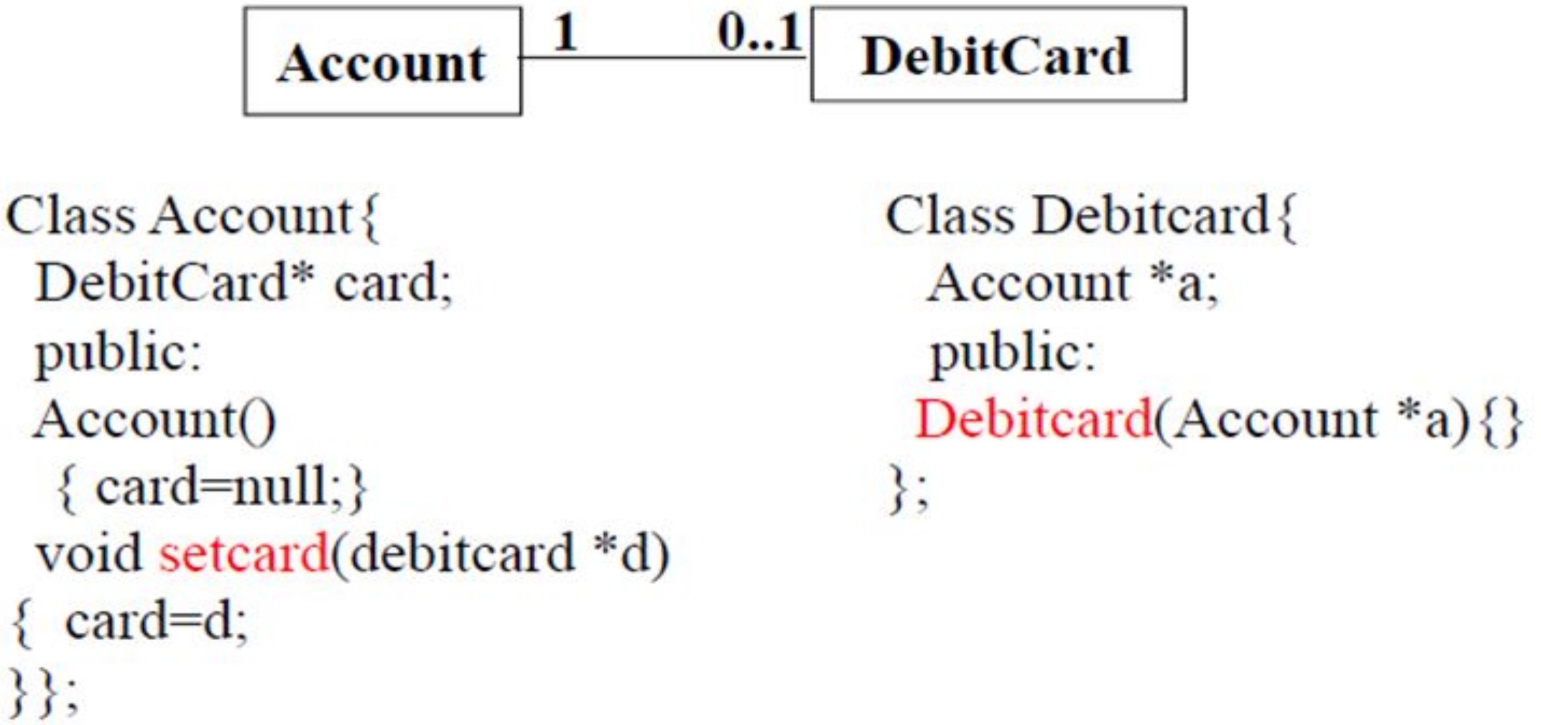
- one to many



```
Class manager{
  Account * acc[10];
  int index;
 public:
 void addaccount(Account  *a)
  { acc[index++]=a;}
};
```

Cont..

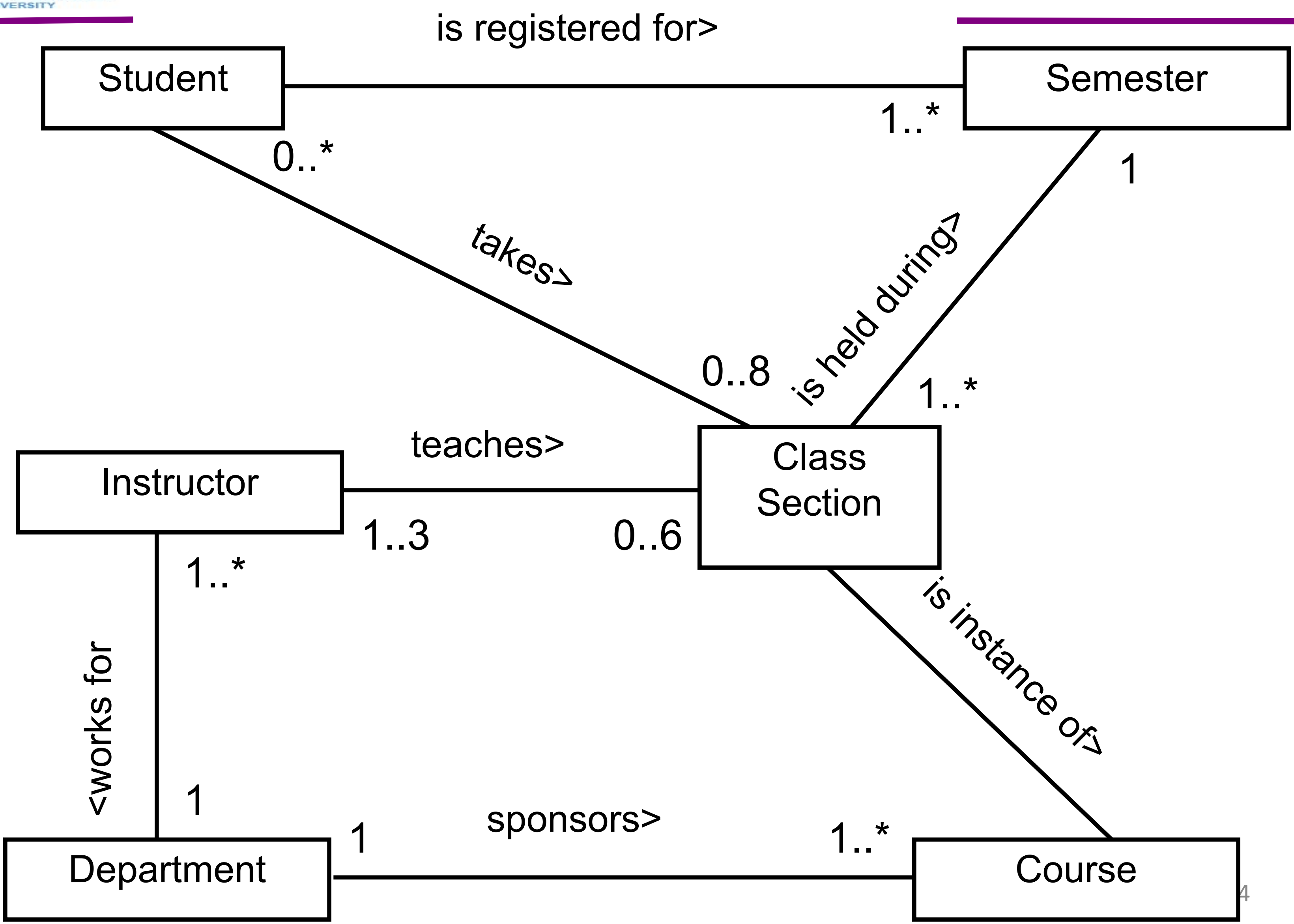
- One-to-one optional Association/mutable in one direction



```
Class Account{
 DebitCard* card;
 public:
 Account()
  { card=null;}
 void setcard(debitcard *d)
{  card=d;
}};
```

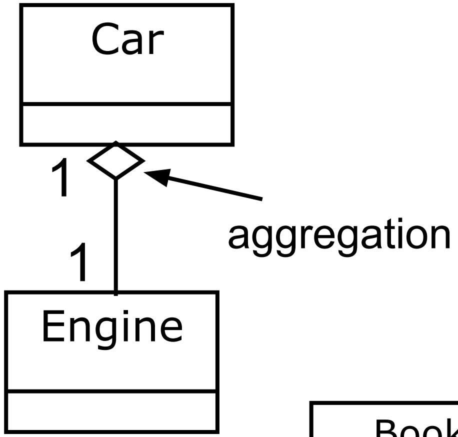
```
Class Debitcard{
  Account *a;
  public:
 Debitcard(Account *a){}
};
```

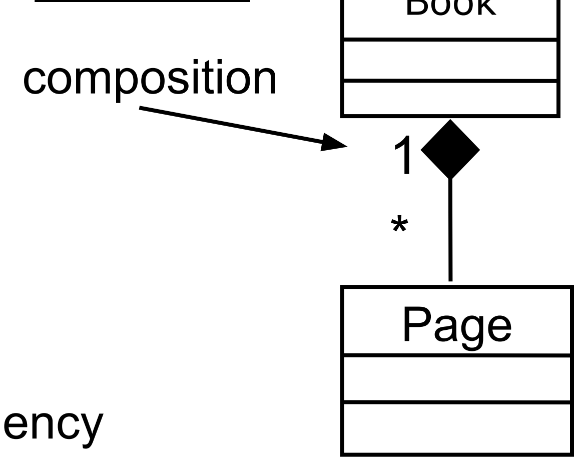
Cont..

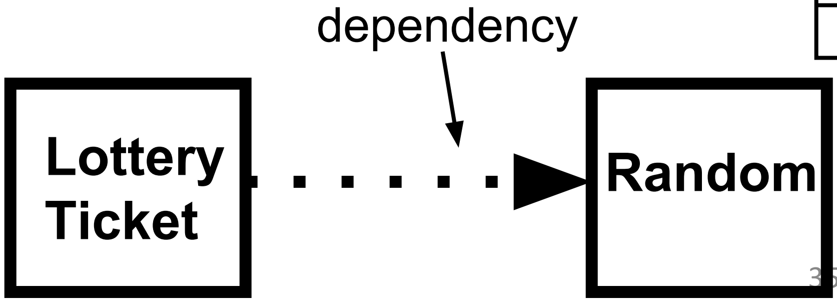

is registered for>
Student
Semester
1..*
0..*
1
takes>
is held during>
0..8
1..*
teaches>
Class Section
Instructor
1..3
0..6
1..*
is instance of>
<works for
1
sponsors>
1
1..*
Department
Course

Association types

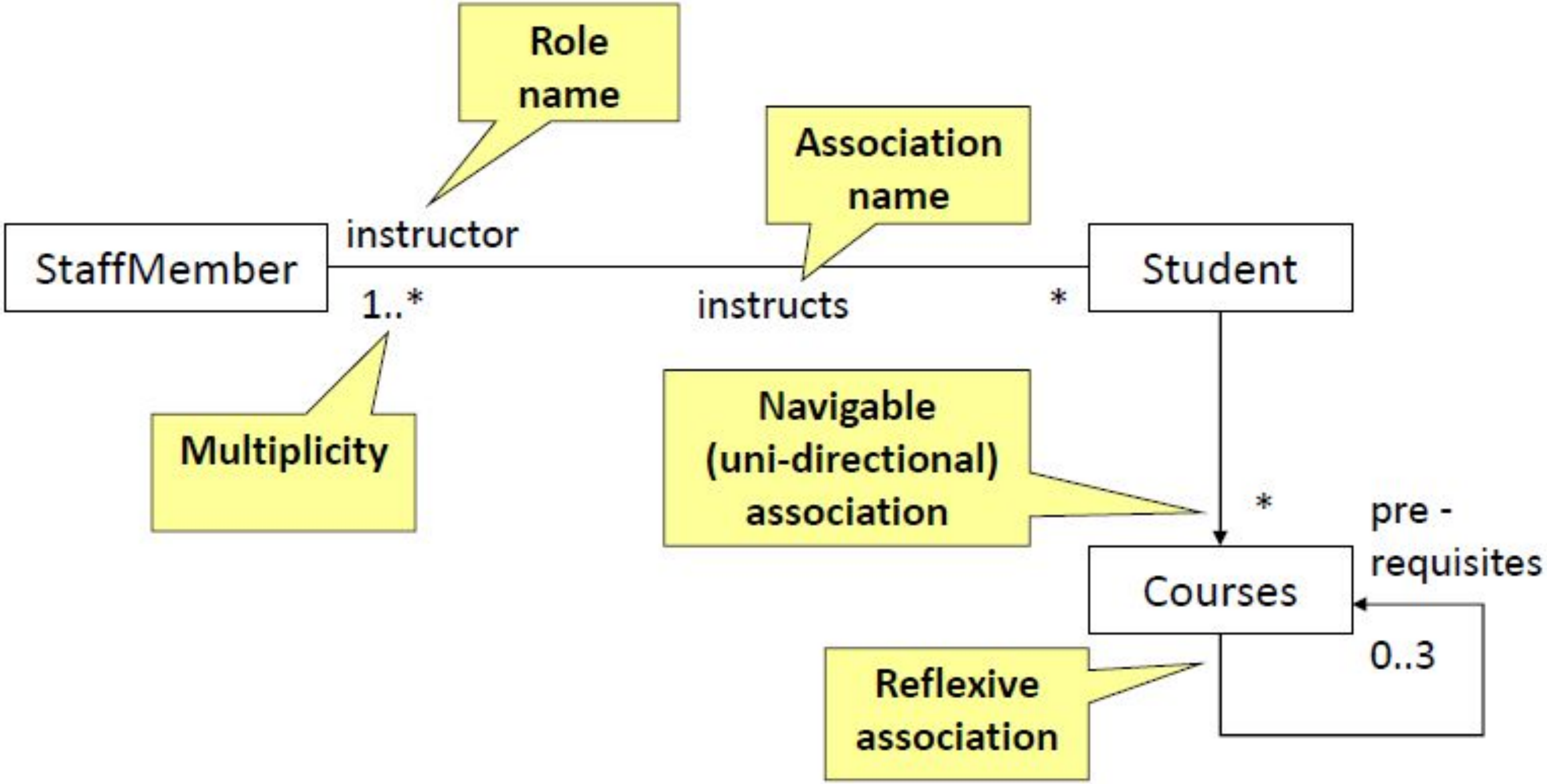
- **aggregation**: "is part of"
  - symbolized by a clear white diamond
- **composition**: "is entirely made of"
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond
- **dependency**: "uses temporarily"
  - symbolized by dotted line
  - often is an implementation detail, not an intrinsic part of that object's state







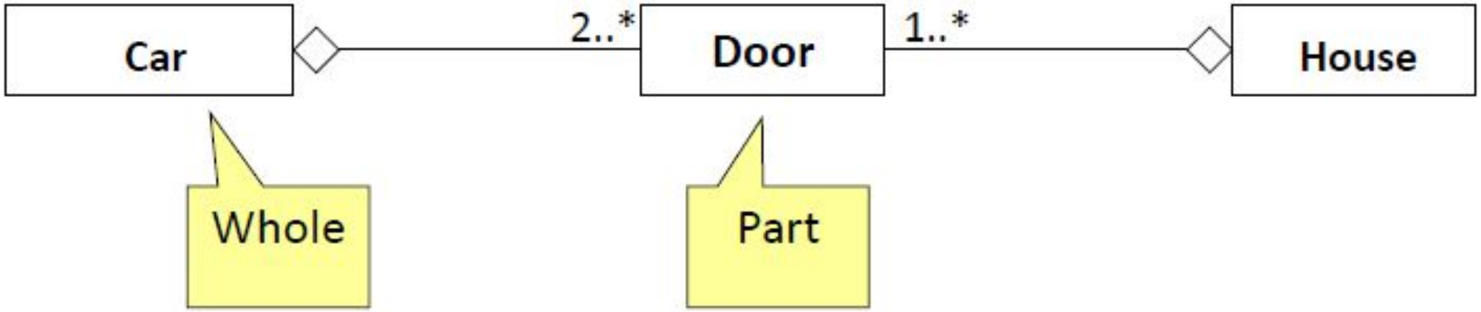
Association



Aggregation

- Aggregation is a special type of association used to model a "whole- parts" relationship. Where one class is considered as whole, made up of one or more classes comprising its parts.

- To represent an aggregation relationship, you draw a solid line from the parent class to the part class, and draw an unfilled diamond shape on the parent class's association end.
- A parts class can exist without a whole, but when they are aggregated to a whole, they are used to comprise that class.



Composition

A strong form of aggregation:

- The whole is the sole owner of its parts. The part object may belong to only one whole.

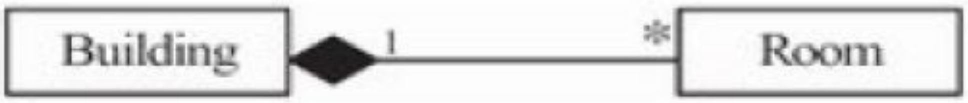
- The life time of the part class depend upon the whole class.

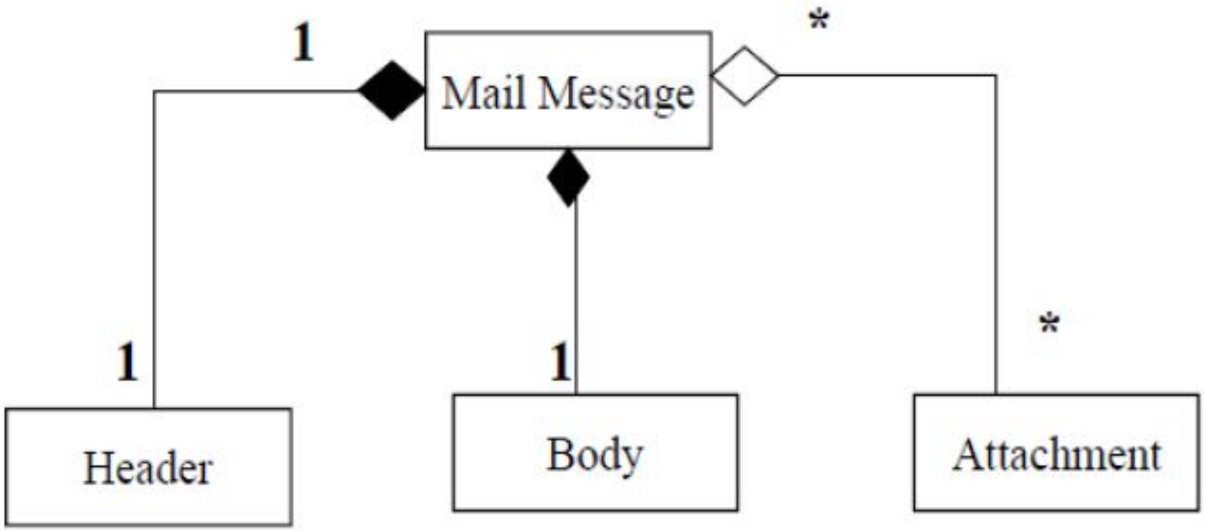
- Destruction of the whole class means the destruction of the part classes.

- Part classes are mandatory, multiplicity of at least one is always implied for the whole class.

Composition

- Composition relationship is drawn like the aggregation relationship, but this time the diamond shape is filled.





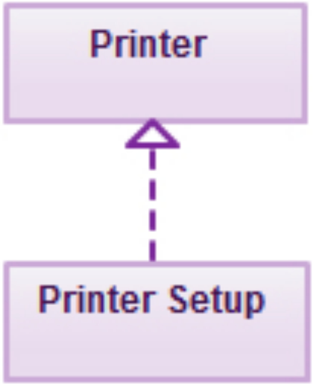
Realization

- In UML modeling, a realization relationship is a relationship between two model elements, in which one model element (the client) realizes (implements or executes) the behavior that the other model element (the supplier) specifies.

Realizations can only be shown on class or component diagrams. A realization is a relationship between classes, interfaces, components and packages that connects a client element with a supplier element.

```
symbolic of realization --------▷
```

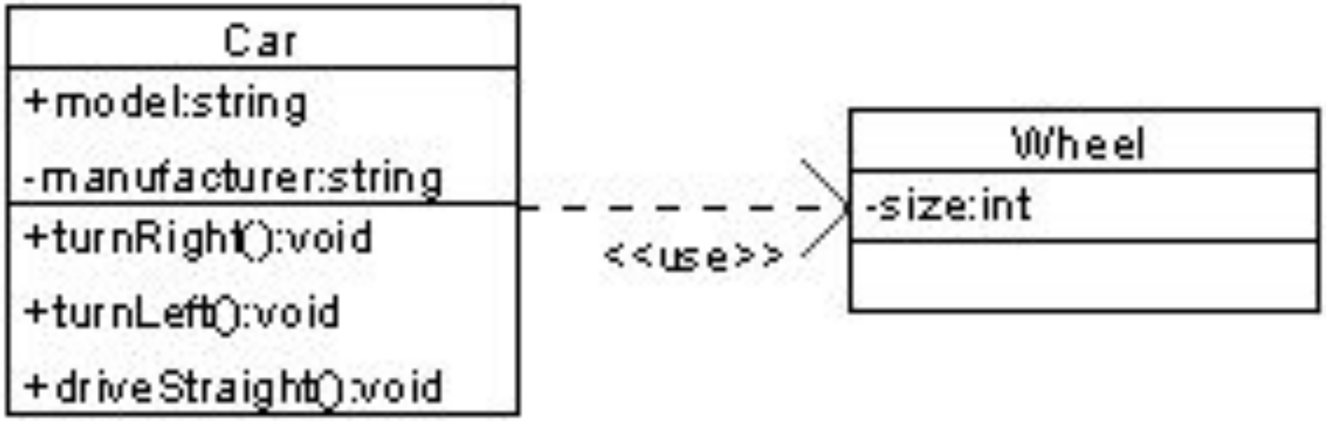
In the example, the printing preferences that are set using the printer setup interface are being implemented by the printer.



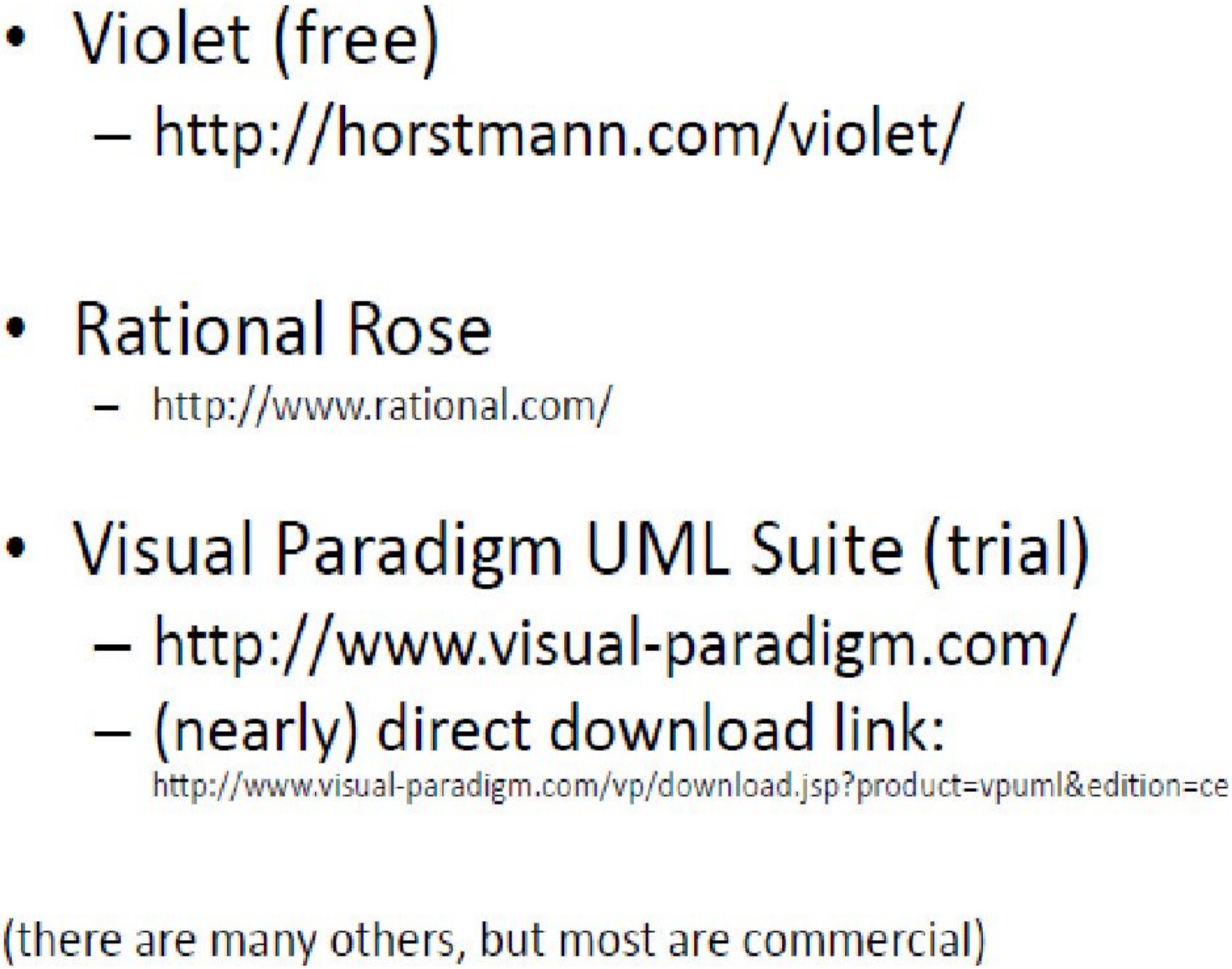
Dependencies

- Dependency is a weaker form of relationship which indicates that one class depends on another because it uses the other at some point in time.

A *dependency* is a semantic connection between dependent and independent model elements. It exists between two elements if changes to the definition of one element (the server or target) may cause changes to the other (the client or source). This association is uni-directional.



Tools for creating UML diagrams



- Violet (free)
  - http://horstmann.com/violet/
- Rational Rose
  - http://www.rational.com/
- Visual Paradigm UML Suite (trial)
  - http://www.visual-paradigm.com/
  - (nearly) direct download link:
    http://www.visual-paradigm.com/vp/download.jsp?product=vpuml&edition=ce

(there are many others, but most are commercial)

Class diagram pros/cons

- Class diagrams are great for:
  - discovering related data and attributes
  - getting a quick picture of the important entities in a system
  - seeing whether you have too few/many classes
  - seeing whether the relationships between objects are too complex, too many in number, simple enough, etc.
  - spotting dependencies between one class/object and another

- Not so great for:
  - discovering algorithmic (not data-driven) behavior
  - finding the flow of steps for objects to solve a given problem
  - understanding the app's overall control flow (event-driven? web-based? sequential? etc.)

Online shopping domain model

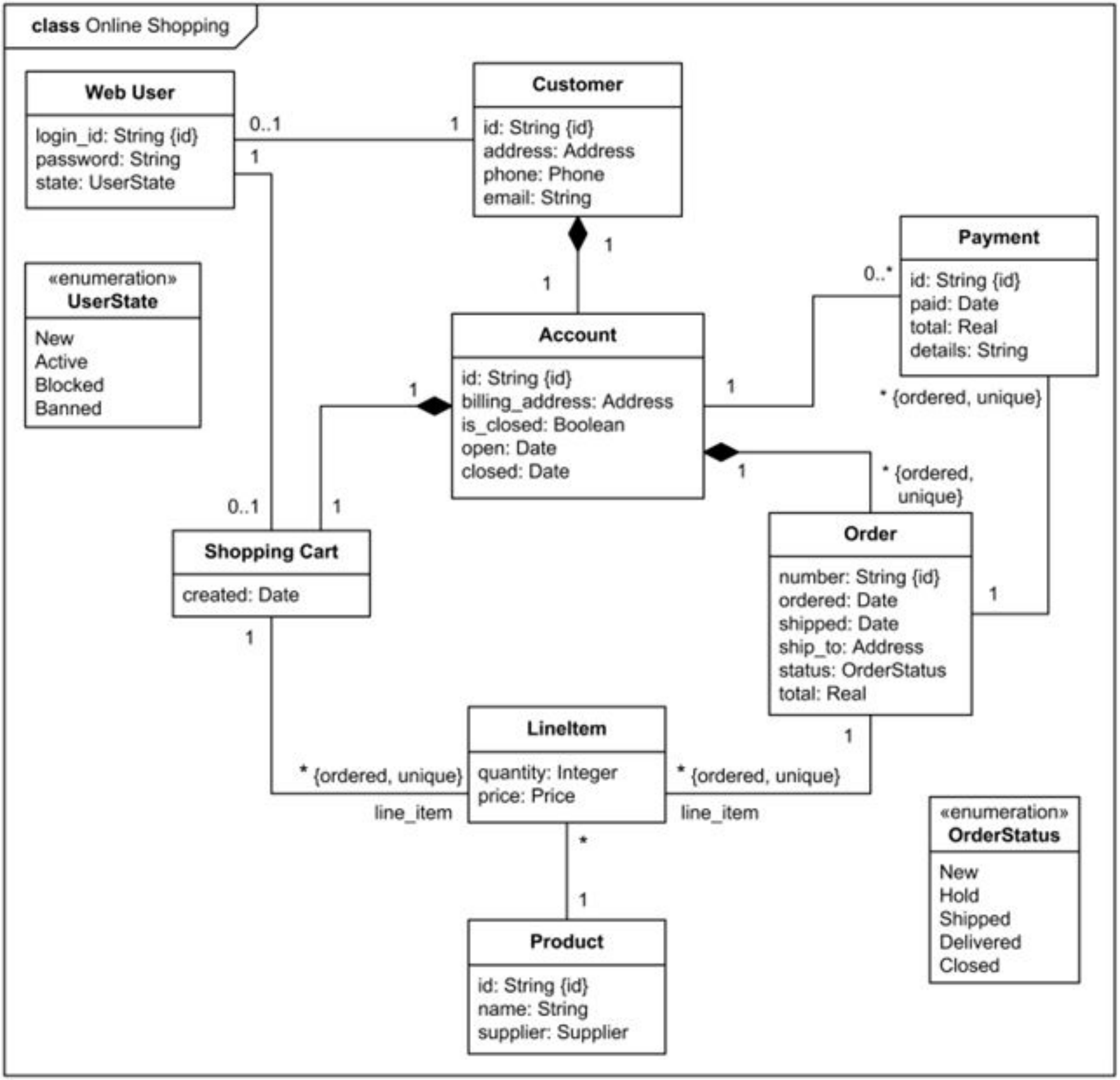
***Purpose**: Show some domain model for online shopping - Customer, Account, Shopping Cart, Product, Order, Payment.*

***Summary**: Each customer has unique id and is linked to exactly one account. Account owns shopping cart and orders. Customer could register as a web user to be able to buy items online. Customer is not required to be a web user because purchases could also be made by phone or by ordering from catalogues. Web user has login name which also serves as unique id. Web user could be in several states - new, active, temporary blocked, or banned, and be linked to a shopping cart. Shopping cart belongs to account.*

Account owns customer orders. Customer may have no orders. Customer orders are sorted and unique. Each order could refer to several payments, possibly none. Every payment has unique id and is related to exactly one account.

Each order has current order status. Both order and shopping cart have line items linked to a specific product. Each line item is related to exactly one product. A product could be associated to many line items or no item at all.

Example


class Online Shopping
Web User
login_id: String {id}
password: String
state: UserState
0..1
1
Customer
id: String {id}
address: Address
phone: Phone
email: String
1
1
1
«enumeration»
UserState
New
Active
Blocked
Banned
Payment
id: String {id}
paid: Date
total: Real
details: String
0..*
Account
id: String {id}
billing_address: Address
is_closed: Boolean
open: Date
closed: Date
1
1
1
* {ordered, unique}
* {ordered, unique}
0..1
1
Shopping Cart
created: Date
Order
number: String {id}
ordered: Date
shipped: Date
ship_to: Address
status: OrderStatus
total: Real
1
1
1
LineItem
quantity: Integer
price: Price
* {ordered, unique}
line_item
* {ordered, unique}
line_item
*
1
Product
id: String {id}
name: String
supplier: Supplier
«enumeration»
OrderStatus
New
Hold
Shipped
Delivered
Closed