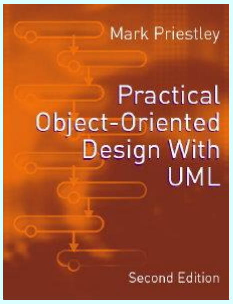
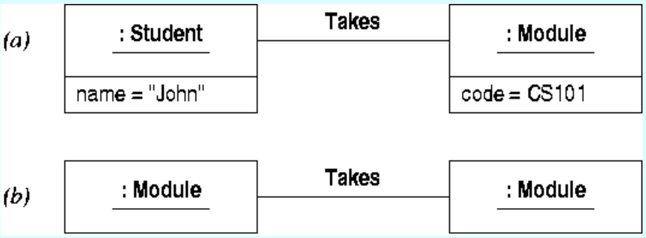
PRACTICAL OBJECT-ORIENTED DESIGN WITH UML 2e



Chapter 8:
Class and Object Diagrams

Static Models

- *Static* models describe a system's data
- Object diagrams show a 'snapshot' of the data at a given moment
- They can show both valid and invalid states:

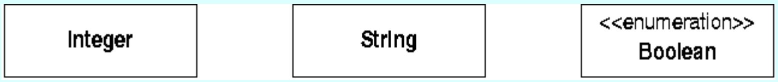


Class diagrams

- Class diagrams specify a system’s data structures, including:
  - what objects can exist
  - what data they encapsulate
  - how they can be related
- Valid object diagrams are ‘instances’ of a class diagram
  - eg the class diagram would specify that only students can take modules

UML Data Types

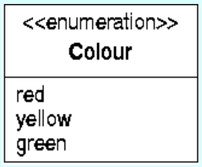
- UML defines familiar primitive data types
  - *data values* are instances of data types
  - unlike objects, values have no identity
- Data types are represented as classes:



- The values of these types are left implicit

Enumerations

- New enumerations can be defined
  - – values are *enumeration literals*
  - – specified in lower section of icon



<<enumeration>>
Colour

red
yellow
green

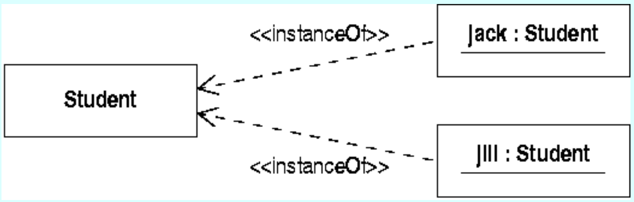
- Programming language types can also be used in UML models

Multiplicity

- Multiplicities specify how often an entity can occur in some context
  - – a general notion used throughout UML
- Represented by *ranges*
  - – a range has lower and upper bounds, eg 0..9
  - – * represents an unbounded multiplicity, eg 1..*
  - – 0..* (‘zero or more’) is often abbreviated as *
  - – 0..1 represents an optional entity
  - – 1..1 is abbreviated to simply 1

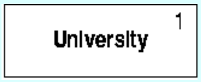
Classes

- A *class* describes a set of similar objects
  - – eg that share data and operations
- The objects are the *instances* of the class



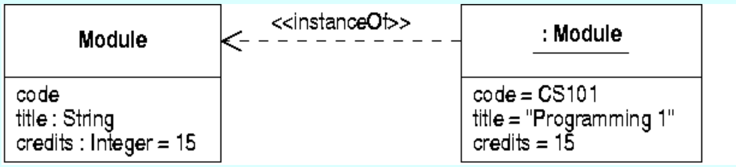
Class multiplicity

- Class multiplicity specifies the number of instances a class can have
- The default is ‘0..*’, ie there is no limit placed on the number of instances
- Sometimes it is useful to specify a *singleton* class that can only have one instance



Attributes

- Attributes describe data fields
  - in a class, attributes can have a *type*
  - which defines the *values* that an object can hold



Module

code
title : String
credits : Integer = 15

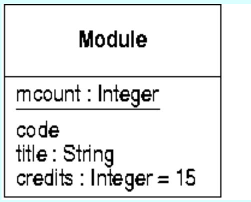
<<instanceOf>>

: Module

code = CS101
title = "Programming 1"
credits = 15

Attribute scope

- By default attributes have *instance scope*
  - each instance can have a different value
- An attribute with *class scope* has one value
  - shared by all instances of the class (‘static’)
  - attributes with class scope are underlined



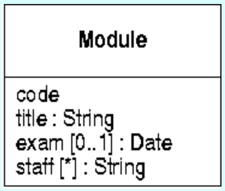
Module

mcount : Integer

code
title : String
credits : Integer = 15

Attribute Multiplicity

- Attribute multiplicity defines how many values an object stores for a attribute
  - – default is ‘exactly 1’
  - – ‘optional’ multiplicity shows possible null values
  - – arrays modelled by ‘many’ multiplicity

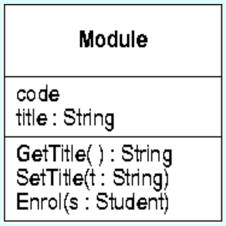


Module

code
title : String
exam [0..1] : Date
staff [*] : String

Operations

- Operations define behaviour provided by every instance of the class
  - – defined in optional lower section of class icon
  - – parameters and return types optional



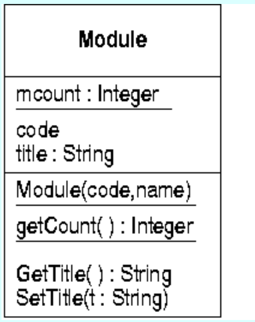
Module

code
title : String

GetTitle( ) : String
SetTitle(t : String)
Enrol(s : Student)

Operation Scope

- Operations can have instance or class scope
  - static functions and constructors shown with class scope



Module

mcount : Integer

code
title : String

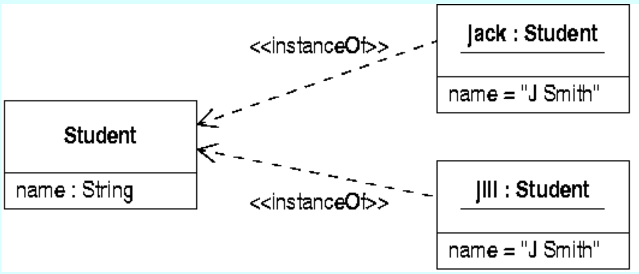
Module(code,name)

getCount( ) : Integer

GetTitle( ) : String
SetTitle(t : String)

Object Identity

- Object identities are implicit
  - not the same as an attribute
  - objects can share all attribute values and still be distinct



Student

name : String

<<instanceOf>>

Jack : Student

name = "J Smith"

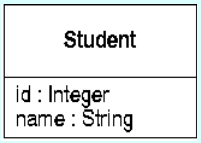
<<instanceOf>>

Jill : Student

name = "J Smith"

Object Identifiers

- Many classes will have attributes with unique values
  - corresponding to real-world identifiers
  - UML notation does not specify uniqueness



Student

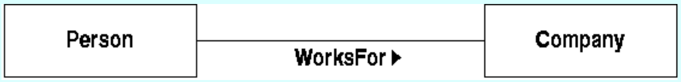
id : Integer
name : String

Visibility of Class Features

- Attributes and operations can have a *visibility*
  - parallel to Java/C++ access levels
- UML defines four levels of visibility:
  - *public* (+): visible to all objects
  - *package* (~): visible to objects in same package
  - *protected* (#): visible to instances of subclasses
  - *private* (-): visible only in same object

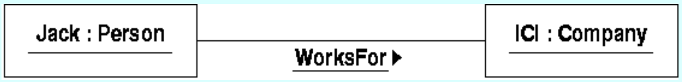
Associations

- Relationships between objects are modelled by *links*
- These relationships are specified by an *association* between the relevant classes
  - – eg a Person can work for a Company



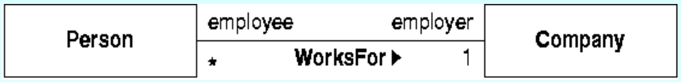
Links

- Links can be shown connecting instances of related classes



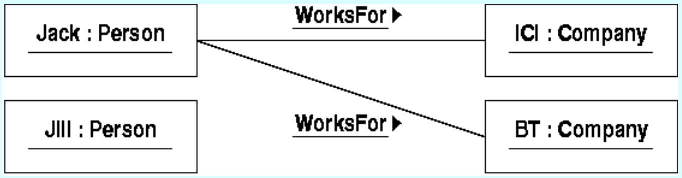
Association Ends

- Association ends can annotated with
  - a label, describing the role played the class at that end in the relationship
  - multiplicity, showing how many instances an object at the other end can be linked to



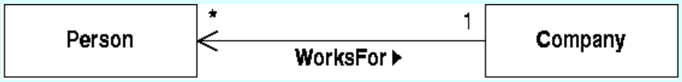
Association Multiplicity

- This association states that:
  - a Person works for exactly 1 Company
  - a Company has zero or more Persons working for it
- This rules out situations like this:



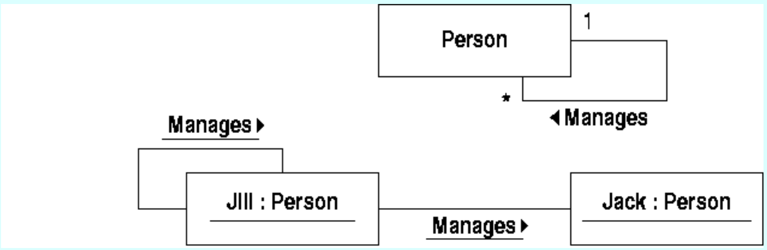
Navigability

- By default, associations can be *navigated* in either direction
  - ie given an object at one end you can access a linked object at the other, and vice versa
- Navigability can be restricted
  - sometimes we only need access in one direction



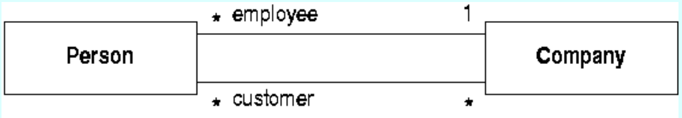
Types of Association

- Most associations are *binary*
- Some associations relate objects of the same class
  - these can be shown as *self-associations*



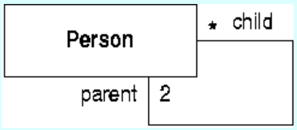
Labelling Associations

- All association labels are optional
- Multiplicity information is usually shown
- Labels are used where necessary
- Some labelling is required to distinguish associations between the same classes



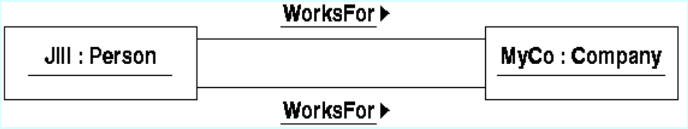
Use of Role Names

- Role names are often used to distinguish the ends of a self-association



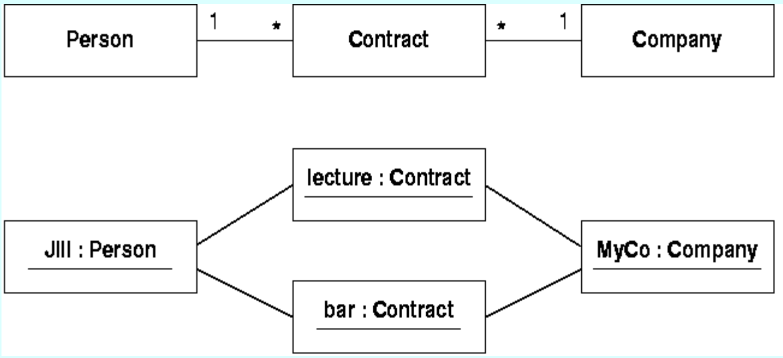
Semantics of Associations

- There can only be one instance of an association linking a given pair of objects
  - for example, a person might have two contracts with a given company
  - the model below is wrong, however



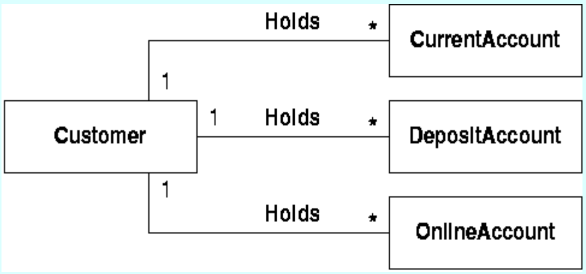
Reifying Associations

- Introduce a ‘linking’ class to deal with repeated links



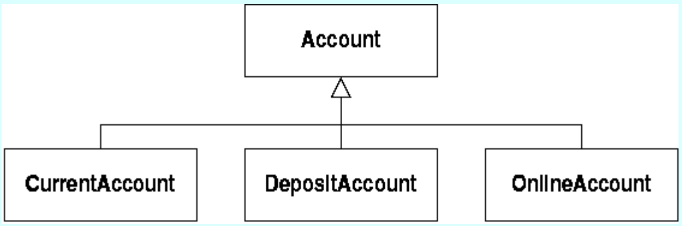
Shared Properties

- Often groups of classes share properties
  - they have the same attributes and operations
  - they share associations with other classes



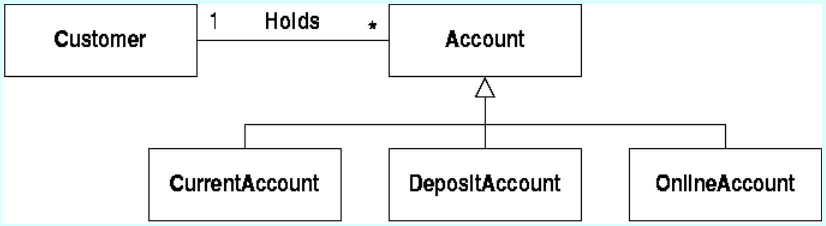
Generalization

- Models this relationship between classes
  - define a *superclass* representing the general shared properties of accounts
  - other account types are specialized *subclasses*



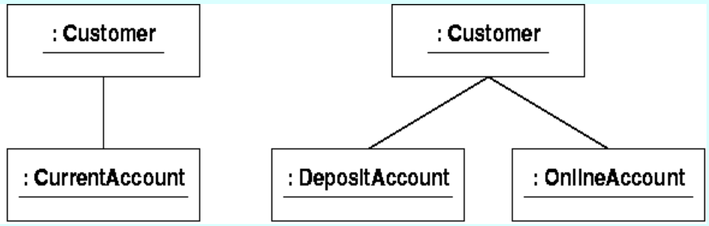
The Meaning of Generalization

- The superclass defines the properties shared by all the specialized classes
  - eg customers can hold accounts of any sort



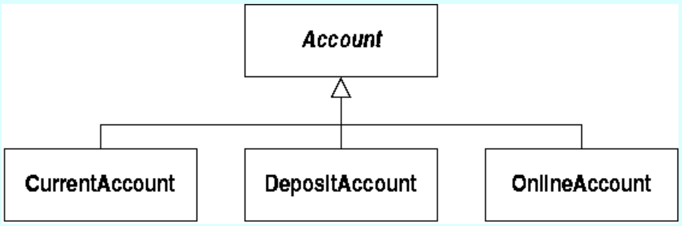
Substitutability

- This model connects customers to accounts
  - but an instance of any subclass can be *substituted* for an account object
  - these links demonstrate *polymorphism*



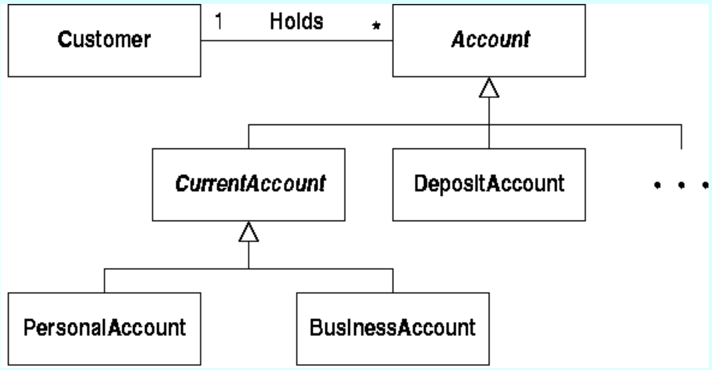
Abstract Classes

- Superclasses are often defined solely to group together shared features
  - it may not make sense to have an instance of a superclass
  - in this case, define the class as *abstract*



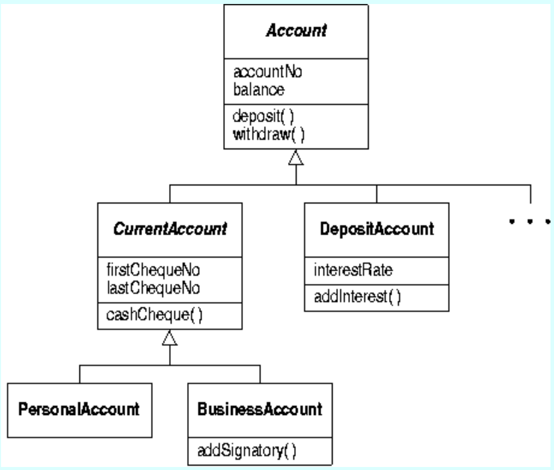
Generalization Hierarchies

- Generalization can be carried out at more than one level



Inheritance

- Inherited features also belong to subclasses



Account

accountNo
balance

deposit( )
withdraw( )

CurrentAccount

firstChequeNo
lastChequeNo

cashCheque( )

DepositAccount

interestRate

addInterest( )

. . .

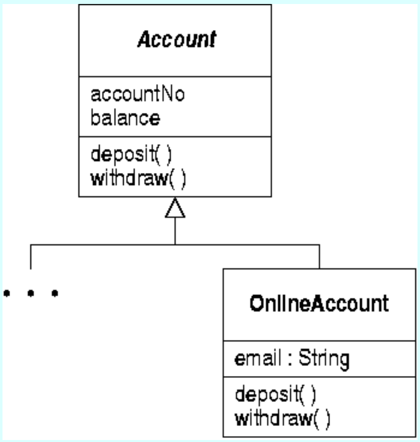
PersonalAccount

BusinessAccount

addSignatory( )

Modifying Subclasses

- Subclasses can:
  - add features to model special properties
  - override operations to implement specialized behaviour



Account

accountNo
balance

deposit( )
withdraw( )

. . .

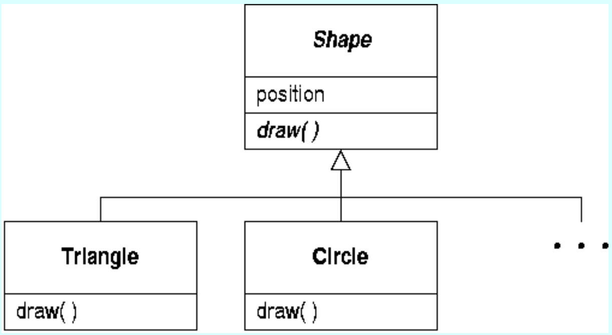
OnlineAccount

email : String

deposit( )
withdraw( )

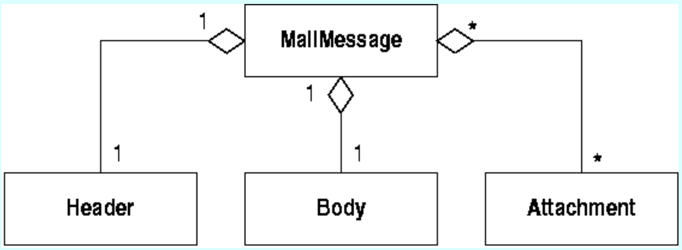
Abstract Operations

- Some operations cannot be implemented in abstract classes
  - define them as abstract and override them



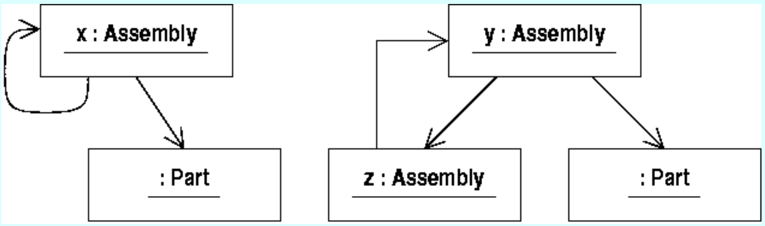
Aggregation

- Informal ‘whole-part’ relationships can be modelled using *aggregation*
  - a specialized form of an association
  - can have standard annotations on ends



Cyclic Object Structures

- Aggregation is useful for ruling out invalid cyclic object structures
  - eg where an assembly contains itself, directly or indirectly

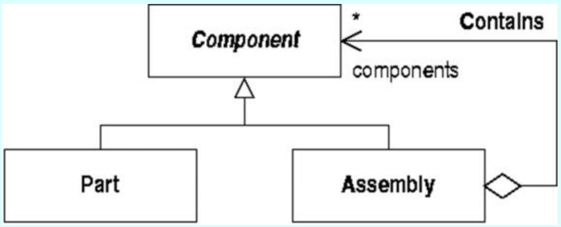


Properties of Aggregation

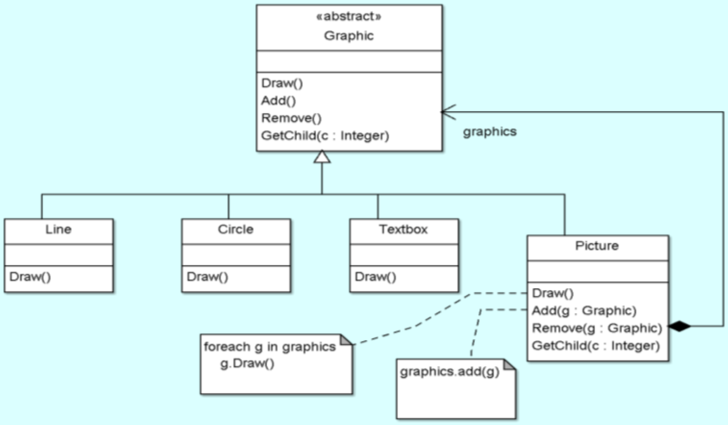
- Aggregation rules this out because it is
  - *antisymmetric*: an object can’t link to itself
  - *transitive*: if *a* links to *b* and *b* to *c*, *a* links to *c*

Recursive Data Structures

- Part-whole assemblies are related to the Composite Design Pattern

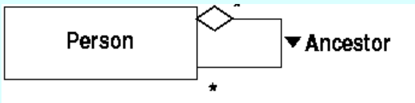


Composite Design Pattern



Meaning of Aggregation

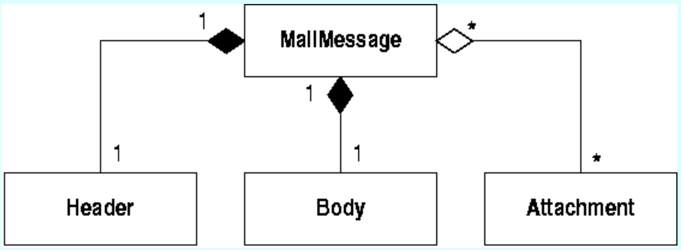
- Sometimes there is a conflict
- Eg people cannot be their own ancestors
  - this can be specified using aggregation



  - but a person’s ancestors are not parts of them!

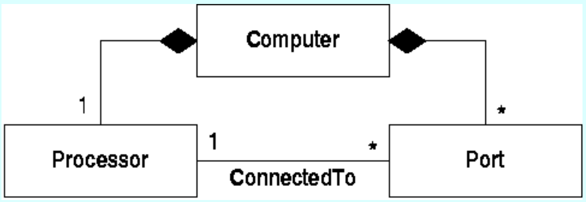
Composition

- Composition is a strong form of aggregation
  - parts can only belong to one composite at a time
  - parts are destroyed when a composite is



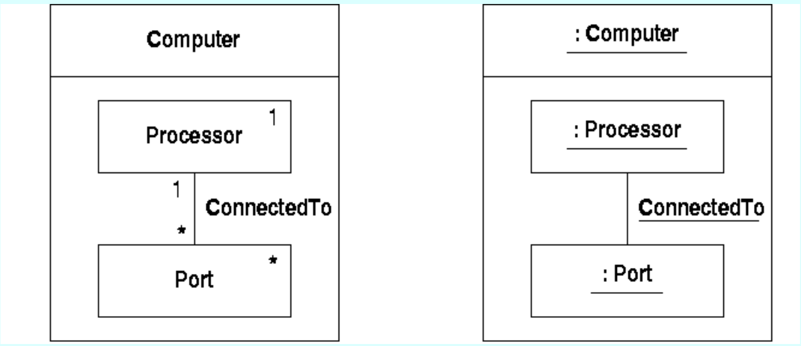
Component Relationships

- Component parts can be related even if they don’t belong to the same composite
  - sometimes this is not what is needed



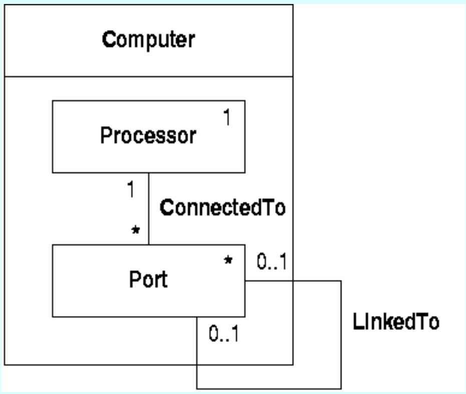
Associations and Composites

- An alternative notation allows associations to be defined inside composites



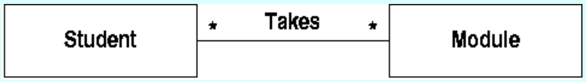
Composite Boundaries

- Associations can cross the boundary to link objects in different composites



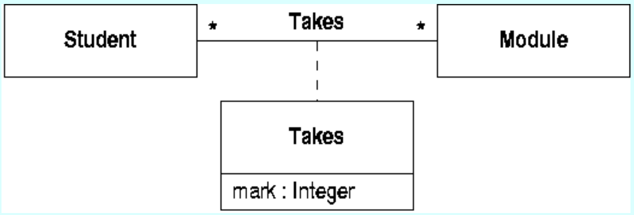
Properties of Links

- Sometimes data belongs to a link
  - a student takes a module and gets a mark for it
  - the mark only makes sense if we know the student and the module
  - so it’s not simply an attribute of either class



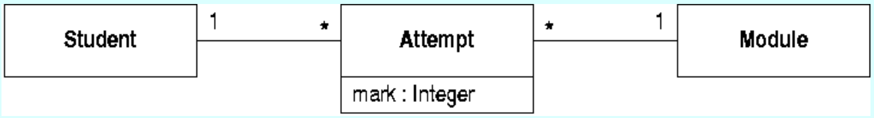
Association Classes

- Association classes share the properties of associations and classes
  - they can define links between objects
  - they allow attribute values to be stored



Reification

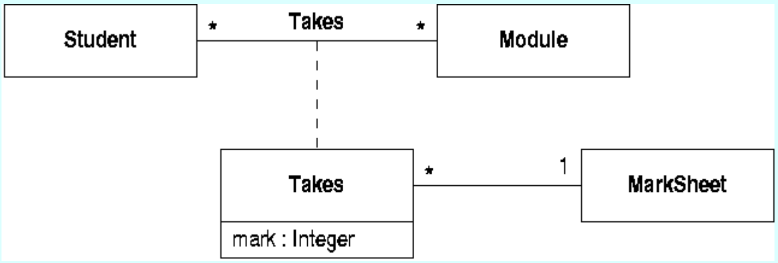
- Students’ marks could alternatively be stored in an intermediate class:



  - this has the property of allowing students to take a module more than once

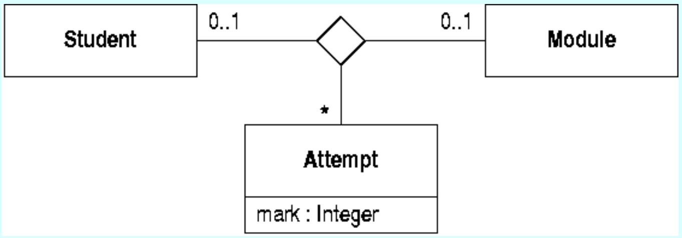
Association Class Properties

- Association classes are classes and so can participate in associations



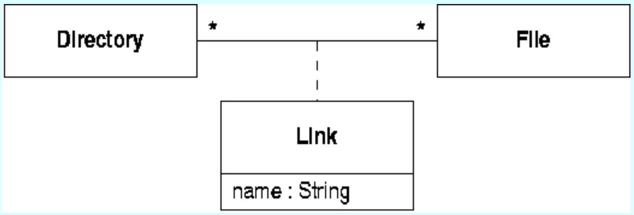
N-ary Associations

- Associations can connect more than two classes
  - A 3-way association could be used to store marks



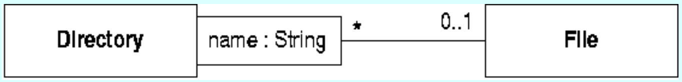
Modelling Unix Files

- Unix files can appear in many directories, under different names
- This could be modelled with an association class



Qualified Associations

- There are two problems with this:
  - it doesn’t allow the same file to appear twice in a directory (under different names)
  - it doesn’t specify that names can only be used once in each directory
- Using a qualified association solves these
  - can be implemented with a HashMap

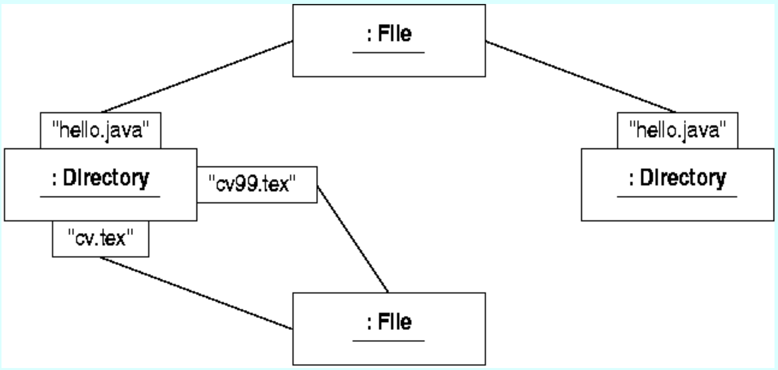


Qualifiers

- The ‘name’ attribute is known as a *qualifier*
- It acts like a ‘key’: within a directory, each name maps to zero or one file
  - this guarantees that names are unique within directories
- Files are linked to names within directories, so multiple occurrences within a directory are possible

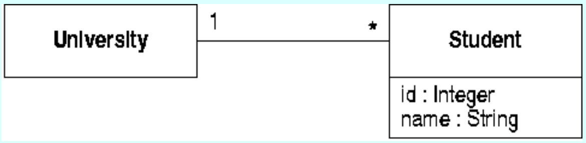
Qualified links

- Here is a typical structure of objects linked with qualifiers



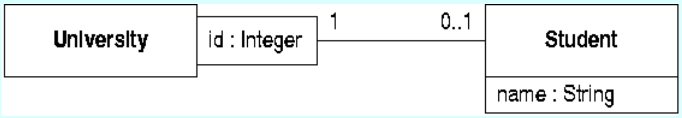
Qualifiers and Identifiers

- Use a qualifier to model an identifying attribute that is unique within some context



University 1 * Student

id : Integer
name : String

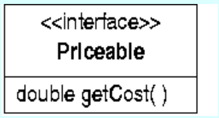


University id : Integer 1 0..1 Student

name : String

Interfaces

- An *interface* in UML is a named set of operations
  - – shown as a stereotyped class



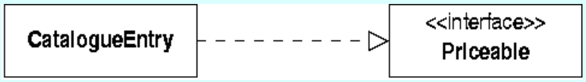
<<interface>>
Priceable

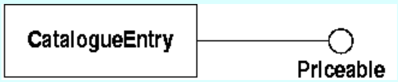
double getCost( )

- Generalization can be defined between interfaces

Realizing an Interface

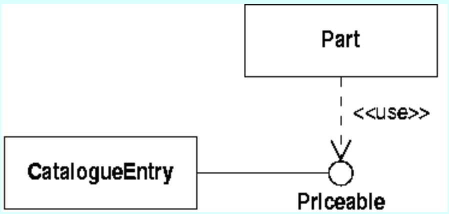
- A class *realizes* an interface if it provides implementations of all the operations
  - in Java we say it *implements* an interface
- UML provides two equivalent ways of showing this relationship



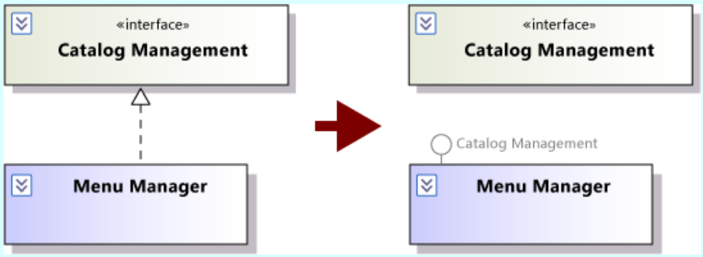


Interface Dependency

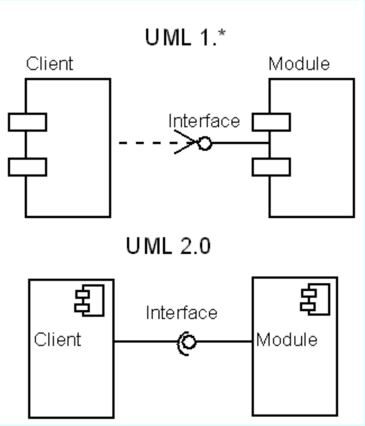
- A class can be *dependent* on an interface
  - this means that it only makes use of the operations defined in that interface



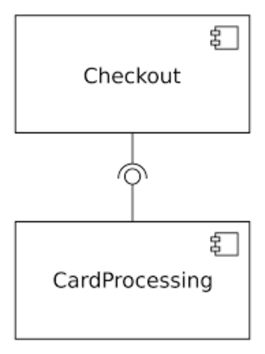
Interfaces & UML



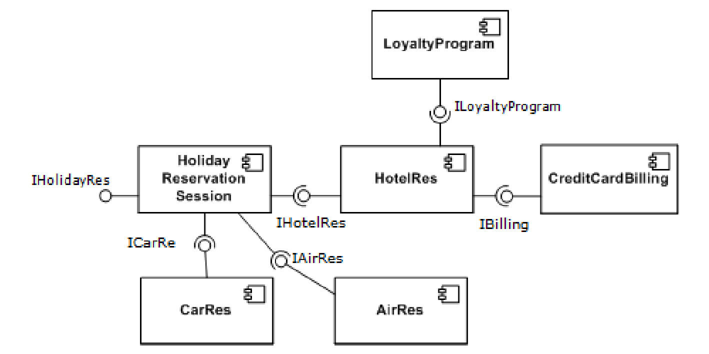
Interfaces in UML 1 & UML 2

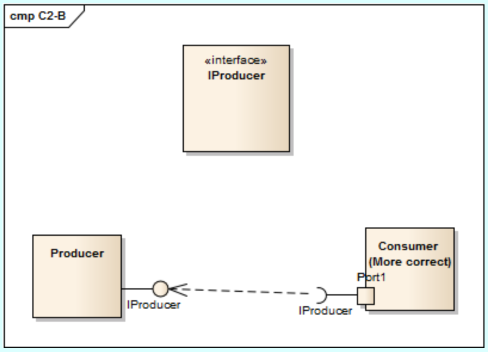


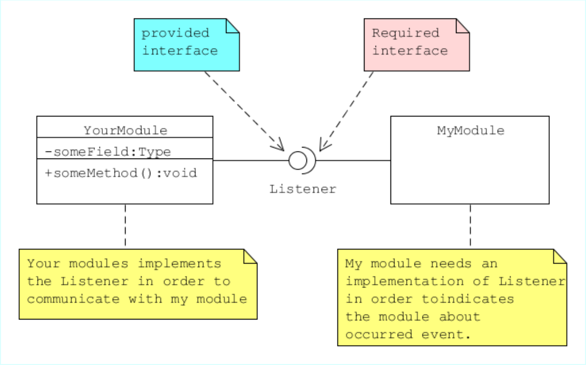
Interface - Ball & Socket notation



Components & Interfaces





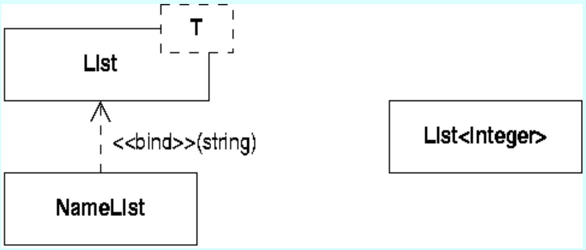


Your modules implements the Listener in order to communicate with my module

My module needs an implementation of Listener in order toindicates the module about occurred event.

Templates

- Parameterized model elements can be shown as *templates*
  - these are commonly used to show *generic* or template classes and operations (as in C++)


T
List
<<bind>>(string)
NameList
List<Integer>