Module 2
An Introduction to UML

CSCI E-247
Fall, 2002

Outline of this Module

- In this module we
  - Cover basic concepts of Object Modeling
  - Introduce notation from the Unified Modeling Language (UML)
  - Do an example to illustrate UML
  - Discuss reuse with inheritance vs reuse with composition

The Unified Modeling Language (UML)

- For many years, Rumbaugh's Object Modeling Technique(OMT), Booch's O-O methodology, and Jacobsen's Objectory methodology were the three primary, but competing, O-O methodologies
- UML combines the notation from these three into one unified object model
- UML has been adopted as a standard for Object-Oriented Analysis and Design by the Object Management Group (OMG)
- The course will use a subset of the notation of UML version 1.1 throughout
- Note: UML 2.0 is available at www.omg.org

Note that UML is a *notation* for diagramming object models, but is not a methodology, in that it does not prescribe how a project team or organization should proceed through OOA and OOD.

UML documentation and other information is available at

http://www.rational.com/uml/ and at http://www.omg.org.

The OMG is a consortium of over 850 companies devoted to defining standards of object technology, including the Common Object Request Broker Architecture (CORBA).

Visit the OMG home page at http://www.omg.org.

Basic O-O Concepts in UML

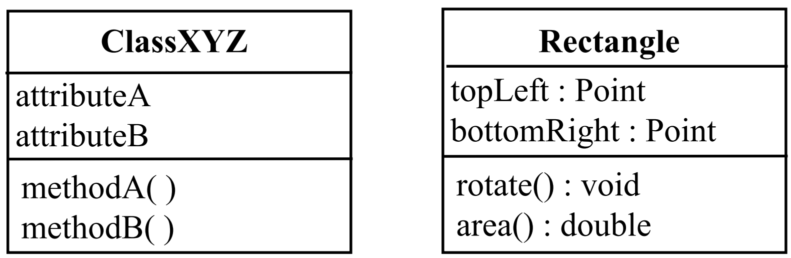
- On the next few slides, we will show how to describe the following O-O concepts in the Unified Modeling Language (UML)
- Inheritance
- Associations
- Aggregation
- Composition

There are many other kinds of relationships between classes and other kinds of features that UML defines for modeling (such as state-transition models).

We focus our attention here on the most common structural (static) relationships among classes.

UML Notation for the Class Construct

- A class defines a set of attributes (which express the state of the class) and methods (which express the behavior of the class)



The diagram above shows each of the following in a separate box:

the name of the class (in bold)

the attributes

the methods

In declaring attributes, we put the attribute name first, and if we choose to specify the attribute's type we follow the attribute name with the type, for example

`topLeft : Point`.

We use a similar convention for method declarations, for instance,

`area() : double`.

We will follow the same naming conventions for classes, data members, and methods as we do for Java.

Public, Private, Protected Access

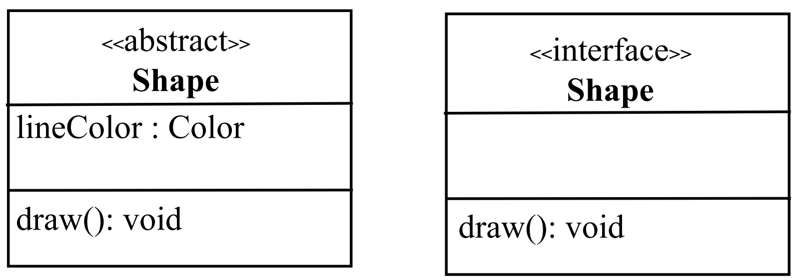
- UML does have a notation for specifying which data members and methods are public, private, or protected
- In our use of UML notation, the methods are assumed to be public methods.
- The attributes are assumed to be private, with public accessors (get/set) are implicitly defined [if an attribute is read-only, only the get accessor is defined]
- In the case of `Rectangle` on the previous slide, there are four implicitly defined accessors, which are not shown in the diagram:

```
public Point getTopLeft();
public setTopLeft(Point p);
public Point getBottomRight();
public setBottomRight(Point p);
```


We will follow a similar convention for associations, to be defined later

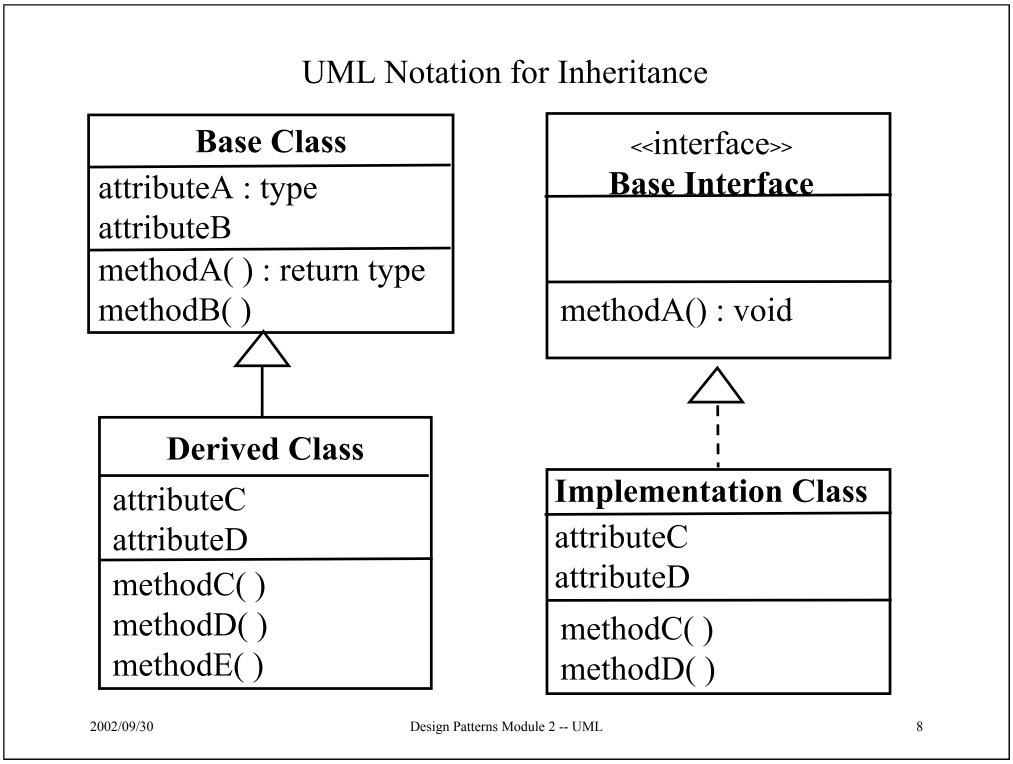
Use of Stereotypes

- Often, we want to specify that a class is abstract, or that we have an interface rather than a class
- This happens infrequently at analysis time
- The notation on a class is called a 'stereotype' in UML. It's denoted as follows



On the left, `Shape` is an abstract class, since there is a need to define an attribute common to all `Shape` objects.

On the right, `Shape` defines only methods, so it can be an interface.



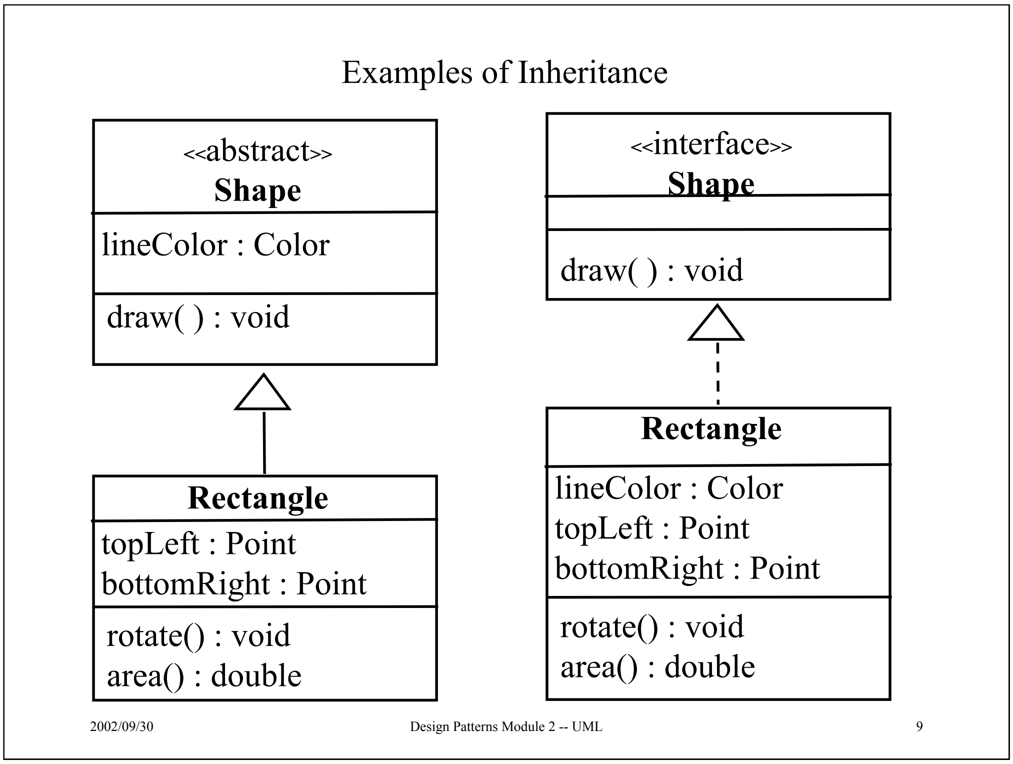
Inheritance is sometimes referred to as 'Is-A'.

You should use inheritance only when an instance of the derived class can be substituted everywhere that the base class appears.

In C++, public derivation is the means of implementing inheritance relationships. By contrast, the Java programming language allows one to declare an interface explicitly, and a class can declare that it implements multiple interfaces, while can extend only one other class.

Inheritance, as described here, is sometimes referred to as 'interface inheritance', which means the incremental definition of an interface by including other interfaces.

When there are many classes to be shown on the same diagram, it is often useful to suppress detail, such as the method list or attribute list, showing instead only a rectangle with the class name.

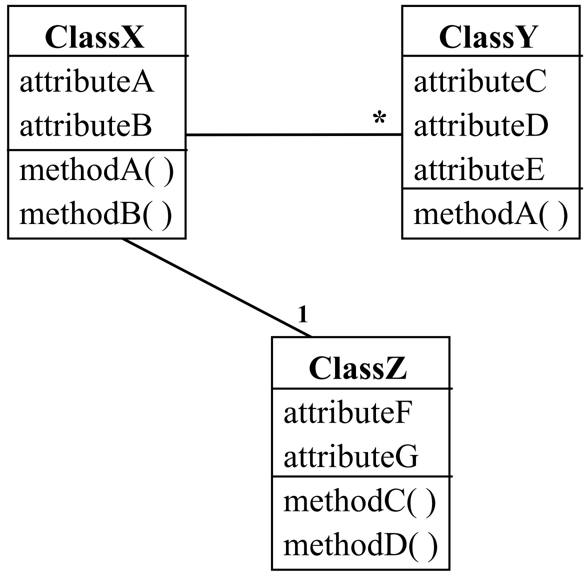


The left diagram shows class `Rectangle` extending (in Java-speak) class `Shape`. The `Shape` class shown has a very limited interface. The attribute `lineColor` is public and has two related public access methods: `getLineColor()` and `setLineColor()`.

The derived class `Rectangle` supports all the methods in `Shape` as well as the new accessor functions for its own state variables and the two new public methods.

The right diagram shows class `Rectangle` implementing (in Java-speak) interface `Shape`.

Associations



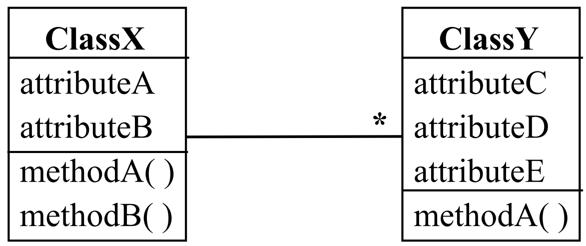
2002/09/30 Design Patterns Module 2 -- UML 10

The asterisk (*) shown at the right-hand end of the top line above means that the association involves zero or more instances of the attached class.

As written, an instance of `ClassX` is associated with zero or more instances of `ClassY` and exactly 1 instance of `ClassZ`. When the instance of `ClassX` is deleted, the associated instances might or might not be deleted as well. We say that the lifetime of the related instances of `ClassY` and `ClassZ` are independent from that of the associated instance of `ClassX`.

The * can be omitted or replaced by a specific integer or a range of integers (e.g., 10 or 1..10).

Convention for Associations



- For attributes, we expect two public accessors (get/set)
- For associations, there is a need for mechanisms to modify the association and to retrieve values from the association. Typically there are three public methods we will expect to be defined implicitly

```
void add(ClassY cy)
void delete (ClassY cy)
Iterator list ()
```

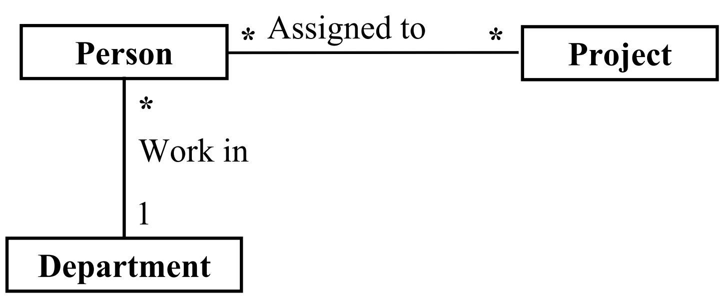

The add/delete methods mention could throw exceptions.

The list method returns an `Iterator` (perhaps a type-safe one, where `currentItem()` returns an object of type `ClassY`)

If a class participates in multiple associations, it might be necessary to differentiate the list method with multiple names (e.g., `listClassY()`, `listClassZ()`).

Example: Associations

- The diagram below models the associations described in the statement "a person works in one department and a person can be assigned to multiple projects, while projects have many persons assigned"

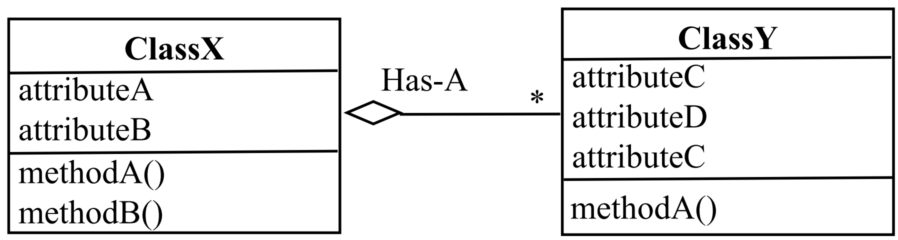


The association between the `Person` and `Project` classes is known as a ‘many-to-many’ association, while the association between `Department` and `Person` classes is known as a ‘1-to-many’ association.

Note that in this diagram, which emphasizes associations, we suppress detail about the attributes and methods on the associated classes.

Aggregation

- Aggregation is a special kind of association in which an object of a class X *contains* or *has* objects of a class Y

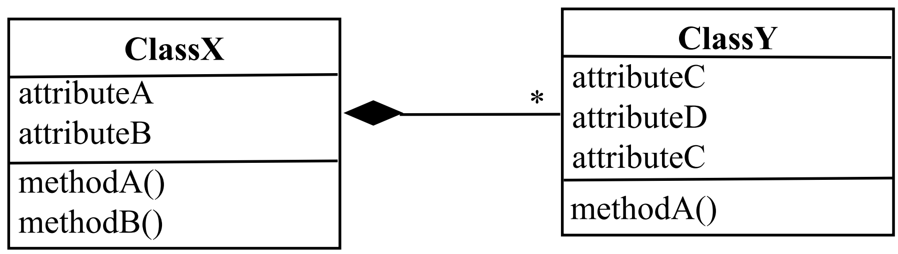


The open diamond notation in the diagram above indicates that an instance of `ClassX` has (or is the parent of) zero or more instances of `ClassY`. Aggregation is also referred to as shared aggregration, which means that the part (an instance of `ClassY` in the case above) could be shared among more than one parent instance. The open diamond does not imply that when the instance of `ClassX` is deleted, all of its parts are also deleted. We will contrast this with the UML concept of composition on the next slide.

As before, the * can be omitted or replaced by a specific integer or range of integers.

Composition

- Composition is a special kind of aggregation in which the contained object cannot be shared among multiple containers and must be deleted when the container is deleted



The notation in the diagram above indicates that an instance of `ClassX` has (or *is the parent of*) zero or more instances of `ClassY`, with the additional constraints that

a) parts cannot be shared simultaneously by two containers

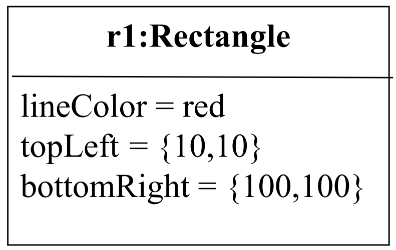
b) when the instance of `ClassX` is deleted, all of its parts are also deleted. In UML, this lifecycle dependency is referred to as *coincident lifetimes*. However, parts can be added to and removed from an instance of `ClassX` at any time, and can be moved from one container to another.

For example, a directory in a file system is a container of files and other directories. In some file systems, we would use UML composition to model this association between directories and other directories, and between directories and files. This implies that if we delete a directory, we also delete all subdirectories and files within it.

The UML use of the term "composition" differs from the use of the term in the design patterns text, where composition is more like a general association in UML. Thus the UML concept of association is more restrictive than that of the GoF, as described above.

Instance Diagrams

- An instance diagram shows how some instances of the classes in a class diagram might be related
- An instance is shown as a rectangle with the instance name followed by the class name or just an instance name. Values of the instance's attributes can be shown. Relationships to other instances are shown as arrows



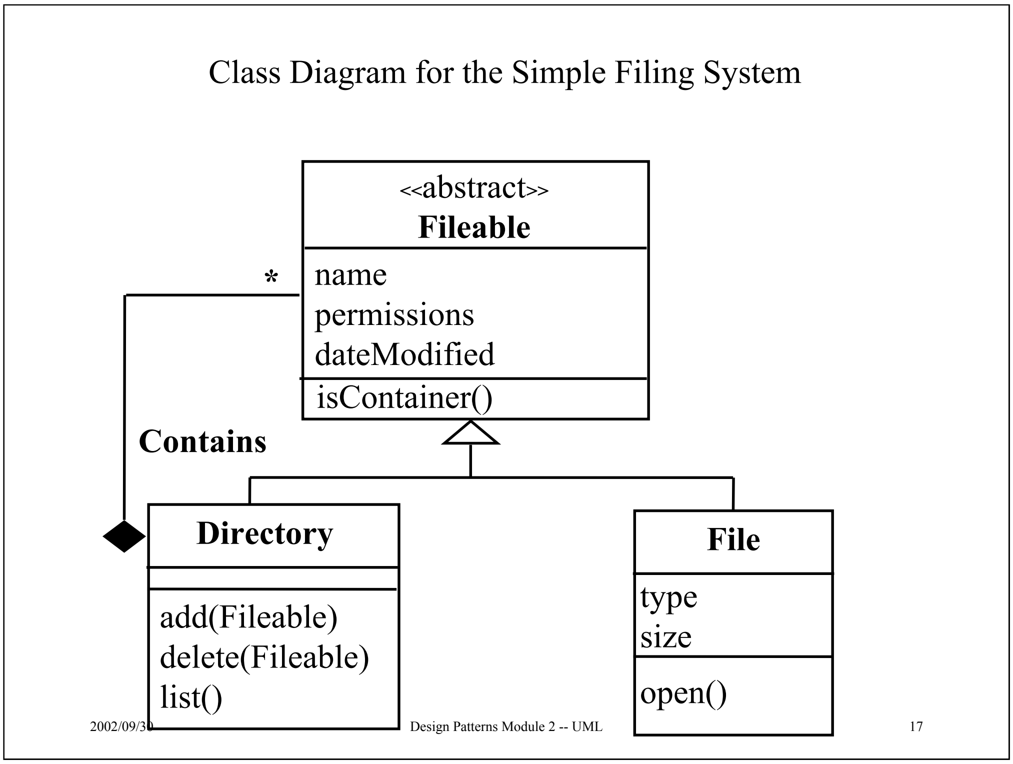
An instance diagram can be very useful to illustrate the associations that have been defined in one or more class diagrams. This usage supports the concept of walking through a scenario to validate the model.

Example: A Simple Filing System

- We want to model the objects in a simple filing system
- Every object in the filing system has a name, a date modified, and set of permissions (read, write, execute)
- Some objects, known as directory objects, are containers of any kind of object in the filing system. Other objects, known as file objects, do not contain other filing system objects
- Objects cannot be shared among containers

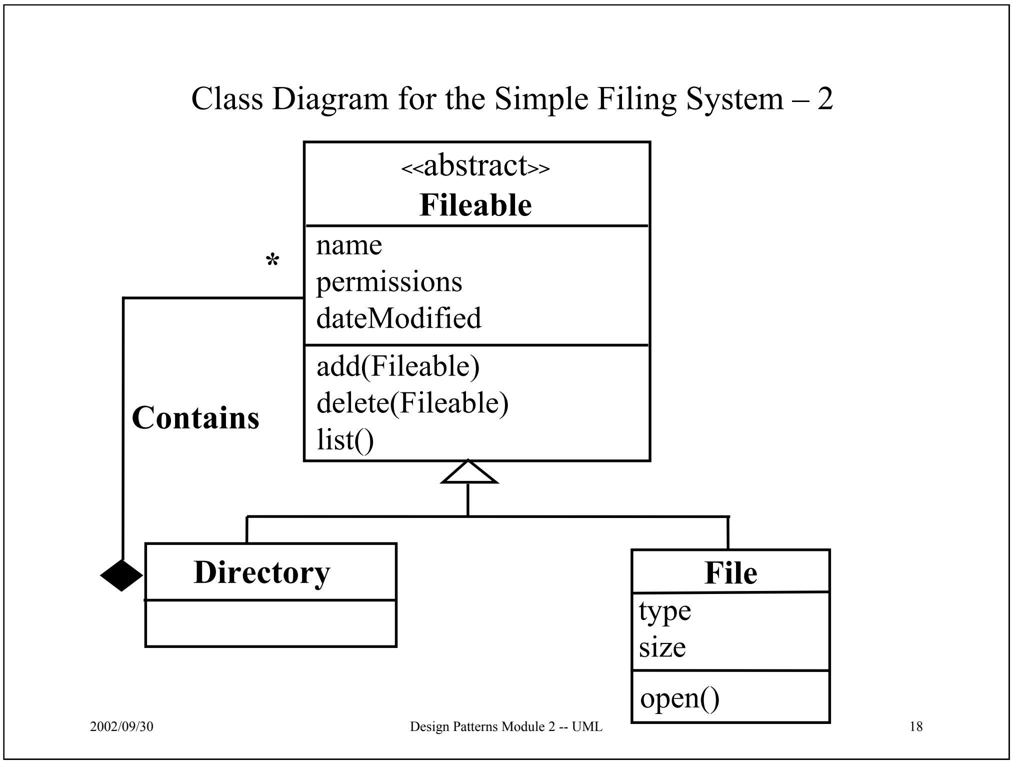
A directory object allows addition, deletion, and listing of any of its contents.

A file object has a size (in bytes) and a file type associated with it (e.g., text file, executable file,…) as well as an open method. For a text file, the open method should launch a text editor; for an executable file, the open method should launch the executable.



Note that with the interaction of the inheritance relationship and the aggregation relationship, the model defines arbitrarily deep instance hierarchies, as shown in the instance diagram on the next slide.

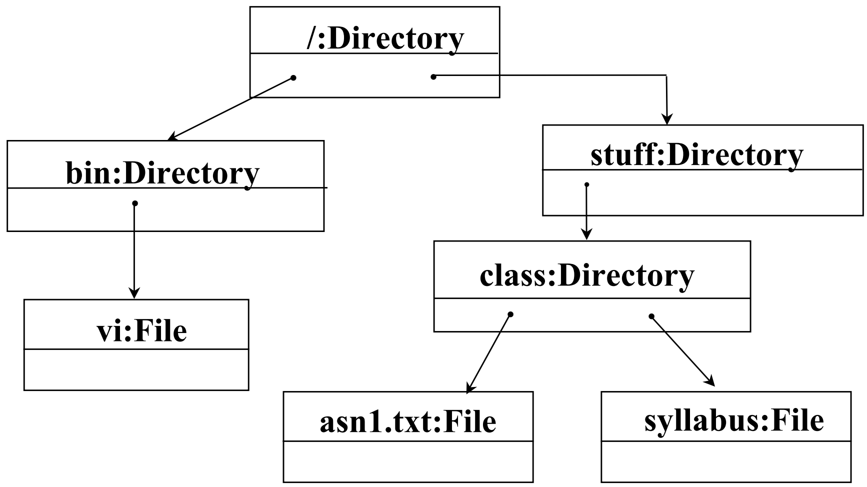
The `isContainer`() method allows a client of a collection of `Fileable` objects to test if a given `Fileable` object is a container (e.g., a `Directory`) so that it knows whether it can add, delete, or list any contained objects).



Structurally, this allows the same nested hierarchies of `Fileable` objects as the previous diagram. In this version, the "container-ish" methods (`add()`, `delete()`, and `list()`)have been placed on the most generic class, i.e., `Fileable`, rather than on the class `Directory`. This provides a uniform means of dealing with hierarchies of `Fileable` objects. A consequence is that `File` objects must ignore these methods or throw an exception.

This second model is actually an example of a pattern called the Composite Pattern. The model above is simpler than that on the previous page, in that a client of the filing system does not to continually test whether an object in the system is a `File` or a `Directory`.

An Instance Diagram for the File System



This instance diagram shows one (of an unlimited number) of the possible sets of `Fileable` objects that can be created that satisfy the class diagram on the previous page.

In this example, there is a top-level (root) directory named “/” (slash), which has two subdirectories named `bin` and `stuff`. We are used to forming names of directories and files by concatenating the names of the instances along the path (for instance, the full name of the file `syllabus` is `/stuff/class/syllabus`).

This diagram is a typical *instance hierarchy*, which is defined by an aggregation relationship, such as that between class `Directory` and instances of class `Fileable`, and can have an unlimited number of possible instances. This is in contrast with the concept of an *inheritance hierarchy*, which refers to the "is-a" relationships that exist between derived classes and their base class(es). For instance, in the filing system, the inheritance hierarchy consists of the base class `Fileable` and the derived classes `File` and `Directory` and is fixed. An instance hierarchy is defined by an aggregation relationship, such as that between class `Directory` and instances of class `Fileable`, and can have an unlimited number of possible instances.

A Few Words on Reuse

- Object technology supports reuse through interoperable components; for instance, reuse of third-party foundation class libraries is now routine, and many organizations are building reusable business objects such as Customer, Account, etc
- Design patterns enable the reuse of conceptual designs for collaborating components; this is the thrust of the course we are just beginning

Object technology is touted as a mechanism for finally bringing widespread reusable components to software development organizations.

When we talk about reusable design patterns, we are dealing at a conceptual level. In most cases there will not be a class library that embodies the design pattern. It will be up to you to understand the solution proposed in the design pattern and implement it within your own context.

Two Principles for Reuse

- Program to an interface, not to an implementation
  - In C++ and Java this means using only the methods declared as public, restricting the use of data members to read/write access methods, and limiting the use of protected methods and (in C++) 'friend' functionality
- Favor composition over inheritance
  - This means to look for opportunities to supply an object at run-time with the desired functionality rather than constraining the choice of object to use to compile-time inheritance

In C++, supplying an object at run-time usually requires using a pointer to refer to the assigned object. In Java, a reference to an object is supplied.

Clients should maintain a deliberate lack of awareness of the implementation details behind an interface. When the implementation changes, a client can continue to use the same interface, and polymorphic behavior also provides a binding to the correct subclass implementation.

Composition means reusing other objects by combining them(via associations, in the GoF sense) into collaborative sets of objects that deliver the desired functionality. Composition enables run-time choice of composable objects, which offers more flexibility than inheritance, which imposes compile-time restrictions.

Note that the second bullet above does not mean that you should not use inheritance, but that you can increase the flexibility of your system through composition.

Reuse Through Inheritance

- Is-A relationships define interface inheritance. A subclass provides the same interface as the parent class
- When you reuse a class by inheriting from it, you also often gain some implementation inheritance, in that you can “see into” the parent’s implementation, for instance, into its public and protected methods, which you can override
- Reuse through inheritance requires declaring the relationship at compile-time

In the early days (a mere 10-15 years ago!) of object design, inheritance was the primary means of providing reuse. As time has gone on, there has been a recognition that there are other means of combining existing objects to attain reuse. For instance, many people advocate more use of ‘composition’ of objects to obtain reuse.   Note that ‘composition’ in UML is different from the meaning in the reuse context, where composition means any association.

Reuse Through Composition

- Composition, by contrast with inheritance, allows the binding of the reused objects at runtime, and therefore increases the system flexibility

Note that 'composition' here is used in the general sense defined in the Gang of Four.

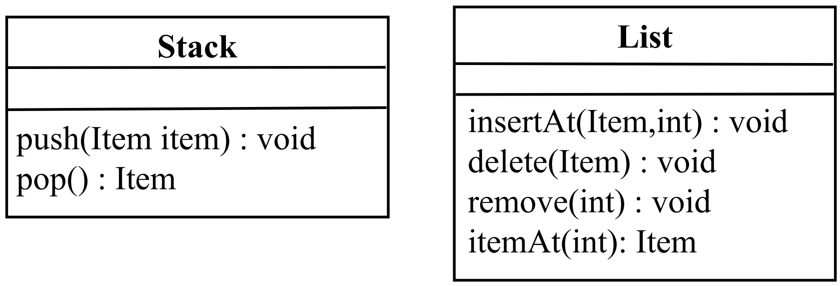
The Two Types of Patterns

- The GoF have introduced two terms to highlight the difference between reuse through inheritance and reuse through composition
- “Class-based patterns” use Inheritance relationships (e.g., the Factory Method Pattern,)
- “Object-based patterns” use Composition relationships (e.g., the Class Factory Pattern,)
- We will find examples of both types of patterns within each of the 3 categories defined previously

The terms “class-based” and “object-based” were introduced by the GoF to further differentiate design patterns. The class-based patterns tend to be a little less flexible than the object-based patterns. As a designer, you have to decide how much flexibility is required and choose the implementation accordingly.

Example: Inheritance vs. Composition

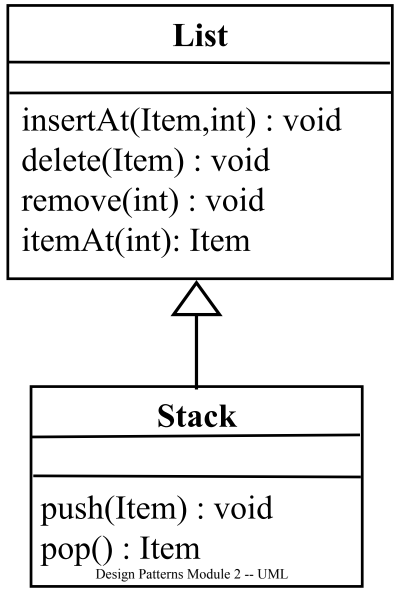
- Suppose you’re trying to define a `Stack` class, and want to implement the `Stack` in terms of a `List` container class



The diagram above shows the `Stack` and `List` classes. The interface for the `Stack` class consists of the two standard methods, `push()` and `pop()`. The input to `push()` is anything of type Item. The return value from `pop()` is also anything of type `Item`. Similar comments apply to the methods `insertAt()` and `delete()` in the class List. This limits the utility of the `Stack` and `List` classes to one kind of object. In C++, one can use the template facility to define `Stack` and `List` classes that allow their use on any kind of object. In Java, which has no template facility, one can approach the problem differently, as we'll show a little later.

The goal is to implement the `Stack` methods in terms of the `List` class methods for insertion, deletion, and retrieval of items in the `List`. There are two approaches to doing this: use inheritance or use composition.

Using Inheritance to Define the Stack Class



Note: this approach is not desirable, as described below

There are two flaws with using inheritance to implement the `Stack` class.

- The `Stack` interface is just a '`push()/pop()`' pair, so any other methods for accessing the `List` would be inappropriate if exposed as public methods on `Stack`

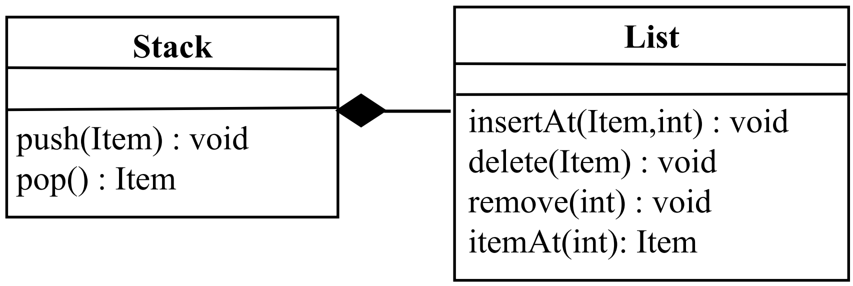
- If later on you want to change the implementation, you have to change the declaration of the `Stack` class and recompile every client that uses the `Stack` class.

A `Stack` is not a `List`. You do not want to use a `Stack` wherever a `List` could be used.

C++ supports "implementation inheritance" through private inheritance. Java does not support private inheritance.

Unfortunately, the class `java.util.Stack` is declared to extend the class `Java.util.Vector`. Hence it should be used with care, since a user of `Stack` has access to the general access mechanisms of `Vector`.

Using Composition to Define the Stack Class



It’s easy to see how to code `push()` in terms of the `insertAt()` method and `pop()` in terms of `itemAt()` and `remove()`

The composition approach is clearly superior to the use of inheritance in this case.

- The `Stack` interface is not polluted with the `List` class's methods
- If later on you want to change the implementation, you don’t have to change the declaration of the `Stack` class and recompile every client that uses the `Stack` class.

Implementing the Class `Stack` using `List`

```
public class List
{
 // details of the List implementation
 // are omitted
 public void insertAt(Object item,
                             int where) {};
 public void delete(Object item) {};
 public void remove(int where) {};
 public Object itemAt(int where) {}

// Internal storage for the list
// is private
}// end List
```


We start with the definition above, then cover the implementation of `Stack` on the next slide. We omit the details of the `List` method implementation.

This specification of `List` is very much like the Java class `java.lang.Vector`.

As we will in many examples, we have not declared any exceptions; for instance, `remove(int where)` would likely throw an `IndexOutOfRange` exception when the parameter is negative, 0, or larger than the number of elements in the list.

Implementing the `Stack` Class

```
public class Stack {
  public void push(Object item) {
    items.insertAt(item, 0);
  }
  public Object pop() {
    Object item = items.itemAt(0);
    items.remove(0);
    return item;
  }

  private List items;
}// end Stack
```


`push()` and `pop()` are the two methods you'd expect to see on a stack.

A more realistic version of `push()` would declare an exception such as `StackFull`, and `pop()` would declare an exception, such as `StackEmpty`.

These implementations are straightforward. Note that we have omitted any exception handling (for instance in the case of pop(), the stack could be empty, in which case `itemAt(0)` would raise an exception. `pop()` could either handle the exception or raise its own exception to its caller.

This generic `Stack` deals in the most general Java class, namely, `java.lang.Object`. This means a user has to exercise care when trying to ensure that only objects of a certain type can be pushed onto or popped off the stack.

Implementing Type-specific `Stack` Classes

```
public class ItemStack {

  private Stack internalStack;

  public ItemStack() {
    internalStack = new Stack();
  }
  public void push(Item thingToPush){
    internalStack.push(thingToPush);
  }
  public Item pop(){
   Object value = internalStack.pop();
   return (Item)value;
  }
} // end ItemStack
```


Suppose we want to create a specific `Stack` to hold objects of type `Item`. We can use the generic `Stack` and ensure that we push only objects of type `Item`, and when we pop the `Stack`, we can check that we've really got something of type `Item`. This puts the burden on the user of the generic `Stack` to do run-time type checking.

The alternative is shown above, where we show how to reuse the generic `Stack` through composition to provide a type-safe version of `Stack` for objects of type `Item`. Note that we would have to propagate any exceptions thrown by the methods in the generic `Stack` .

It is a nuisance to have a separate class for each kind of `Stack` we want to create.