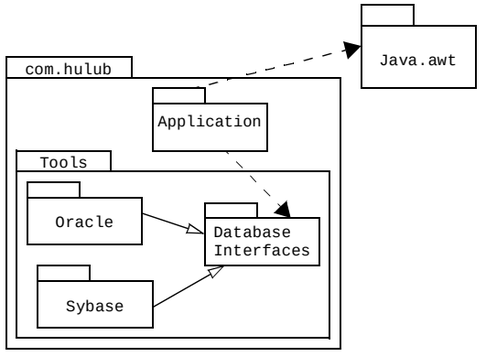
UML REFERENCE CARD

Available from <http://www.holub.com>.

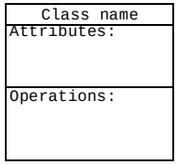
Static Model Diagrams

Packages



- C++ **namespace**.
- Group together functionally-similar classes.
- Derived classes need not be in the same package.
- Packages can nest. Outer packages are sometimes called **domains**. (In the diagram, “Tools” is arguably an outer package, not a domain).
- Package name is part of the class name (e.g. given the class *fred* in the flintstone package, the **fully-qualified** class name is *flintstone.fred*).
- Generally needed when entire static-model won’t fit on one sheet.

Classes (Box contains three compartments)

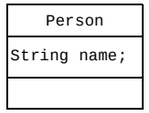


1. The name compartment (required) contains the class name and other documentation-related information:

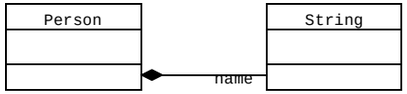
   E.g.:

   Some_class «abstract»
   { author: George Jetson
   modified: 10/6/2999
   checked_out: y
   }

   - Guillemets identify stereotypes. E.g.: «static», «abstract» «JavaBean». Can use graphic instead of word.
   - Access privileges (see below) can precede name.
   - Inner (nested) classes identify outer class as prefix of class name: (**Outer.Inner** or **Outer::Inner**).
2. The *attributes compartment* (optional):
   - *During Analysis*: identify the attributes (i.e. defining characteristics) of the object.
   - *During Design*: identify a relationship to a stock class.
     This:

     is a more compact (and less informative) version of this:

     Everything, here, is private. Always. Period.
3. The *operations compartment* (optional) contains method definitions. Use implementation-language syntax, except for **access privileges**:

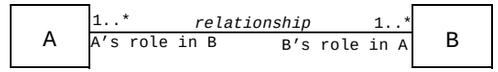
+	public
#	protected
-	private
~	package (my extension to UML)[1]

   - Abstract operations (C++ virtual, Java non-final) indicated by *italics* (or underline).
   - **Boldface** operation names are easier to read.

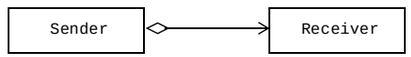
If attributes and operations are both omitted, a more complete definition is assumed to be on another sheet.

[1] *Java, unfortunately, defaults to “package” access when no modifier is present. In my “flavor” of UML, a missing access privilege means “public”.*

Associations (relationships between classes)



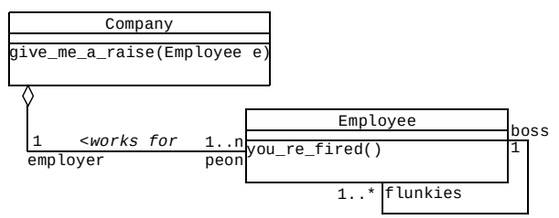
- Associated classes are connected by lines.
- The relationship is identified, if necessary, with a < or > to indicate direction (or use solid arrowheads).
- The role that a class plays in the relationship is identified on that class's side of the line.
- Stereotypes (like «friend») are appropriate.
- Unidirectional message flow can be indicated by an arrow (but is implicit in situations where there is only one role):



- Cardinality:

1	Usually omitted if 1:1
n	Unknown at compile time, but bound.
0..1	(1..2 1..n)
1..*	1 or more
*	0 or more

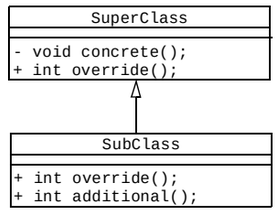
- Example:



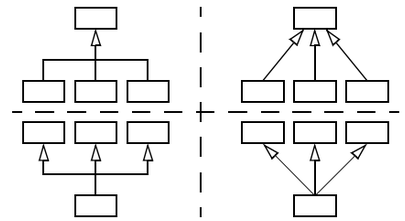

```
class Company
{
  private Employee[] peon = new Employee[n];
  public void give_me_a_raise( Employee e ) { ... }
}

class Employee
{
   private Company  employer;
   private Employee boss;
   private Vector   flunkies = new Vector();
   public  void     you_re_fired() { ... }
}
```

(A Java Vector is a variable-length array. In this case it will hold Employee objects)

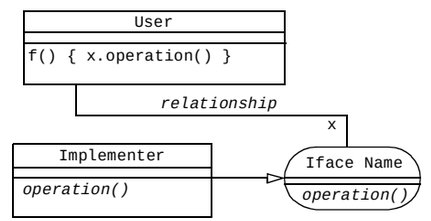
Implementation Inheritance



Outline arrows identify derivation relationships: extends, implements, is-a, has-properties-of, etc. Variations include:

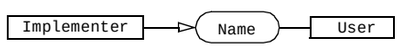


Interface Inheritance

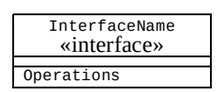


In C++, an interface is a class containing nothing but pure virtual methods. Java supports them directly (c.f. “abstract class,” which can contain method and field definitions in addition to the abstract declarations.)

My extension to UML: rounded corners identify interfaces. If the full interface specification is in some other diagram, I use:

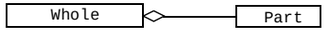


Strict UML uses the **«interface»** stereotype in the name compartment of a standard class box:



Interfaces contain no attributes, so the attribute compartment is always empty.

Aggregation (comprises)



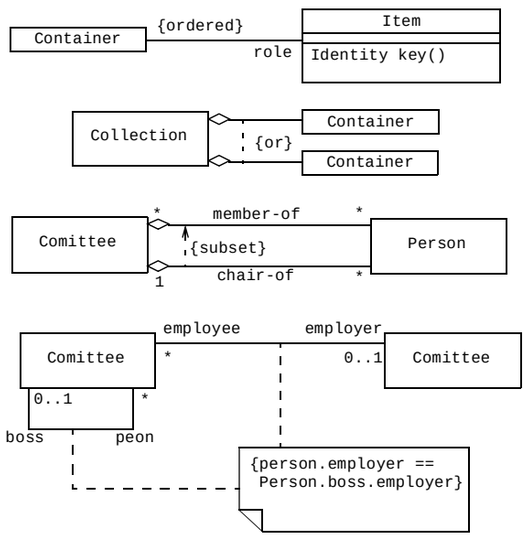
- Destroying the “whole” does not destroy the parts.
- Cardinality is allowed.

Composition (has) relationship

- The parts are destroyed along with the whole.
- Doesn’t really exist in Java.
- In C++:

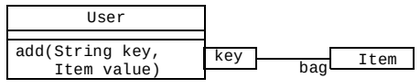
```
class Container
{
     Obj item1;
     Obj *item2;
   public:
     Whole() { item2 = new Obj; }
     ~Whole(){ delete item2;    }
};
```

Constraint



- A constrained relationship requires some rule to be applied (e.g. {ordered}). Often combined with aggregation, composition, etc.
- In the case of {or}, only one of the indicated relationships will exist at any given moment (a C++ union, or reference to a base class).
- {subset} does the obvious.
- In official UML, put arbitrary constraints that affect more than one relationship in a “comment” box, as shown. I usually leave out the box.

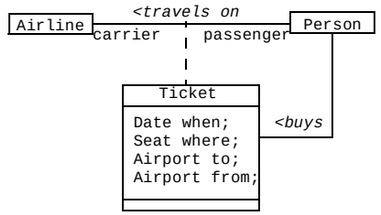
Qualified Association



- Hash tables, associative arrays, etc.

```
class User
{
   // A Hashtable is an associative array, indexed
   // by some key and containing some value.
   private Hashtable bag = new HashTable();

   private void add(String key, Item value) {
      bag.put(key, value);
   }
}
```

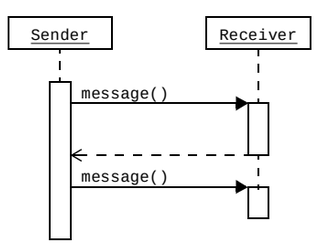
Association Class



- Use when a class is required to define a relationship.
- Somewhere, an additional relationship is required to show ownership. (The one between person and Ticket in the current example).

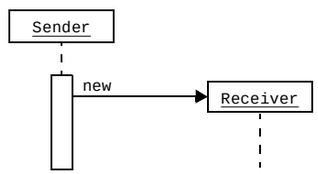
Dynamic-Model (Sequence) Diagrams

Objects and Messages (new style)

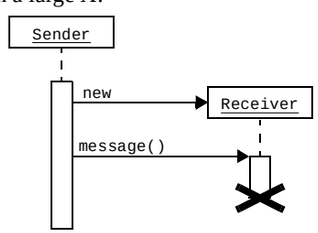


- Top boxes represent objects, not classes. You may optionally add “`:class`” to the name if desired.
- Vertical lines represent the objects “life line”, or existence.
- Broken lifeline indicates the object is inactive, a rectangle indicates the object is active.
- ⟶ represent messages being sent.
- ⟵ (optional if synchronous) represent method return. (May label arrow with name/type of returned object).
- Sending object’s class must have:
  1. An association of some sort with the receiving objects class.
  2. The receiver-side class’s “role” must be the same as the name of the receiving object.

Object Creation



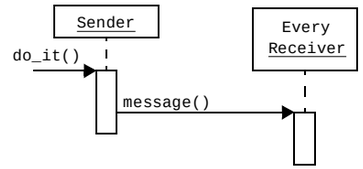
- The new instance appears at end of creation message arrow.
- Destruction is accomplished by terminating the lifeline with a large X:



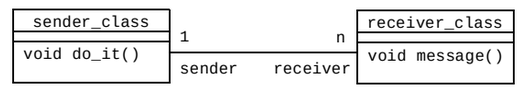
Conditions

- Message sent only if conditional expression is true.
- The `cond_expr` is typically expressed in the implementation language.

Loops (extension to UML)



- Don’t think loops, think what the loop is accomplishing.
- Typically, you need to send some set of messages to every element in some collection. Do this with **every**.
- You can get more elaborate (every receiver where x<y)
- The diagram above comes from:



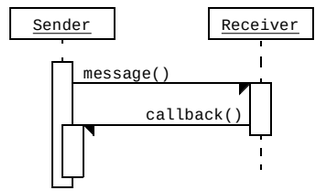
and maps to the following code:

```
class sender_class
{
   receiver_class receiver[n];
   public do_it() {
      for(int i = 0; i < n; ++i)
         receiver[i].message();
   }
}
```

Arrow Styles for Messages

Symbol	Type	Description
→	Simple	Don’t care. Usually read as the same as synchronous.
—x→	Synchronous	Sender blocks until return.
—→	Asynchronous	Handler returns immediately and both sender and receiver work simultaneously.

Asynchronous Callbacks



- Callback occurs while Sender is potentially executing something else.