

Allen Holub's UML Quick Reference

Version 2.1.2 (2005/08/10)

<http://www.holub.com/goodies/uml/>

© 2005, Allen I. Holub (<http://www.holub.com>). All rights reserved.

You may link to this page, but please do not "mirror" it (make a local copy). I revise this reference periodically, and local copies will become obsolete. This document may be reproduced and distributed freely, provided that the entire document is distributed without modification (including the copyright notice, my url, and this paragraph).

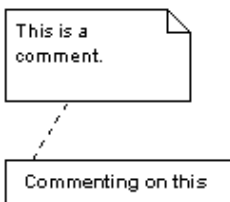
This reference covers the notation described in the OMG *UML version 2.0 Superstructure* document found at <http://www.omg.org/cgi-bin/doc?ptc/2003-08-02>. You can also find other UML information on the OMG UML site: <http://www.uml.org>.

UML Training by Allen Holub

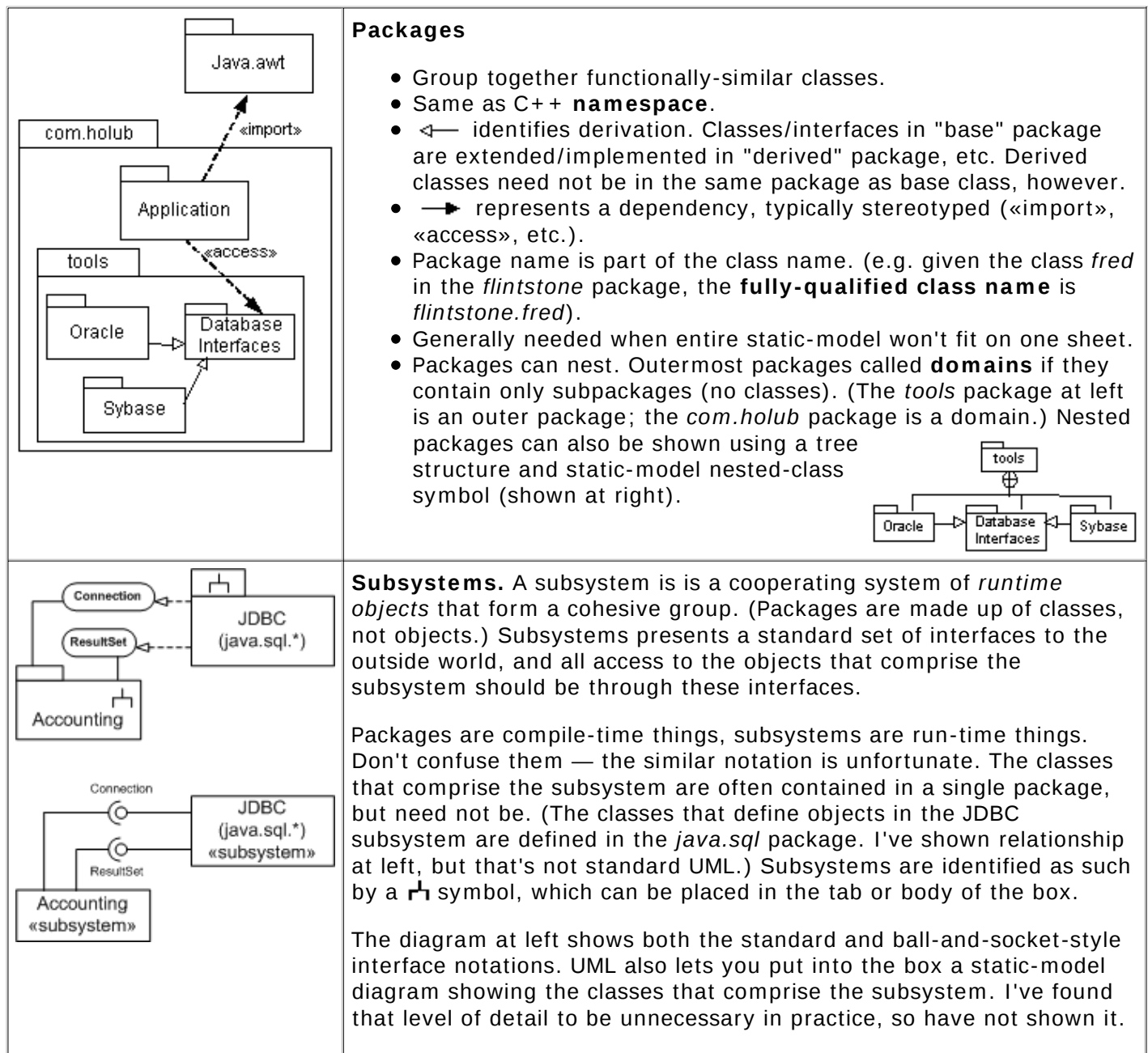
The best in-house Object-Oriented-Design and Java training available.

www.holub.com

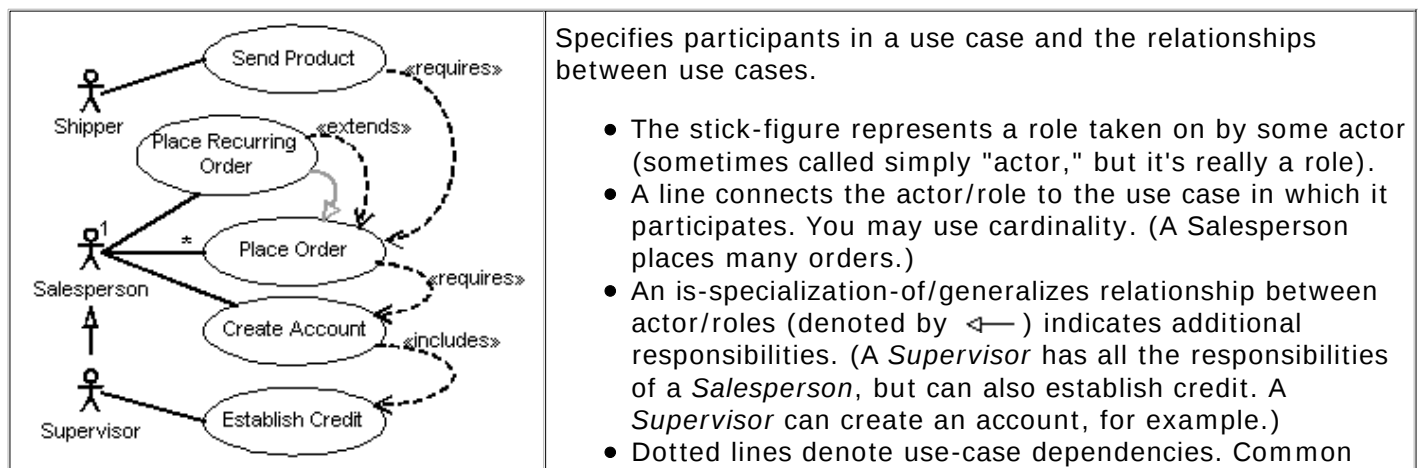
Miscellany

 A diagram showing a comment box with the text "This is a comment." and a dashed line pointing to a box labeled "Commenting on this".	Comments. Any kind of information that isn't easily representable in UML, including comments, implementation-level code, etc. Also used for a long constraint. This symbol is used in all of the UML diagrams.
--	---

Organizational Diagrams



Use-Case Diagram



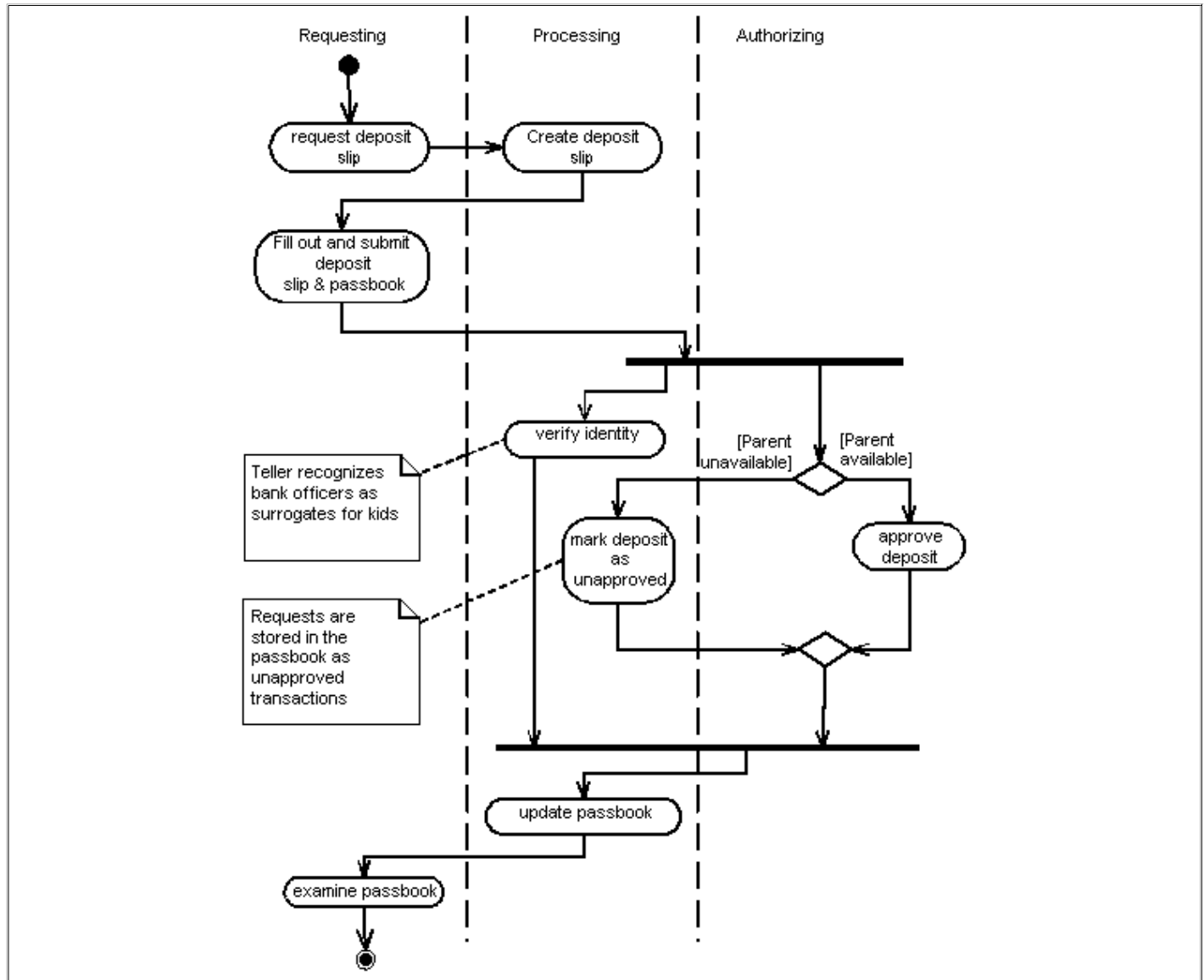
	dependencies are: «equivalent» Equivalent use cases have identical activities and identical flow, but end users think of them as different. ("Deposit" and "Withdrawal" might have identical activities, though the objects involved might be different.) «extends» When <i>extension</i> extends <i>base</i>, all the activities of the <i>base</i> use case are also performed in the <i>extension</i> use case, but the <i>extension</i> use case adds additional activities to—or slightly modifies existing activities of—the <i>base</i> use case. (To place a recurring order, you must perform all the activities of placing an order plus set up the recurrence.) If a set of activities occur in several use cases, it's reasonable to "normalize" these common activities out into a <i>base</i> use case, and then extend it as necessary. <i>Holub Extension:</i> This relationship is really a form of derivation, so I use the derivation arrow () instead of a dashed line. As in a class diagram, the arrow points from the <i>extension</i> to the <i>base</i> use case. «includes» A subcase. If <i>case</i> includes <i>subcase</i>, then the activities of <i>subcase</i> are performed one or more times in the course of performing <i>case</i>. (An "Authenticate" subcase may be included in several larger use cases, for example.) The subcase is usually represented in the using use case as a single box marked with the subcase name and the stereotype «use case». «requires» «follows» If <i>follower</i> requires <i>leader</i>, then <i>leader</i> must be completed before you can execute the <i>follower</i> use case. (You must create an account before you can place an order.) «resembles» Two use cases are very similar, but do have different activities. Actors/roles are mostly uninteresting to programmers. The dependencies are valuable in determining which use case to implement first. (I often implement the use cases that have the most incoming arrows first, since other use cases depend on them.)
--	---

Activity and State Diagrams

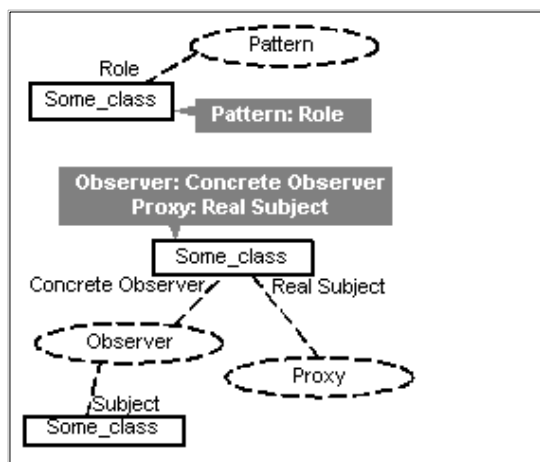
State diagrams share many notational elements with activity diagrams. The main difference is that state diagrams "decorate" the transitions (directed lines between states) to indicate the method call or condition that caused the transition.

	Starting and Stopping. The solid circle indicates the beginning of the sequence of activities. The circle with an X represents an end of a "flow" but not the end of the entire use case. In other words, some subtask completes, but the entire use case is not yet complete. The "target" indicates that the entire use case is complete.
	Subcase (Sub-Activity). The "rake" symbol indicates that the "activity" is complex enough to merit its own activity diagram. In use-case analysis, this is a "subcase"---a stand-alone activity that occurs in more than one use case but is not large enough to be a use case in its own right.
	Synchronization (Fork / Join). Used either when several activities can go on in parallel or when the order in which a set of activities execute is immaterial. The heavy bar at the top is a <i>fork</i>. After the fork, all activities can (but are not required to) go on in parallel. Progress cannot continue past the bar on the bottom (the <i>join</i>) until all the activities that feed into the join complete. You can label the join with a constraint (e.g. {joinspec= (A and B) or C}) to specify the condition that allows progress to continue. If there's no constraint, AND is assumed.
	Guards (tests). This path is used only if the text in the brackets is true.
	Decision (Branch / Merge). A decision activity, the guard labels the decision that was made. The diamond with outgoing arrows (the <i>branch</i>) specifies an OR operation, with a condition imposed by the guard. The diamond with incoming arrows (a <i>merge</i>) simply provides an end to the OR operation. A merge can occur without an associated branch if the diagram has multiple start states.

Example Activity Diagram

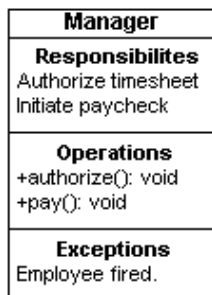
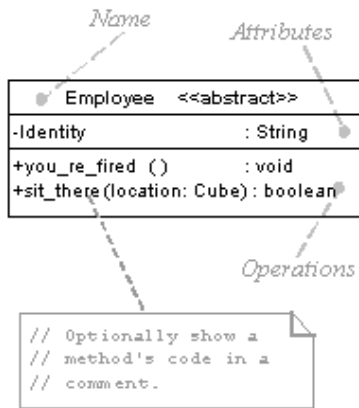


Static-Model (Class) Diagram



Design Pattern (Collaboration)

- The dashed circle is UML, ver. 1.5. The grey box is Erich Gamma's notation, as presented in John Vlissides' book *Pattern Hatching* (Reading: Addison Wesley, 1998). Use one or the other.
- A single class can have different roles with respect to several patterns. In the bottom example, the class serves as both the "Concrete Observer" in the "Observer" pattern and also the "Real Subject" in the "Proxy" pattern. The UML notation can identify all participating classes if they happen to be in physical proximity.



Classes. Standard representation contains three *compartments*:

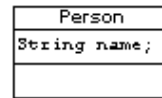
1. The *name compartment* (required) contains the class name and other documentation-related information:
E.g.:

Some_class «abstract»

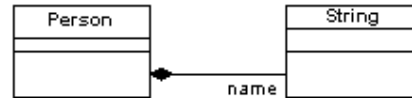
```
{ author:      George Jetson
  modified:    10/6/2999
  checked_out: y
}
```

- Guillemets identify **stereotypes**. E.g.: «**utility**», «**abstract**» «**interface**».
 - Can use a graphic instead of word. («**interface**» often represented as small circle)
 - Access privileges (see below) can precede name
 - Use italics for abstract-class and interface names.
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 except constant values must be private. Always. Period.

3. The *operations compartment* (optional) contains method definitions:

message_name(arguments): return_type

Resist the temptation to use implementation-language syntax.

Visibility (**access privileges**) indicated as follows:¹

-
- + public
 - # protected
 - ~ package²
 - private
 - implementation visibility (inaccessible to other objects)²
 - (+) forced public. Override of an interface method that should be treated as private, even if it's declared public.²
-

UML 2.0 permits C++-style grouping:
public

```

a(): int
b(): void
private
c(): void

```

Properties (new in UML 2.0.):

/	Derived attribute. Synthesized at runtime. Combine with access. (e.g. -height , -width , / +area)
{property}	Standard properties are: {readOnly}, {union}, {subsets property-name}, {redefines property-name}, {ordered}, {bag}, {seq} (or {sequence}), {composite}.
example: / +area: integer {readOnly}	

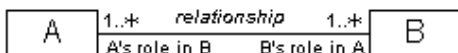
abstract operations indicated by *italics* (or underline).

If attributes and operations are both omitted (yielding a box with a class name in it), a more-complete definition is assumed to be on another sheet.

Introduce nonstandard compartments simply by naming them, as is shown in the bottom example at left.

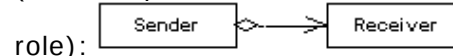
¹ Java, unfortunately, defaults to "package" access when no modifier is present. UML does not support the notion of a default access. UML has no notion of "implementation visibility" (accessible only within an object — other objects of the same class cannot access it).

² These are Allen Holub's personal extensions. The ~ was incorporated into the UML standard with version 1.5. The other's are not standard UML.



Associations (relationships between classes).

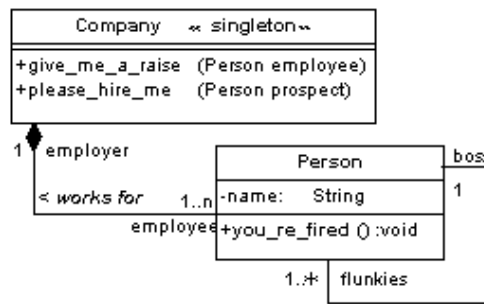
- Associated classes are connected by lines.
- The **relationship** is identified, if necessary, with a < or > to indicate direction (or use solid arrowheads).
- The role that a class plays in the relationship is identified on that class's side of the line.
- Stereotypes (like «**friend**») are appropriate.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



- 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:



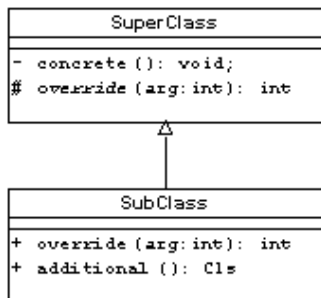
```

class Company
{
    private Person[] employee=new Person[n];
    public void give_me_a_raise(
        Person employee)
    {
        /*...*/
    }
    public void hire_me( Person prospect )
    {
        /*...*/
    }
}
  
```

```

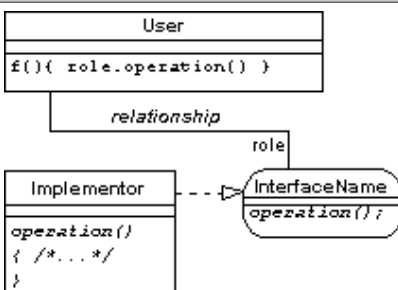
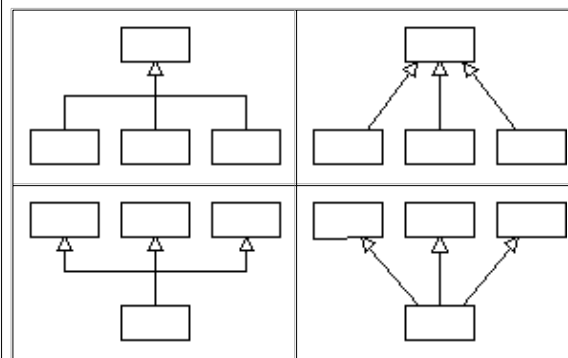
class Person
{
    private String name;
    private Company employer;
    private Person boss;
    private Vector flunkies=new Vector();
    public void you_re_fired(){...}
}
  
```

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



Implementation Inheritance (Generalize / Specialize)

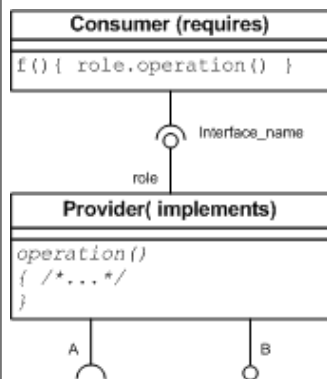
◁ identifies derivation The derived class *is* the base class, but with additional (or modified) properties. Derived (sub) class is a specialization of (extends) the base (super) class. Variations include:



Interface Inheritance (Specifies / Refines).

A contract that specifies a set of methods that must be implemented by the derived class. 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.)

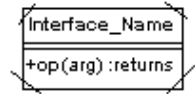
Interfaces contain no attributes, so the "attributes" compartment is always empty.



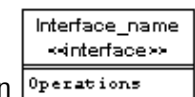
The "inheritance" relationship line is dashed if the base class is an interface.

The "ball and socket" notation at left is new in UML 2.0. Classes that consume (require) an interface display a "socket" labeled with the interface name (A at left). Classes that provide (implement) an interface display a "ball" labeled with the interface name (B at left). Combining the two is a compact way to say that the Consumer talks to the provider via the named interface.

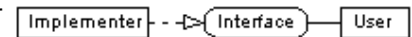
My UML extension: Rounded corners identify interfaces. Since rounded corners are often difficult to draw by hand, I sometimes use the version at right for hand-drawn diagrams.



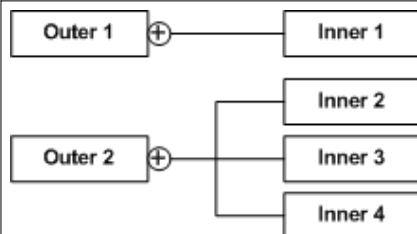
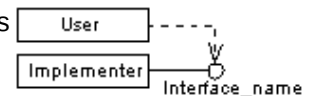
Strict UML uses the «**interface**» stereotype in the name compartment of a standard class box. A small circle in a corner of the compartment can also indicate an interface.



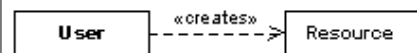
If the full interface specification is in some other diagram, I use the "ball" notation or



Microsoft-style "pin" notation (at right) is obsolete as of UML 2.0. Don't use it.



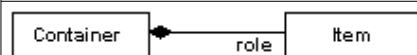
Nesting, Inner Class.. Identifies nesting (containment) relationships in all diagrams. In a class diagram: an "inner" class whose definition is nested within the another class definition. Typically puts the inner class in the name space of the outer class.



Dependency. *User* uses *Resource*, but *Resource* is not a member of (field in) the *User* class. If *Resource* is modified, some method of *User* might need to be modified. *Resource* is typically a local variable or argument of some method in *User*. The line is typically stereotyped (e.g. «creates» «modifies»)



Aggregation (comprises) relationship relationship.¹ Destroying the "whole" does not destroy the parts.



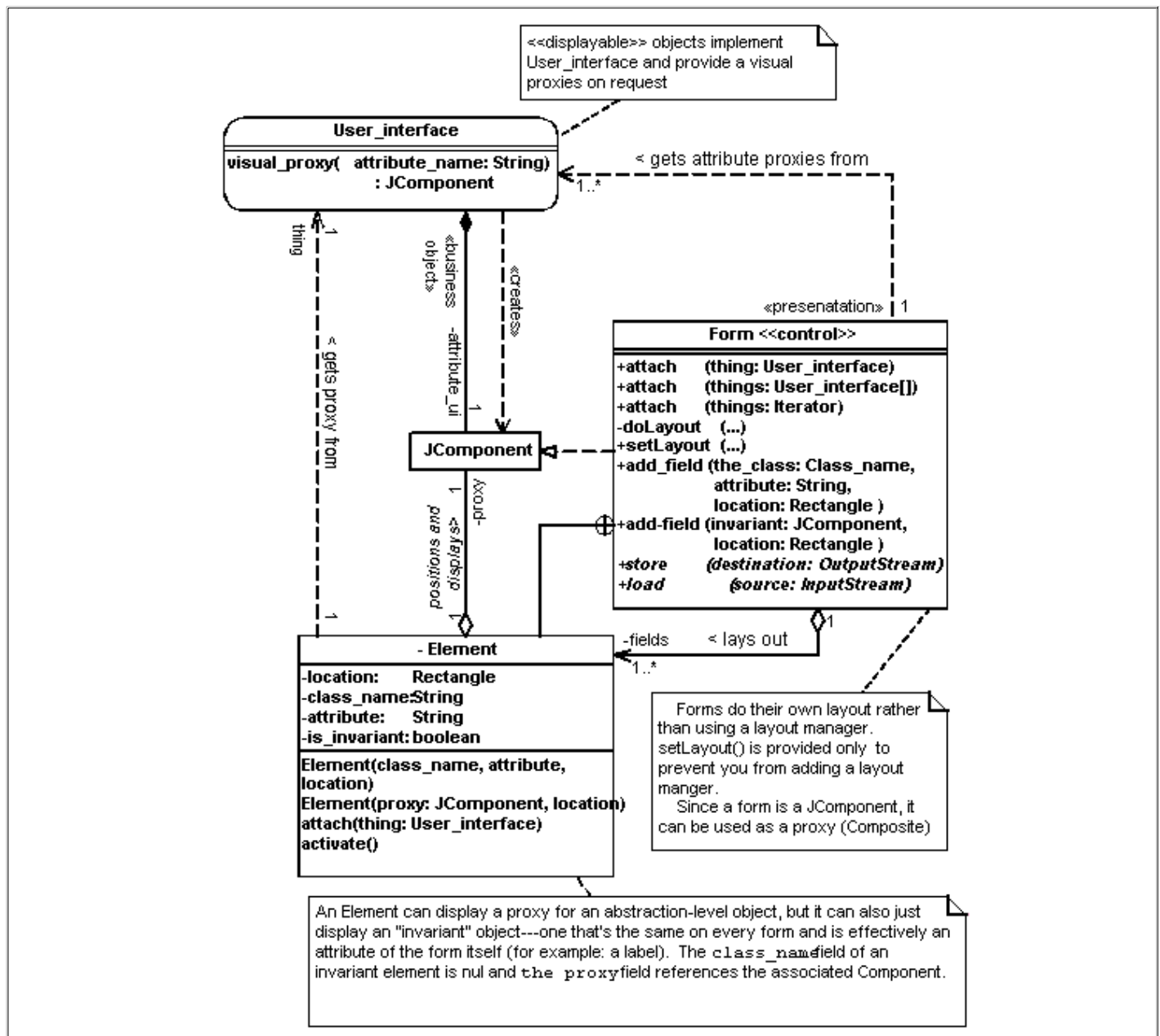
Composition (has) relationship.¹ The parts are destroyed along with the "whole." Doesn't really exist in Java. In C++:

```

class Container
{
    Item item1;    // both of these are
    Item *item2;   // "composition"
public:
    Container() { item2 = new Item; }
    ~Container() { delete item2; }
}
  
```

	Navigability Messages flow in direction of arrow (only). An unmarked line is "unspecified" navigability. An X indicates non-navigable (Uml 2.0). Typically, if a role is specified, then navigability in the direction of that role is implicit. If an object doesn't have a role in some relationship, then there's no way to send messages to it, so non-navigability is implicit.
	Constraint A constrained relationship requires some rule to be applied. (e.g. <i>{ordered}</i>) Often combined with aggregation, composition, etc.
	Complex Constraint Comments • In the case of the or, only one of the indicated relationships will exist at any given moment (a C++ union). • 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-table, associative array, "dictionary"). <pre> class User { // A Hashtable is an associative array, // indexed by some key and containing // some value; in this case, contains // Item objects, indexed by UID. private Hashtable bag = new Hashtable(); private void add(UID key, Item value) { bag.put(key, value); } } </pre>
	Association Class • Use when a class is required to define a relationship. • If this class appears only as an association class (an class-to-class association like the one between Person and Ticket doesn't exist), objects of the association class must be passed as arguments to every message.

Example Class Diagram



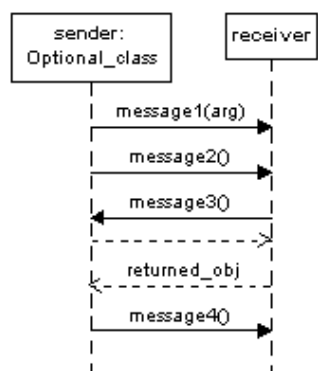
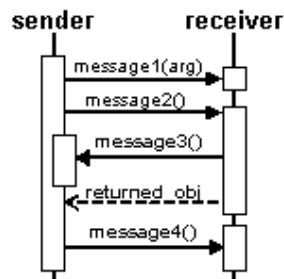
Dynamic-Model Diagrams

Dynamic model diagrams show how runtime objects act out a use case by sending messages to each other.

There are two sorts of dynamic-model diagrams: Sequence Diagrams and Collaboration Diagrams. The two forms present identical information in different way. Which one you use is largely a matter of taste.

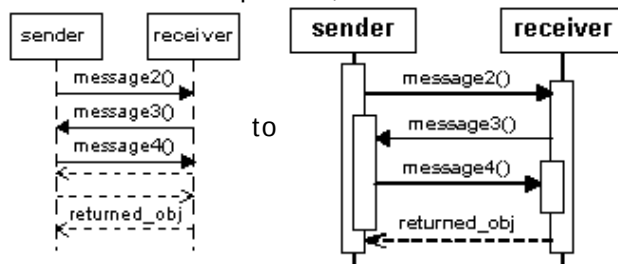
Sequence Diagram

Without Activations:	Objects and Messages - Vertical lines represent objects, not classes. - May optionally add a ":class" to the box if it makes the diagram more readable. → represents synchronous message. (message handler
-----------------------------	--

**With Activations:**

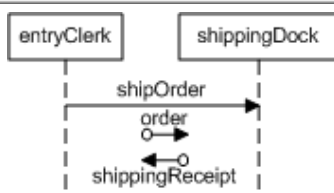
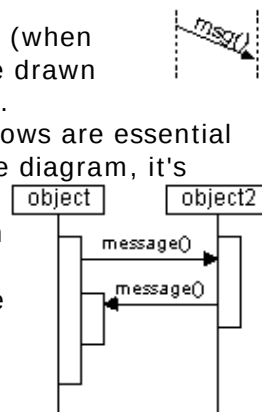
doesn't return until done).

- $\leftarrow$ represents return. Label arrow with the name [and optional :type] of the returned object. (Example at left translates to: `returned_obj=receiver.message2();`)
- Sending object's class must have:
 1. A association of some sort with receiving-object's class.
 2. The receiver-side class's "role" must be the same as the name of the receiving object.
- Activations are optional, but much easier to read. Compare:

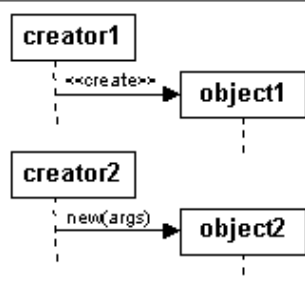


Return implied by bottom of box when activations present. A $\leftarrow$ is unnecessary on methods that don't return values. Explicit returns, when shown, are unambiguous because they emit from the activation for the message that returns the value.

- Messages that take a long time to arrive (when sent over a network for example) can be drawn with a diagonal line, as is shown at right.
- When activations are missing, return arrows are essential for disambiguating control flow. In above diagram, it's unclear whether message2 or message4 returns returned_obj when unlabeled returns are omitted.
- When drawing by hand, I use a solid line and put the activation boxes at the side of the line, as is shown at right.

**Data "Tadpoles"**

Data "tadpoles" are often more readable than return arrows or message arguments. At left, the entryClerk object sends the shippingDock an object called order. The shippingDock returns the shippingReceipt object.

**Object Creation**

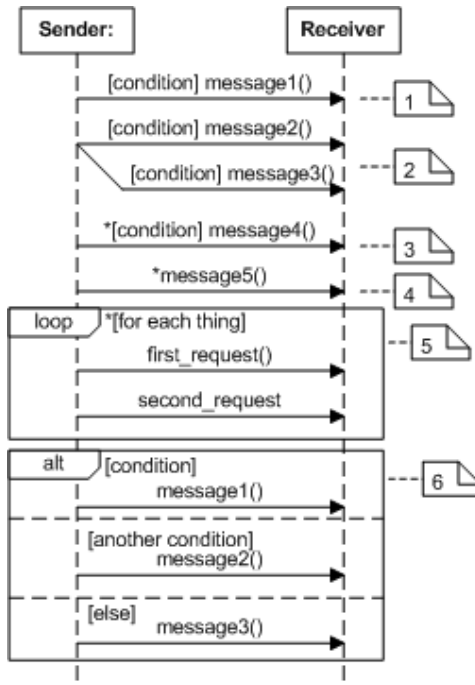
- Name box appears at point of creation.
- $\ll$ creates form for automatic creation. In C++: An object (not a reference to one) is declared as a field in a class. The closest Java equivalent is an object created by instance-variable initialization:

```

class X
{ private Cls o = new Cls();
  //...
}
  
```

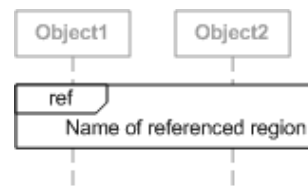
- If message shown instead of $\ll$ creates, then the message

handler creates the object. Think of `new Fred()` as `Fred.new()`. Method does not have to be `new()`.



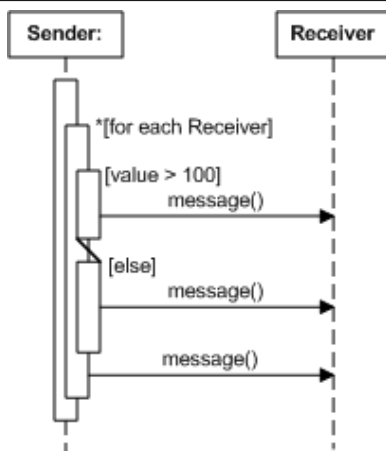
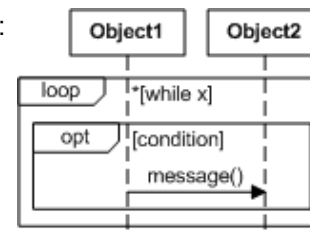
Conditions, Branches, Loops, Grouping

1. `message1()` is sent only if the condition specified in the *guard* (in brackets) is true.
2. A branch. Sender sends either `message2()` or `message3()`. Guards should be exclusive. I use «else» instead of a guard when appropriate.
3. Iteration. Sender sends `message4()` as long as the condition is true.
4. "For each." If the receiver is a collection of objects (indicated by the cardinality of the associated role in the static model), send the message to all of them.
5. UML 2.0 **Interaction Frame**. As shown, condition specifies a loop. Can also use:
 - loop A loop, executes multiple times
 - opt Optional. An "if" statement.
 - region A named region.
 - ref A reference to a named region (elsewhere in diagram or on another diagram):



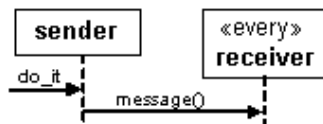
6. As shown, an if/else structure. Can use:
 - alt Execute one of the alternatives, controlled by guards
 - par Execute regions in parallel

Interaction frames can nest:



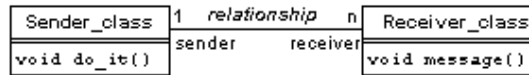
Alternative (Nonstandard) Branch/Loop Notation Use "pseudo-messages" and guards to indicate control flow.

Diagonal line indicates an "alternative" flow.



Loops, Alternative (My own extension to UML.)

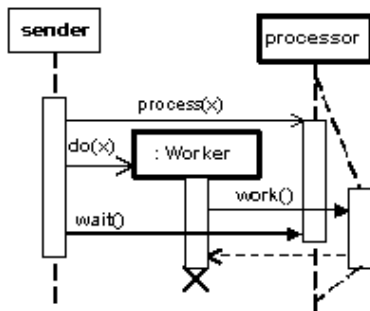
- Don't think loops, think what the loop is accomplishing.
- Typically, you need to send some set of messages to every element in some collection. Do this with **every**.
- You can get more elaborate: "every receiver where $x < y$ ".
- The diagram at left comes from this model:



and maps to the following code:

```

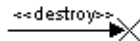
class sender_class
{
    receiver_class receiver[n];
    public do_it()
    {
        for( int i = 0; i < n; ++i )
            receiver[i].message();
    }
}
  
```



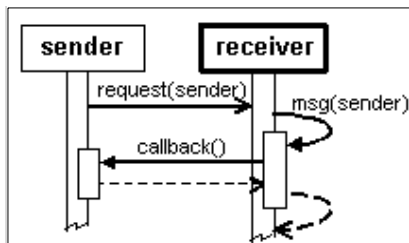
Active Objects

Active objects process messages on one or more auxiliary background threads. They are indicated by a heavyweight outline. The messages sent to an active object are typically *asynchronous*: they initiate some activity but don't wait around for the activity to complete.

- The "stick" arrowhead means *asynchronous* message — the call returns before message is fully processed. A return value from an asynchronous message can indicate that work started, but can't indicate any sort of completion status.
- The box on the lifeline means *activated* — some activity is being performed by the object, perhaps on a background thread.
- A separate lifeline [shown at left when the `work()` message activates the processor object] implies a separate thread for processing the message.
- The large X indicates that an object deletes itself when done handling message. An external kill is represented as:



At left, the `process(x)` message activates processor. The `process(x)` message is asynchronous, so the requesting method returns immediately and the processor object does the work in the background. While `process(x)` is being handled, the sender object sends a `do(x)` message, which brings an anonymous Worker object into existence. (The `do()` method is a static method of the Worker class that creates an anonymous object to handle the request.) This anonymous object does some work, sending a synchronous `work()` message to the processor object. Since the `work()` handler is synchronous, it doesn't return until the work is complete. The anonymous worker waits for `work()` to return, then deletes itself (killing any associated threads). The processor object continues to exist, waiting for something else to do.



Callbacks, Recursion

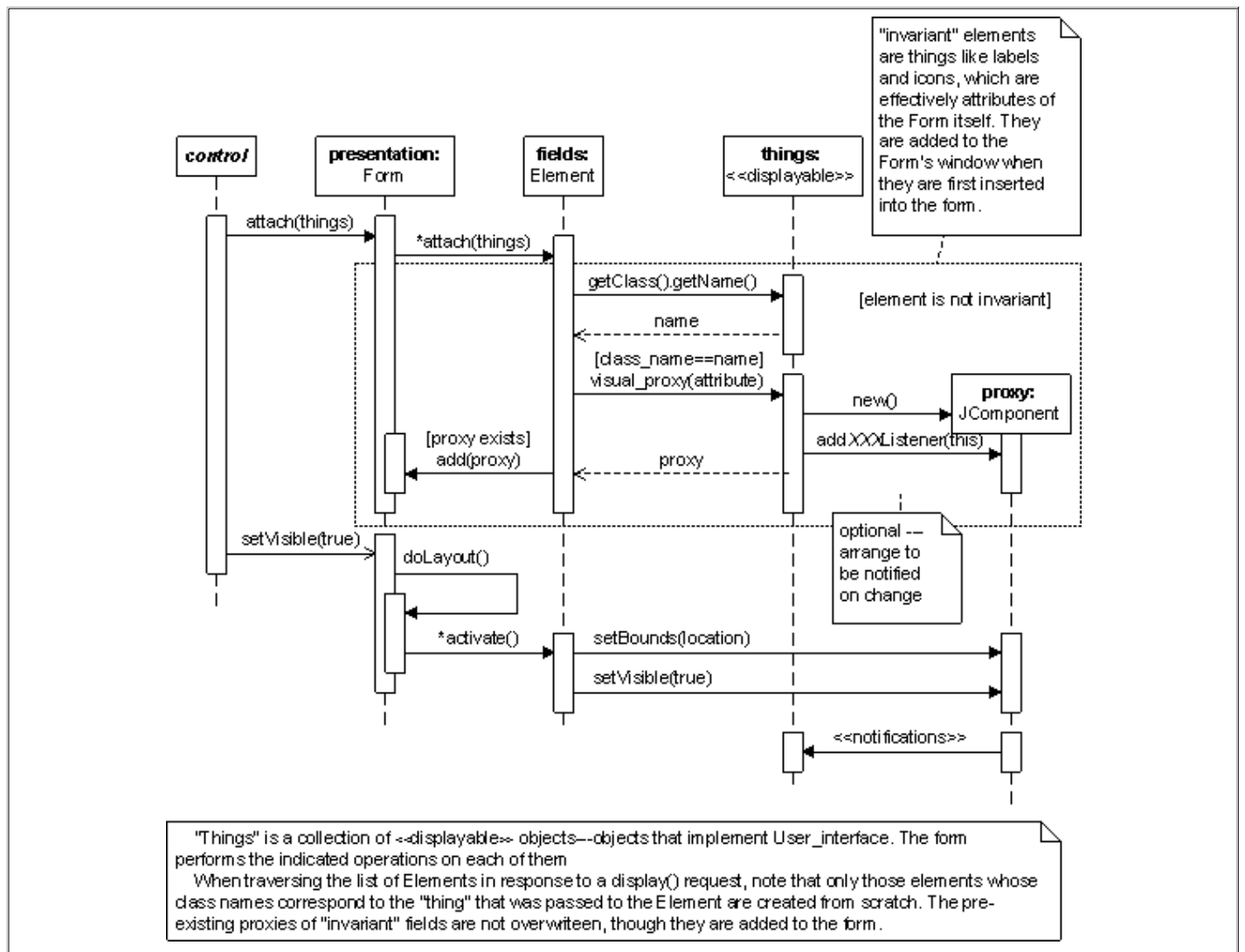
At left, the sender sends an asynchronous message to the active-object receiver for background processing, passing it the object to notify when the operation is complete (in this case, itself). The receiver calls its own `msg(...)` method to process the request, and that method issues the `callback()` call when it's done. Note that:

- The `callback()` message is running on the receiver object's message-processing thread.
- The `callback()` method is, however, a member of the sender object's class, so has access to all the fields of the sender object.
- Since the original thread (from which the original `request()` was issued) is also running, you must synchronize access to all fields shared by both `callback()` and other methods of the sender object.

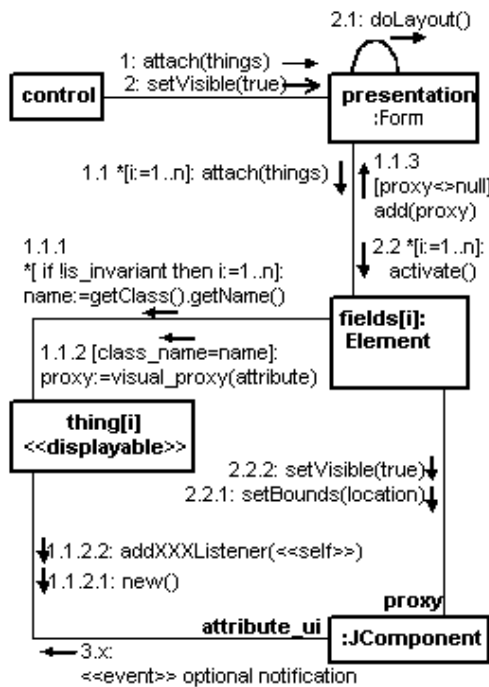
Message Arrowheads

Symbol	Message Type	Description
	Asynchronous	The handler returns immediately, but the actual work is done in the background. The sender can move on to other tasks while processing goes on.
	Synchronous	The sender waits until the handler completes (blocks). This is a normal method call in a single-threaded application.
	Asynchronous	Obsolete (UML version 1.3 or earlier.)
	Balking	The receiving object can refuse to accept the message request. This could happen if an "active" object's message queue fills, for example. Not part of "core" UML.
	Timeout	The message handler typically blocks, but will return after a predetermined amount of time, even if the work of the handler is not complete. Not part of "core" UML.

Example Sequence Diagram



Collaboration Diagram



Collaboration Diagrams are an alternative presentation of a sequence diagram. They tend to be more compact, but harder to read, than the equivalent sequence diagrams. The example at left is identical in meaning to the Sequence-Diagram example at the end of the previous section. (It represents the same objects and message flow.)

The boxes are objects. Lines connecting two boxes indicates that the objects collaborate with (send messages to) one another. Use a multiplicity indicator in the box (such as `*`) to indicate that all elements of an aggregation receive a message.

The object name typically goes inside the box, but can go outside the box when different collaborators refer to it by different names. E.g.: the `JComponent` at the lower right of the diagram is referenced by `Element` objects through a field called `proxy` it's referenced from `thing[i]` via a field named `attribute_ui`.

Use the following qualifiers on names:

<<parameter>> name	Method parameter.
<<local>> name	Local variable.
name {new}	Object created during execution
name {destroyed}	Object destroyed during execution
name {transient}	Object created during execution, used, then destroyed

Usually, the instance name (or reference through which the instance is accessed) is the same as the role the instance plays in the collaboration. When the name and role aren't identical, use **instance / role:Class**. E.g.: given `tutor/teacher:Person` and `lecturer/teacher:Person`, an object of class *Person*, used in the role of *teacher*, is called *tutor* in some portion of the code and *lecturer* elsewhere in the code.

Messages that flow from one object to another are drawn next to the line, with an arrow indicating direction. Arrowhead types have the same meaning as in sequence diagrams. The message sequence is shown via a numbering scheme. Message 1 is sent first. Messages 1.1, 1.2, etc., are sent by whatever method handles message 1. Messages 1.1.1, 1.1.2, etc., are sent by the method that handles message 1.1, and so forth. Message sequence in the current example is:

- 1.
- 1.1
- 1.1.1
- 1.1.2
- 1.1.2.1
- 1.1.2.2
- 1.1.3
- 2.
- 2.1

Nomenclature

There are three broad categories of diagrams.

Structure Diagrams

include class diagrams, deployment diagrams, etc.

Behavior Diagrams

include activity, use-case, and state diagrams.

Interaction Diagrams (are a subclass of Behavior Diagrams)

include Sequence and Collaboration diagrams.

Collaboration diagrams are called "Communication Diagrams" in UML 2.

What's Missing

A few parts of UML aren't shown here. (Some of these are useful, I just haven't gotten around to adding them yet.)

- **State-Diagram Symbols.** There are a few symbols used in state diagrams that aren't shown in the earlier Activity/State-Diagram section.
- **Deployment diagrams.** Show how large modules in the system hook up. Useful primarily for marketing presentations, executive summaries, and pointy-haired bosses.
- **Paramaterized Classes.** C++ templates/Java generics.
- **N-ary Associations.** are better done with classes. Don't use them.
- **Component Diagrams.** The only difference between a component and a subsystem is size. Component diagrams are almost identical to subsystem diagrams.
- **Activity Realization Diagram.** is an activity diagram redrawn to look more like a collaboration-diagram.

Refer to the [UML Superstructure](#) document for more details.

Footnotes

(1) Composition vs. Aggregation:

Neither "aggregation" nor "composition" really have direct analogs in Java. An "aggregate" represents a whole that comprises various parts; so, a Committee is an aggregate of its Members. A Meeting is an aggregate of an Agenda, a Room, and the Attendees. At implementation time, this is not really a containment relationship since the parts of the aggregate might be doing other things elsewhere in the program. In other words, There's no implementation-level difference between aggregation and a simple "uses" relationship (an "association" line with no diamonds on it at all). In both cases an object has references to other objects. Though there's no implementation difference, it's definitely worth capturing the relationship in the UML, both because it helps you understand the domain model better, and because there are subtle implementation issues. I might allow tighter coupling relationships in a composition than I would with a simple "uses," for example.

Composition involves even tighter coupling than aggregation, and definitely involves containment. The basic requirement is that, if a class of objects (call it a "container") is composed of other objects (call them the "elements"), then the elements will come into existence and also be destroyed as a side effect of creating or destroying the container. It would be rare for a element not to be declared as `private`. An example might be an Customer's name and address. A Customer without a name or address is a worthless thing. By the same token, when the Customer is destroyed, there's no point in keeping the name

and address around. (Compare this situation with aggregation, where destroying the Committee should not cause the members to be destroyed---they may be members of other Committees).

In terms of implementation, the elements in a composition relationship are typically created by the constructor or an initializer in a field declaration, but Java doesn't have a destructor, so there's no way to guarantee that the elements are destroyed along with the container. In C++, the element would be an object (not a reference or pointer) that's declared as a field in another object, so creation and destruction of the element would be automatic. Java has no such mechanism. It's nonetheless important to specify a containment relationship in the UML, because this relationship tells the implementation/testing folks that your intent is for the element to become garbage collectable (i.e. there should be no references to it) when the container is destroyed.

Allen I. Holub & Associates

1041 Shattuck Ave. Berkeley, CA 94707

510.528.3620

request@holub.com

©2004 Allen I. Holub (**www.holub.com**). All Rights Reserved.