Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*

Henry Lieberman

Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Mass. 02139 USA

Electronic mail (Arpaner):
Henry@AI.AI.MIT.Edu, Henry@MIT-AI

Abstract

A traditional philosophical controversy between representing general concepts as abstract *set*s or *classes* and representing concepts as concrete *prototypes* is reflected in a controversy between two mechanisms for sharing behavior between objects in object oriented programming languages. *Inheritance* splits the object world into *classes*, which encode behavior shared among a group of *instances*, which represent individual members of these sets. The class/instance distinction is not needed if the alternative of using prototypes is adopted. A prototype represents the *default* behavior for a concept, and new objects can re-use part of the knowledge stored in the prototype by saying how the new object differs from the prototype. The prototype approach seems to hold some advantages for representing default knowledge, and incrementally and dynamically modifying concepts. *Delegation* is the mechanism for implementing this in object oriented languages. After checking its idiosyncratic behavior, an object can forward a message to prototypes to invoke more general knowledge. Because class objects must be created before their instances can be used, and behavior can only be associated with classes, inheritance fixes the communication patterns between objects at instance creation time. Because any object can be used as a prototype, and any messages can be forwarded at any time, delegation is the more flexible and general of the two techniques.

1. Sets vs. prototypes: a philosophical dilemma with practical consequences

How do people represent knowledge about generalizations they make from experience with concrete situations? Philosophers concerned with the theory of knowledge have debated this question, but as we shall see, the issue is not without practical consequences for the task of representing knowledge in object oriented systems. Because much of object oriented programming involves constructing representations of objects in the real world, our mechanisms for storing and using real world knowledge get reflected in mechanisms for dealing with objects in computer languages. We'll examine how the traditional controversy between representing concepts as sets versus representing concepts as prototypes gives rise to two mechanisms, *inheritance* and *delegation*, for sharing behavior between related objects in object oriented languages.

When a person has experience in a particular situation, say concerning a particular elephant named Clyde, facts about Clyde can often prove useful when encountering another elephant, say one Fred. If we have mental representations of a concept for (Clyde, and a concept for Fred, the question then : How do the representations of Clyde and Fred share knowledge? How can we answer questions, such as Fred's color, number of legs, size, etc. by reference to what we already know about Clyde? In the absence of any mechanism for sharing knowledge between related concepts, we'd have to repeat all the knowledge about Clyde in a representation of Fred.

There are two points of view we can consider adopting. The first is based on the idea of abstract

* Published In: Proceedings of First ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR, September 1986.

sets. From learning about Clyde, we can construct a concept of the *set [or class] of elephants*, which abstracts out what we believe is true about all individual animals sufficiently similar to Clyde to be called elephants. The description of the set can enumerate all the "essential" properties of elephants. We can view Clyde as a *member* or *instance* of this class. In an object oriented system, the set approach involves creating an object to represent the set of elephants, and establishing a link representing the membership relation between the object representing Clyde and the set object. Since the description of the set represents what is true about all its members, we can answer questions about Clyde by referring to the description of the set. Establishing the same kind of membership link between Fred and the set of elephants enables Fred and Clyde to share some of the same knowledge. If Fred and Clyde share some additional properties, such as that of being Indian elephants, that are not shared by some other elephants, these can be embodied in a subclass object, which shares all the properties of the elephant set, adjoining the additional properties relevant to India.

But there's an alternative point of view. We can consider Clyde to represent the concept of a *prototypical elephant.* If I ask you to "think of an elephant", no doubt the mental image of some particular elephant will pop to mind, complete with the characteristics of gray color, trunk, etc. If Clyde was the elephant most familiar to you, the prototypical elephant might be an image of Clyde himself. If I ask you a question such as "How many legs does an elephant have?", a way to answer the question is to assume that the answer is the same as how many legs Clyde has, unless there's a good reason to think otherwise. The concept of Fred can have a connection marking its prototype as Clyde, as a mechanism for sharing information between the two weighty pachyderms. The description of Fred can store any information that is unique to Fred himself. If I ask "How many legs does Fred have?", you assume the answer is the same for Fred as for Clyde, in the absence of any contrary evidence. If you then learn that Fred is a three-legged elephant, that knowledge is stored with Fred and is always searched before reference to the prototype is made.

2. Prototypes have advantages for incremental learning of concepts

Though the concept of a set has proven fruitful in mathematics, the prototype approach in some ways corresponds more closely to the way people seem to acquire knowledge from concrete situations. The difficulty with sets stems from their abstractness; people seem to be a lot better at dealing with specific examples first, then generalizing from them than they are at absorbing general abstract principles first, and later applying them in particular cases. Prototype systems allow creating individual concepts first, then generalizing them by saying what aspects of the concept are allowed to vary. Set-oriented systems require creating the abstract description of the set first, before individual instances can be installed as members.

In mathematics, sets are defined either by enumerating their members, or by describing the unifying principles that identify membership in the set. We can neither enumerate all the elephants, nor are we good at making definitive lists the essential properties of an elephant. Yet the major impetus for creating new concepts always seems to be experience with examples. If Clyde is our only experience with elephants, our concept of an elephant can really be no different than the concept of Clyde. After meeting other elephants, the analogies we make between concepts like Fred and Clyde serve to pick out the important characteristics of elephants.

Prototypes seem to be better at expressing knowledge about defaults. If we assert grayness as one of the identifying characteristics of membership in the set of elephants, we can't say that there are exceptional white elephants without risking contradictions. Yet it is easy to say that Fred, the white elephant, is just like Clyde, except that he is white. As Wittgenstein observed, it is difficult to say, in advance, exactly what characteristics are essential for a concept. It seems that as new examples arise, people can always make new analogies to previous concepts that preserve some of the "defaults" for that concept and ignore others.

3. Inheritance implements sets, delegation implements prototypes

Having set the stage with our philosophical discussion of the issues of concept representation, we turn now to how these issues affect the more mundane details of implementation of object oriented programming systems.

Implementing the set-theoretic approach to sharing knowledge in object oriented systems is traditionally done by a mechanism called *inheritance*, first pioneered by the language Simula, later adopted by

SmallTalk, flavors and Loops, among others. An object called a *class* encodes common behavior for a set of objects. A class also has a description of what characteristics are allowed to vary among members of the set. Classes have the power to generate *instance* objects, which represent members of a set. All instances of a class share the same behavior, but can maintain unique values for a set of state variables predeclared by the class. To represent Clyde, you create a description for the class elephant, with an instance variable for the elephant's name, values of which can be used to distinguish Clyde and Fred. A class can give rise to *subclasses*, which add additional variables and behavior to the class.

Implementing the prototype approach to sharing knowledge in object oriented systems is an alternative mechanism called *delegation*, appearing in the actor languages, and several Lisp-based object oriented systems such as Director [Kahn 79], T [Rees 85], Orbit [Steels 82], and others. Delegation removes the distinction between classes and instances. Any object can serve as a prototype. To create an object that shares knowledge with a prototype, you construct an *extension* object, which has a list containing its prototypes which may be *shared* with other objects, and *personal* behavior idiosyncratic to the object itself. When an extension object receives a message, it first attempts to respond to the message using the behavior stored in its personal part. If the object's personal characteristics are not relevant for answering the message, the object forwards the message on to the prototypes to see if one can respond to the message. This process of forwarding is called *delegating* the message. Fred the elephant would be an extension object that stored behavior unique to Fred in its personal part, and referenced the prototype Clyde in its shared part.

4. Tools for representing behavior and internal state are the building blocks of object oriented systems

Each object oriented system must provide some linguistic mechanisms for defining the behavior of objects. The philosophy of object oriented programming is to use the object representation to encode both the procedures and data of conventional languages. Rather than define the procedural behavior or the data content of an object all at once, it is convenient to break both aspects of an object into a set of parts that can be accessed or modified individually by name.

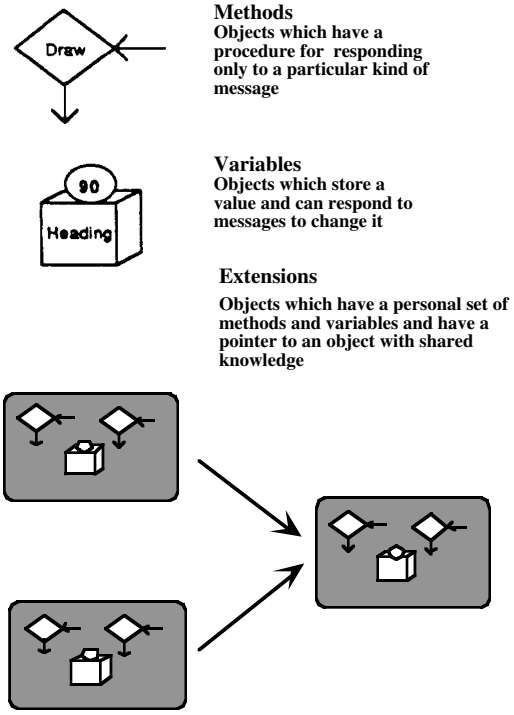
An object's internal state consists of *variables* or *acquaintances*, which can be accessed in most object oriented systems by sending the object a message consisting of the variable's name. An object's procedure for responding to messages [in actors, we say its *script*] can be composed of a set of procedures called *methods*, each of which is specialized for handling only a certain subset of the messages the object receives, identified by name. Breaking up an object's state into named variables means that different portions of the state can be modified incrementally, without affecting the others. Breaking up an object's behavior into named methods means that different portions of the behavior can be modified incrementally, without affecting the others. The language must then provide ways of combining groups of methods and variables to form objects and some means of allowing an object to share behavior [implemented as methods and variables] residing in previously defined objects. We will call these composite objects *extensions*. These building blocks are represented in the illustration "*Tools for sharing knowledge*" with "icons" to be used in further discussion.

Many object oriented languages supply primitive linguistic mechanisms for creating objects with methods, variables and extensions. An alternative approach, which is advocated in the actor formalism, is to define methods, variables and extensions as objects in their own right, with their behavior determined by a message passing protocol among them. Obviously, an object representing a method cannot itself have methods, otherwise infinite recursion would result. Using simple objects primitive to the system, a variable is defined to be an object that remembers a name and a value, and responds to access and modification messages. A method responds only to those messages for which it is designed, rejecting others. Extension objects use delegation to forward messages from one part of the object to another to locate the appropriate response.

Everyone who is already convinced of the utility of object oriented programming shouldn't have much trouble discerning the advantages of using object oriented programming in the implementation of the knowledge sharing mechanisms. Foremost among them is the ability to define other kinds of objects which implement alternatives to the standard versions. Instead of an ordinary variable, one might like to have "active" variables that take action when changed, "read-only" variables, maybe even "write-only" variables, each of which could be defined as a

different type of variable object. Alternative kinds of method objects can use differing strategies to combine behavior from contributing components, replacing the so-called "method combination" feature of the flavors system and making "multiple inheritance" easier. Different kinds of extension objects can make different efficiency tradeoffs on the issue of copying versus sharing.

Tools for Sharing Knowledge

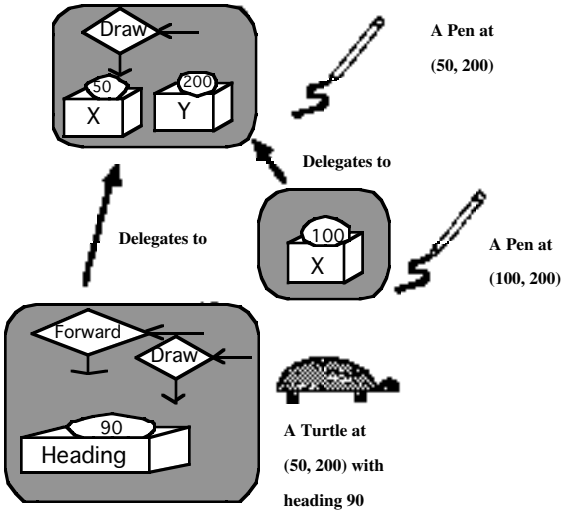


The mechanisms for sharing knowledge in object oriented languages have now grown so complicated that it is impossible to reach universal consensus on the best mechanism. Using object oriented programming itself to implement the basic building blocks of state and behavior is the best approach for allowing experimentation and co-existence among competing formalisms.

5. A Logo example illustrates the differences between delegation and inheritance

An example from the domain of Logo turtle graphics will illustrate how the choice between delegation and inheritance affects the control and data structures in an object oriented system. The delegation approach is illustrated in the figure titled "*Sharing Knowledge with Delegation*". The first thing we would like to do is create an object representing a pen, which remembers a location on the screen, and can be moved to a different location, drawing lines between the old and new locations.

Sharing Knowledge with Delegation



We start out by creating a prototypical pen object, which has a specific location on the screen x=200, y=50, and behavior to respond to the draw message. When we would like to create a new pen object, we need only describe what's different about the new pen from the first one, in this case the x variable. Since the y is the same and behavior for the draw message is the same, these need not be repeated.

The draw method will have to use the value of the x variable, and it's important that the correct value of x is used. When the draw method is delegated from the new pen to the old pen, even though the draw method of the original pen is invoked, it should be the x of the new pen that is used.

To insure this, whenever a message is delegated, it must also pass along the object that originally received the message. This is called the SELF variable in Simula, SmallTalk and flavors, although I find the term "self" a little misleading, since a method originally defined for one kind of object often winds up sending to a "self" of a different kind. In actor terminology, this object is called the client, since the object being delegated to can be thought of as performing a service for the original object. When a

pen delegates a draw message to a prototypical pen, it is saying "I don't know how to handle the draw message. I'd like you answer it for me if you can, but if you have any further questions, like what is the value of my x variable, or need anything done, you should come back to me and ask." If the message is delegated further, all questions about the values of variables or requests to reply to messages are all inferred to the object that delegated the message in the first place.

Suppose now we'd like to create a turtle at the same location as the original pen, using the original pen as a prototype. How is a turtle different from a pen? A turtle shares some of the behavior of a pen, but has additional state, namely it's heading. Remembering a heading is essential in implementing the additional behavior of being able to respond to forward and back messages by relying on the behavior of the response to the draw message. We may choose either to provide a new behavior for the turtle's draw operation, or rely on the draw operation provided by the original pen.

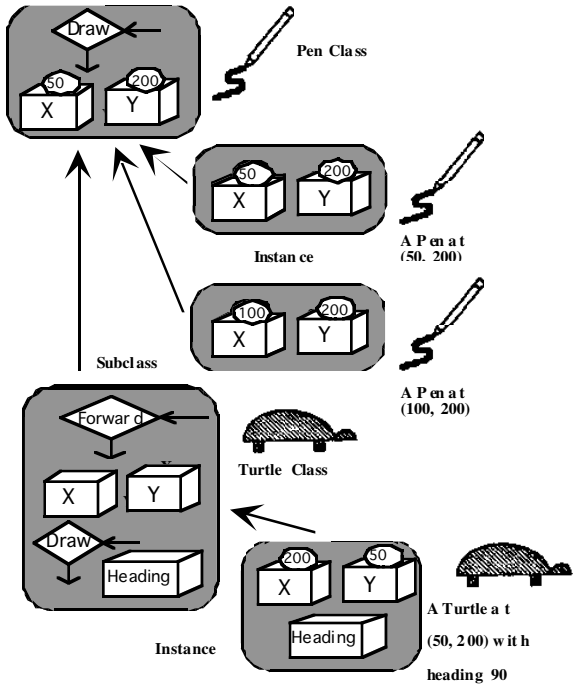
Let's look at the same example with the inheritance approach to sharing knowledge as found in Simula and SmallTalk instead of delegation. This is illustrated in the figure titled "*Sharing knowledge with inheritance*". With inheritance, it is necessary to create objects representing classes. To make a pen, it is first necessary to make a pen class object, which specifies both the behavior and the names of variables.

Individual pens are created by supplying values for all the instance variables of the pen class, creating an instance object. Values for all the variables must be specified, even if they do not have unique values in the instance. No new behavior may be attached to an individual pen. Extending behavior is accomplished by a different operation, that of creating a new subclass. The step which goes from an instance to behavior stored in its class is performed by a "hard-wired" lookup loop in systems like Simula and SmallTalk, not by message passing, as in the delegation approach.

To extend pens with new behavior, we must first create a new class object. Here a turtle class adds a new heading along with new behavior for the forward message. Notice that the variables from the pen class, x and y, were copied down into the turtle class. An individual turtle instance must supply values for all the variables of its class, superclass, and so on. This copying leads to larger instance objects for classes further and further down the inheritance hierarchy. The lookup of methods, performed by a primitive, unchangeable routine instead of message passing, starts a search for methods in the class of an object, and proceeds up the subclass-to-superclass chain.

How does a method inherited from the pen class to the turtle class access a method implemented in the turtle class? Since inheritance systems usually do not use message passing to communicate from subclass to superclass, they can't pass the turtle object along in the message, as we would in delegation.

Sharing Knowledge with Inheritance



Instead most use variable binding to bind a special variable self to the object that originally receives a message. We shall see later on that this leads to trouble.

In addition, inheritance systems also allow the "shortcut" of binding all the variables of an instance so that they can be referenced directly by code naming in methods as free variables. While this is sometimes more efficient, it short-circuits the message passing mechanism, defeating the independence of internal representation which is the hallmark of object oriented programming. Since variable references use different linguistic syntax than message sends, if we

wanted to change the coordinate representation from x and y to polar coordinates using rho and theta, we'd have to change all the referencing methods. Sticking to message passing to access x and y means that even if the coordinates were changed to polar, we could still provide methods that compute the rectangular coordinates from the polar, and the change would be transparent.

I hope these diagrams leave you with the impression that the delegation approach is simpler. To create two pens and a turtle, the inheritance approach requires the additional steps of creating pen class objects and turtle class objects. Also, we have to have two different kinds of links between objects, the subclass link and the instance link, whereas the delegation approach only requires a message passing relationship between the linked objects.

6. Are inheritance and delegation equally powerful?

An obvious question to ask about the preceding discussion of inheritance and delegation is whether the two techniques have the same expressive power. The answer is *no*.

Given delegation, it is easy to see how we could implement the functionality of inheritance. We can create special class objects that respond to messages to create new instances. We need only arrange that the class objects observe the copying of variables from the superclass chain when they create instances. Instance objects are given behavior that implements the lookup methods, roughly as follows.

```
If I'm an INSTANCE object
   and I receive a message
   with a SELECTOR and some  ARGUMENTS:
If the SELECTOR matches
   one of the VARIABLE names
   in my CLASS [or SUPERCLASS, etc.],
I return the corresponding value,
   stored in myself.
Otherwise, I look for a METHOD
   whose NAME matches
      the SELECTOR of the message
   in the list of local METHODS
      of my CLASS.
If I find one,
   I bind the variable SELF to myself
      [the INSTANCE object].
   I bind the names of
      the variables of my CLASS,
   [and all the variables
      up the SUPERCLASS chain]
   to their values in the INSTANCE.
 Then I invoke the METHOD.
If there's no method
   in my CLASS's METHOD list,
 I try to find a method
   in the SUPERCLASS,
   and so on up the SUPERCLASS chain.
```

How about the other way? Can inheritance implement delegation? Unfortunately not. The reason is a little tricky to understand, but it has to do with the treatment of the self variable, which prevents a proper implementation of forwarding of messages.

Often, a method for handling a message may need to ask the object that originally received the message to perform some service. A turtle object which receives a back message would like to turn it into a forward message sent to the same object, but negating the number of steps, so that back 100 is like forward -100. In delegation, when a method is delegated a message, it receives a component called the client in the delegate message, which has the object that originally received the message.

In inheritance systems, a distinguished variable named self is automatically bound to the recipient of a message during the execution of code for a method. When the method search proceeds from the original class to a superclass, the value of the self variable doesn't change, so that superclass methods can reply to the message "as if" they were methods of the original object. However, when a user sends a message, the self variable is always re-bound, so that it is generally *not* possible for the user to designate another object to reply in place of the object which originally received the message. True delegation cannot be implemented in these systems.

An example, illustrated in the figure "The SELF Problem," will make this clear. Suppose we would like to extend a particular turtle object to create a turtle which draws dashed instead of solid lines. The obvious way to do this is to have the dashed-turtle intercept the forward message and break up the interval into pieces, delegating a message to draw a series of shorter lines to a solid-line turtle. If, in an inheritance system, the dashed-line turtle simply sends a forward message to the solid-line turtle, then self will be bound to the solid-line turtle. Our earlier implementation of back in terms of forward will then stop working, since a message to the dashed-line

turtle to go back will try to send a forward message to self and draw a solid line instead!

Be careful about confusing this example with an alternative implementation using inheritance systems, which would create a dashed-turtle class as a subclass of solid-turtle class. While such an implementation could have the correct behavior with respect to the back message, it still wouldn't count as an implementation of delegation. Remember, what we were trying to do was to see if an object could forward messages to some other *already existing* object. A dashed turtle instance wouldn't be forwarding any messages to an instance of solid turtle, since it would just inherit copies of the variables and methods from solid turtle.

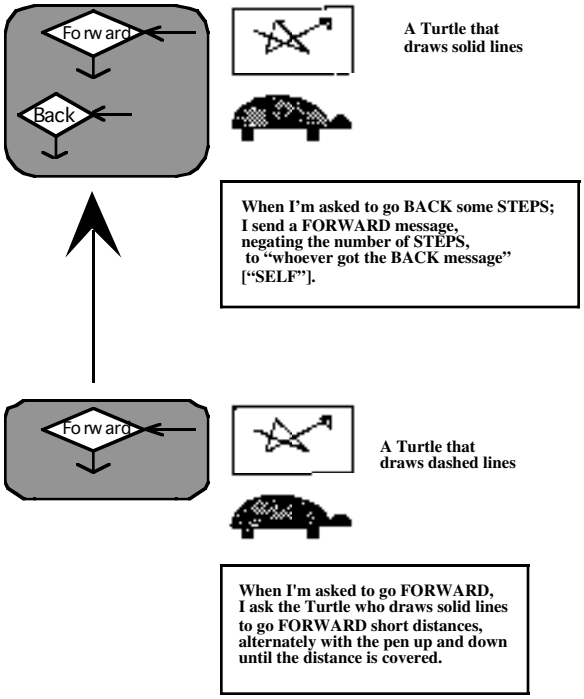
7. What about efficiency?

The efficiency comparison between delegation and inheritance boils down to time/space tradeoffs. Some have argued that inheritance is more efficient because it requires fewer messages, but this comes at the cost of increasing the size of objects. Because variables are copied down from superclass to subclass, instances become larger and larger the farther down you get in the inheritance hierarchy. With delegation, each object need only specify what's different about it from already existing prototypes, so the size of objects does not necessarily depend on the depth in the hierarchy of shared objects. A look at the diagram illustrating the data structures for pen and turtle objects will confirm inheritance's speed advantage and delegation's space advantage.

Smaller objects make for faster object creation times, which can be important in systems that create large numbers of small objects with short lifetimes, as opposed to small numbers of large objects with long lifetimes. Reducing the size of objects may also improve the efficiency of virtual memory, by improving locality of reference, allowing a higher density of frequently referenced objects in the primary memory. With a copying garbage collector, such as that described in [Lieberman and Hewitt 83], smaller objects can improve the efficiency of garbage collection by reducing the copying overhead.

Implementors shouldn't get scared away by the search required to find methods and variables in the delegation approach. There's a simple, effective trick for reducing the search time: *caching* the result of lookups. Caches are a way of trading space for speed, mitigating any negative effects of the speed-for-space tradeoff made by delegation. Caches make a more effective use of the extra memory than indiscriminately copying instance variables, because the memory they do use is sure to be in constant use. Caches don't restrict flexibility in interactively modifying the programming environment the way copying and compilation optimizations do.

The "SELF" problem



On conventional machines, probably no implementation of delegation is going to surpass variable lookup via registers and stack indexing for raw speed. But in their zeal to speed up variable lookup, implementors have forced decisions such as large object size on object-oriented languages, which adversely affect efficiency. Parallel machines with large address spaces will make the attractiveness of such register-oriented optimizations fade.

SmallTalk [Krasner 84] reports a 93% "hit rate" for a moderately sized cache, 1000 objects. This means that any savings by inheritance over delegation in lookup could at best affect the remaining 7%. The best thing to do seems to be to keep a global cache, and invalidate it whenever any changes are made to the sharing hierarchy. A change will then slow the system down for the next 1000 messages, or whatever

time the cache takes to fill up again. “Smarter” alternatives, such as per-object caches are probably not worth the extra trouble they would cause for incremental software modification, since the hit rate on a global cache is so high. Since both inheritance and delegation can be implemented almost equally efficiently, it seems that there's little reason to sacrifice the extra flexibility of delegation on efficiency grounds.

8. Re-directing I/O streams illustrates an important application of delegation

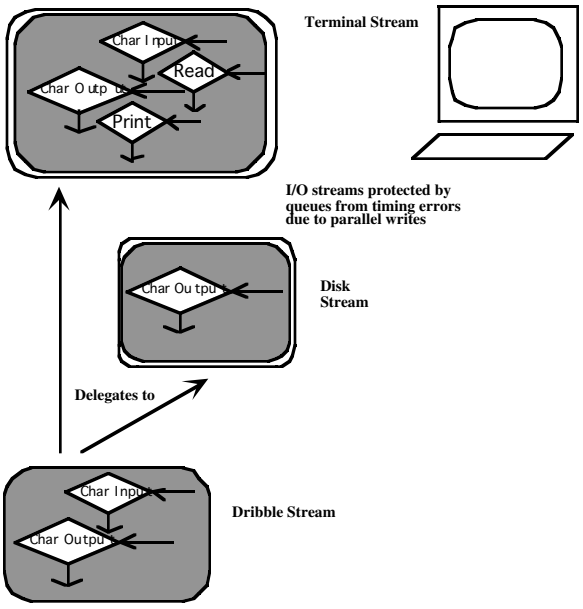
Many object oriented systems make good use of object oriented programming techniques to implement input-output *streams*. Such a stream is an object that receives messages to input or output a character, a line of text, an expression. Systems usually have global variables designating the “current” sources of input and output, which is by default bound to an object representing the stream of characters being displayed on the window of a screen of an interactive display.

The name “stream” suggests the continual flow of characters or pixels between the user and the system. A very useful kind of object is that which implements a “dam” to divert the stream to other destinations, or “plumbing” which connects one stream with another. A *dribble file* is a sequential file maintaining a record on disk of the history of input-output interactions, to provide a more permanent recording of interactions than the ephemeral twinkling of pixels. A dribble file can be implemented by replacing the stream which represents interactions at the terminal with one that writes them to disk also.

The dribble stream needs the ability to masquerade as the terminal stream. It should have the same responses to all the messages that the ordinary terminal stream has, and also provide the additional behavior of writing to the disk. The streams should be considered indistinguishable from the point of view of all programs which perform input-output.

To implement the dribble stream cleanly, we'd like it to be the case that the implementation of the dribble stream shouldn't have to know the precise details of the implementation of the stream which it is replacing. We might, for example, like to use a single dribble stream with both a stream to a directly connected interactive terminal and a stream interacting over a network.

Can a dribble stream “masquerade” as a terminal stream?



The implementation using delegation is convenient and straightforward. Messages which do character output are intercepted and the disk output is interposed.

A DRIBBLE-STREAM is an object
that logs interaction on a STREAM,
and records it on disk using a FILE-NAME
If I'm a DRIBBLE-STREAM and
I get a massage to input or output
a CHARACTER,
I output the CHARACTER
to the disk stream to the FILE-NAME.
Then I delegate the message
to output a CHARACTER
to the original STREAM.
If I'm a DRIBBLE-STREAM and
I get any other message,
I simply delegate the message
to the STREAM.

It works to take care of only the single-character input and output messages because presumably all higher level messages, like print of an object, are ultimately implemented in terms of the single-character versions. The method which performs a higher level print operation would ultimately send a character output message to its client [send to self].

Surprisingly, many inheritance systems make it difficult to implement this simple extension to the behavior of streams. One villain is the insistence of systems like flavors and SmallTalk on defining separate procedures for handling each type of message. Attempting to try to implement dribble-stream as a subclass of stream in systems of this ilk, we would find that there's no easy way to say "... and send all of the irrelevant messages through to the original stream". We would be forced to define one method to intercept the character output message to write to the disk, another to intercept the print message, another to intercept the print-line message, and so on for every relevant message. Every time another message was added to the original stream another method would have to be added to the dribble-stream, with tediously repetitive code. This also has the unfortunate effect of making the implementation of dribble-stream now sensitive to the details of exactly which messages its embedded stream accepts, inhibiting the ability to re-use the implementation with different types of streams.

Adding to the system the definition of a dribble-stream class or flavor would only give the ability to create *new instances* of dribble stream objects. It would not be possible to create a dribble stream which used a previously existing stream object. We'd then have to make new terminal streams, network streams, or other kind of streams, to be able to take advantage of the recording functionality. We shouldn't have to reproduce every kind of stream in the system just to have the dribble capability!

If, instead, we attempt to make a dribble stream which holds the interaction stream as one of its instance variables, we face the problem that there is no way for the dribble stream to correctly forward a message like print to the value of the variable. Because of the way these systems handle the self variable, the forwarding of messages to the original stream won't work, for the same reason as in the turtle example. Sending a print message to the instance variable would re-bind the self variable, so it would result in sending lower-level messages directly to the interaction stream and not to the dribble stream. So it seems as though any straightforward attempt to implement the dribble stream as a simple behavioral extension in many inheritance systems is doomed.

9. Parallelism causes problems in inheritance systems because of the SELF variable

There's an additional problem in the case that the stream can accept messages from more than one parallel process. Because the stream holds modifiable state [such as a screen bitmap], the stream must be protected against timing errors resulting from two processes trying to write to the stream at the same time. A technique such as serializer objects [Hewitt, Attardi, Lieberman 79] or monitors must be used. This means that when the stream receives a write message, it "locks", so that subsequent messages to the stream must wait in a queue for the stream to finish processing the first write message.

Now, if a message to a dribble stream tries to process a print message by sending a character-output message to the self variable, it will find self bound to a stream which is locked waiting for that very print message to complete! Deadlock!

Since delegation uses message passing, when the dribble stream delegates to a terminal system it can supply [as the client in the delegate message] an *unserialized* version of itself, which can process the message without waiting.

10. Delegation is more flexible than inheritance for combining behavior from multiple sources

Often, an object will want to utilize behavior that appears in more than one other already existing object. The behavior that a system needs to implement a particular "feature" can be packaged up as a single object, and sometimes an object will want to combine several of these features to implement its behavior. For example, window objects might have titles, borders, size adjustments, etc. A particular window object may choose some of these features and not others. Features may be independent of one another, or they may interact.

The solution in inheritance systems is to create a class object that mentions a list of other classes whose behavior it wishes to sham. All the methods and variables mentioned in any of the classes are inherited by the combined object. Systems like flavors allow optionally, on a per-method basis, supplying an option for how to combine behavior when more than one component contributes a method. Typical options are to invoke all the contributing methods, impose an order on them, or return a list of the results.

The problem with this style of combining behavior from multiple sources is that it fixes the pattern of communication between objects before the time an instance object is created. This limits the extent to which behavior from previously existing objects can be used dynamically. By contrast, with delegation, the communication patterns can be determined at the time a message is received by an object.

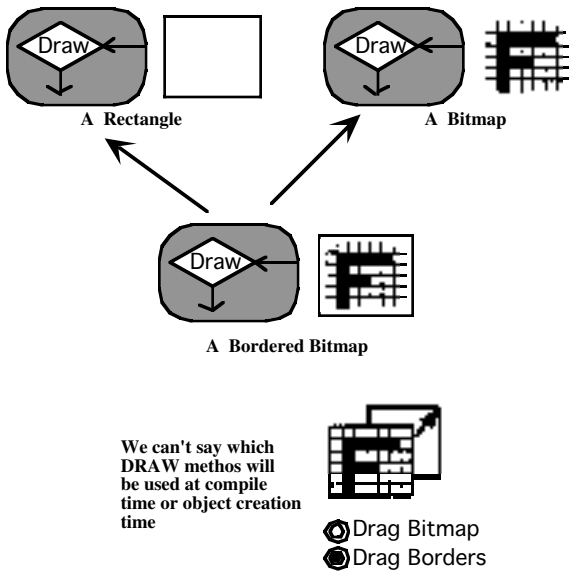
With delegation, a method for an extension object can simply access the prototypical objects from which it derives behavior on the shared list. A window which wants to invoke the draw action of a previously defined rectangle object acting as its borders can simply delegate the draw message to the rectangle object. Thus delegation doesn't require "method combination" or an inventory of esoteric combining operations. The behavior is simply programmed in the method for the combined extension object. Should a programmer wish to build a library of common combination techniques, it is easily done by constructing variants on the standard method object, so delegation could be made as concise as method combination in inheritance systems. With inheritance, if a window class includes a "borders mixin", the window does not contain an independent object representing its borders, so it is not possible to send a message to the borders of a window independent of the window object itself. The window class merely contains a mixture of the methods and variables inherited from the borders and other contributing components.

In highly responsive interactive systems, it is often necessary to wait until a message is received to determine how behavior from component objects will be utilized. Here's a simple example in which dynamic utilization of behavior from multiple sources is required, illustrated in the figure "*Delegation allows communication patterns to be decided at run time*".

A bordered bitmap can be built from a rectangle, which can display its borders, and a bitmap which can transfer an array of pixels to the screen. What should the draw response for the bordered bitmap be? With inheritance you create a bordered-bitmap class that inherits both from rectangle and bitmap, saying that both draw methods are to be used. Fine.

But now suppose we'd like to give the user the option of changing dynamically which behavior is used. When the bitmap is dragged across the screen, the transfer of the entire array on every mouse movement might be too slow, so it might be preferable to give the user the option of just dragging the outline of the bitmap instead. A reasonable thing to do is to give the user an on-screen toggle switch to decide the behavior, and the user can potentially change the behavior at any time. So the behavior of the bordered bitmap cannot be decided before the object is created. With delegation, when the bordered bitmap gets a draw message, it can decide whether to delegate the message to the rectangle object that it contains, or to the bitmap object, or both.

Delegation allows communication patterns to be decided at run time



Inheritance systems are also plagued by what I call the *one-instance class problem*. When systems are composed of large numbers of objects with slightly varying behavior, you wind up having to create new class objects often just to have one or a few instances. It is necessary to create ad-hoc classes such as "window with a wide border, times roman font and no title" just to combine features for a single instance.

11. Delegation is advantageous for highly interactive, incremental software development

An important issue to consider when evaluating the tradeoffs between inheritance and delegation is the consequences for incremental software development.

As we have seen above, inheritance tends to encourage copying of variables and methods, while delegation encourages sharing. If a prototypical object changes behavior, then all objects which mention that prototype on their shared list will automatically "feel" the change. If changes are made to an inheritance hierarchy, such as adding a new instance variable, or changing the class structure, information copied from the old data structure may be rendered obsolete. Broadcasting the result of changes to copies puts a burden on the operations which make incremental changes in the software environment. An extreme example of this occurs in the flavors system where simply adding a method to vanilla-flavor, the root of the inheritance hierarchy, results in recompilation of every flavor in the entire system! This effectively prohibits any modifications to objects near the top of the inheritance hierarchy.

Though delegation has been the minority viewpoint in object oriented languages, it is slowly becoming recognized as important for its added power and flexibility. Part of the reason for neglect of the delegation approach has been historical. Simula, one of the first object oriented languages, adopted the inheritance technique. It fixed communication patterns between objects at compile time, as was appropriate for a compiled language of the Algol family. The specific mechanisms for this were then "inherited" by SmallTalk and others, without reconsidering whether the approach was still appropriate for an interpretive language in a more highly interactive programming environment. I hope the preceding discussion has convinced you that the approach of modeling concepts using prototypes and implementing behavior in object oriented languages using delegation has distinct advantages over the alternative point of view using classes and inheritance.

12. Acknowledgments

Major support for the work described in this paper was provided by the System Development Foundation. Other related work at the MIT Artificial Intelligence Laboratory was supported in part by DARPA under ONR contract N00014-80-C-0505.

Carl Hewitt's ideas concerning actors, and especially the impact of parallelism on object-oriented programming were important influences. Kenneth Kahn and Luc Steels implemented object-oriented languages which adopted delegation mechanisms and also influenced these ideas. Alan Borning reached similar conclusions in the context of the ThingLab system implemented in SmallTalk. Koen de Smedt provided a helpful critique of a talk I gave on these issues in Nijmegen, the Netherlands.

References

[Birtwistle, Dahl, Myhrhaug, and Nygaard 73]
G. M. Birtwistle, O-J Dahl, B. Myhrhaug, K. Nygaard.
Simula Begin.
Van Nostrand Reinhold, New York, 1973.

[Bobrow 85]
D. Bobrow, K. Kahn, M. Stefik, G. Kiczales.
Common Loops.
Technical Report, Xerox Palo Alto Research Center, 1985.

[Bobrow, Stefik 83]
Daniel Bobrow and Mark Stefik.
Knowledge Programming in Loops.
Al Magazine, August, 1983.

[Borning 86]
Alan Borning.
Classes Venus Prototypes in Object-Oriented Languages.
In Fall Joint Computer Conference. ACM/IEEE, Dallas, Texas, November, 1986.

[Goldberg, Robson 83]
Adele Goldberg and David Robson.
SmallTalk-80: The Language and its Implementation.
Addison-Wesley, Reading, MA, 1983.

[Hewitt 79]
Carl Hewitt.
Viewing Control Structures as Patterns of Passing Messages.
In P. Winston and R. Brown [eds.], *Artificial Intelligence, an MIT Perspective*. MIT Press, Cambridge, MA, 1979.

[Hewitt, Attardi, Lieberman 79]
Carl Hewitt, Giuseppe Attardi, and Henry Lieberman.
Security And Modularity In Message Passing.
In *First Conference on Distributed Computing*. IEEE, Huntsville, 1979.

[Kahn 79] Kenneth Kahn.
Creation of Computer Animation from Story Descriptions.
Ph.D. thesis, Massachusetts Institute of Technology, 1979.

[Krasner 84] Glenn Krasner, editor.
SmallTalk-80: Bits of H istory and Words of Advice.
Addison-Wesley, New York, 1984.

[Lieberman 86a] Henry Lieberman.
Concurrent Object Oriented Programming in Act 1.
In A. Yonezawa and Tokoro [eds.], *Concurrent Object Oriented Programming*. NUT Press, Cambridge, MA, 1986.

[Lieberman 86b] Henry Lieberman.
Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object Oriented Systems.
In J. Bezivin, P. Cointe [eds.], *3eme Journees d'Etudes Languages Orientes Objets*. AFCET, Paris, France, 1986.

[Lieberman and Hewitt 83]
Henry Lieberman and Carl Hewitt.
A Real Time Garbage Collector Based on the Lifetimes of Objects.
CACM 26(6), June, 1983.

[Moon, Weinreb 84]
David Moon, Daniel Weinreb, et al.
Lisp Machine Manual.
Symbolics, Inc. and MIT, Cambridge, MA, 1994.

[Rees 85] Jonathan Rees, et al.
The T Manual.
Technical Report, Yale University, 1985.

[Steels 82] Luc Steels.
An Applicative View of Object Oriented Programming.
Technical Report Al Memo 15, Schlumberger-Doll Research, March, 1982.