Delegation Is Inheritance

Lynn Andrea Stein
Brown University
Department of Computer Science
Providence, R.I. 02912

Abstract

Inheritance and delegation are alternate methods for incremental definition and sharing. It has commonly been believed that delegation provides a more powerful model. This paper demonstrates that there is a "natural" model of inheritance which captures all of the properties of delegation. Independently, certain constraints on the ability of delegation to capture inheritance are demonstrated. Finally, a new framework which fully captures both delegation and inheritance is outlined, and some of the ramifications of this hybrid model are explored.

1. Introduction

Delegation and inheritance are mechanisms for sharing in object-oriented systems. Inheritance collects objects into groups according to their "type," or category;. much of the sharing provided by inheritance is achieved through these category, or class, objects. Delegation does not define category objects in this way, but rather allows sharing among arbitrary objects.

[Lieberman] gives perhaps the most complete comparison of these two systems to date; however, he does so in informal terms. Thus, he is able to give the outline for a simulation of inheritance using delegation, but goes on to conclude that the reverse mapping—given inheritance, performing delegation—is impossible.

This paper develops a natural model for inheritance which also captures delegation in its entirety. In this scheme, class (*isa*) inheritance *is* delegation. This inheritance scheme is simply inheritance with strict subtyping; what is unusual is that it is the classes, and not the instances, which are used to simulate delegation.

While this type of inheritance properly contains delegation, supporting all the functionality of Lieberman's "prototype" system, no delegation simulation of inheritance has this property with respect to inheritance. In particular, while delegation can capture the state of several types of inheritance hierarchies at any moment in time, it cannot model the constraints on change over time imposed by inheritance. This result is independent of the map from inheritance to delegation.

The model for inheritance described below has natural extensions which allow for a more dynamic structure. Unique objects can be incorporated into a hierarchy without the creation of extraneous classes; similar objects can be allowed to accumulate around these extended instances; and new classes can be created from these objects as the need arises. The model presented in section 4 provides the full functionality of both inheritance and delegation, as well as the additional functions of *minimal* (*vs.* absolute) *guarantee*, *accumulation*, and *promotion*.

1.1. Notations and Conventions

In the discussion below, I use limited models for both delegation and inheritance. Specifically, only single inheritance and single-parent delegation are described, and actual cancellation of attributes is not discussed. The problems introduced by allowing multiple inheritance or delegation, and the difficulties of cancellation, are similar in the two frameworks and do not in general add insight into the underlying models. Where exceptions occur, these are noted.

In the formal models below, the distinctions between types of attributes of an object—variables and methods—is blurred. This represents no reduction in the power of the model, since a variable can be treated simply as a pair of *get* and *set* messages. However, the need for local, persistent storage, as in the case of the variable to be *got*ten and *set*, cannot be entirely eliminated. Any attribute may require, either implicitly or explicitly, such persistent storage. This need refers to any values that remain constant from one invocation of an attribute to the next, until such times as they are explicitly changed. A special notation is introduced to identify those places where such persistent storage must be allocated.

For an attribute x valid for an object a, $val_a(x)$ indicates that a maintains its own copy of any values required by message x. $Val_a(x)$ may be empty, if the attribute x needs no permanent local storage (*e.g.* square-root, which has nothing to store but computes its value anew each time it is called). Alternatively, it may "store" values for several "variables," if attribute x requires static storage of these multiple values. If more than one attribute uses a single "variable," one attribute is (arbitrarily but in a reproducible manner) selected to maintain its value, while the others are able to use and change that value; *i.e.* $\bigcap_{x \in \text{attributes of obj. } a} val_a(x) = \emptyset$.

For example, the *pens* in figure 1 require X and Y. These can be simulated with $get-X$ and $get-Y$ messages, but somewhere values allowing the computation of X and Y need to be stored. Saying $val_{pen_1}(get-X)$ differs from saying X in that X and Y need not be stored explicitly; for example, $val_{pen_1}(get-X)$ may be $\{ r, \theta \}$ if the *pen*'s location is actually stored in polar coordinates. In this case, $val_{pen_1}(get-Y)$ would be $\emptyset$, since it would use the values of r and θ managed by $get-X$.

Many of the figures and examples used in this paper are borrowed from or based upon the Logo examples of [Lieberman].

2. Mechanisms and Models

2.1. Inheritance

Inheritance schemes involve two kinds of objects: classes and instances. A class defines the "shape" of its instances: attributes that each of its instances will have. Each instance maintains its own storage for the values of these attributes. The state of an object is given by the values of its attributes at any point in time. An example of inheritance is given in figure 1. The dotted boxes denote ***instance templates***, the attributes defined by the class for its instances.

Inheritance allows incremental definition of classes, that is, the "type" of an instance may be defined in terms of the "types" of other instances (*turtles*, as a group, are like *pens*), but an individual instance may not be defined directly in terms of another instance (this ***turtle*** is like that *pen* or that *turtle*).

Inheritance allows instances to share attributes, but not values. Since all instances of a class (and its subclasses) use the definitions of attributes stored in the class, any change to the attribute stored in the class will change all of the instances. However, values are stored in the instance, not in the class, and so are not shared. In inheritance, instances are independent. Changing the state (values) of one instance cannot affect any other instances. This implies that one instance cannot depend on another.

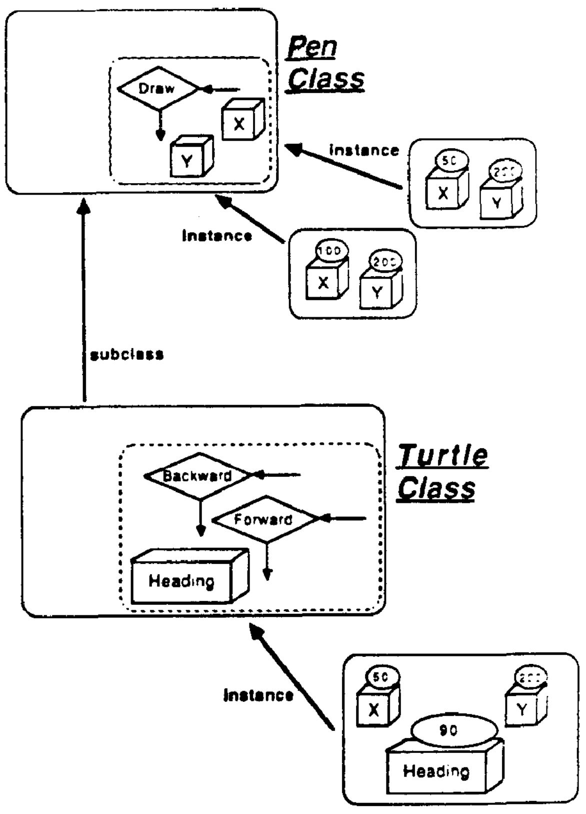


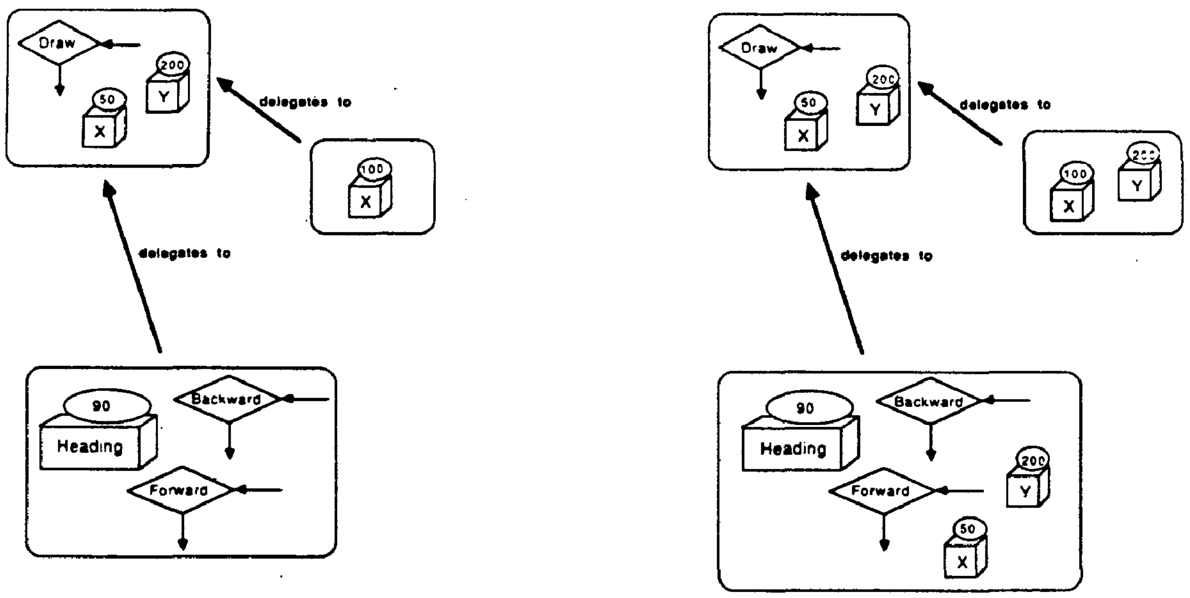
Figure 1. Inheritance.

Instance objects in an inheritance hierarchy must be grouped by type. That is, the instance template ensures that all instances of a single class will be of the same "type"—*i.e.* have exactly the same "shape", and that all instances with the same "shape" will be instances of the same class. This arrangement of the hierarchy is extremely useful for type checking and compilation, as well as for indexing in a database system.

2.2. Delegation

In delegation, there is only one type of object. Since individual real-world items are represented by these objects, they are often taken to be "instances without classes." In fact, their behavior is really closer to that of classes. Figure 2 gives two examples of a delegation hierarchy. In figure 2a, the ***turtle*** always sits directly on top of its parent *pen*—it depends on the *pen* for its location. In figure 2b, however, the three objects are independent. This is the basis for [Lieberman]'s proof that delegation can capture the behavior of inheritance, and is valid to a limited extent. Some of the problems with it are discussed at the beginning of section 3.4.

Delegation allows incremental definition of all objects. Any object may be defined directly in terms of



(*a*) Dependent objects.

(*b*) Independent objects.

Figure 2. Delegation.

any other. There are no "type" objects, like classes, and attributes cannot be declared without being "stored", in the way that instance templates allow classes to declare attributes for their instances.

Delegation allows sharing of both methods and variables. If an object delegates an attribute—method or variable—to a prototype, then any changes to those attributes—or their values—will affect both the object and the prototype. In this way, objects in a delegation hierarchy may be dependent on one another.

Delegation does not enforce grouping by type. In a delegation hierarchy, two objects of different "type," or "shape," may delegate to the same prototype. Similarly, two objects of the same "shape" may delegate to different prototypes—in fact, they may be in radically different parts of the hierarchy.

2.3. Classes and the Axiom Of Upwards Compatibility

In an inheritance model, classes are objects. As such, they may have attributes of their own. These may be attributes defined in the class, like attributes in a delegation system, or—in a world where classes are themselves instances of a *meta*class—they may be instance attributes of the metaclass.

The rules for classes inheriting from metaclasses are exactly the same as for instances inheriting from classes—since in that case classes are just instances, and metaclasses classes whose instances happen also to be classes. The hard question is in determining how classes should inherit attributes from their *super*classes. This inheritance could be like instance inheritance—each class would have its own copy of all class variables. Alternatively, it could be like delegation, and a class could use the values of variables supplied by its superclass unless it redefined these variables locally.

The *Axiom of Upwards Compatibility* makes a strong argument that disallowing sharing provides the wrong semantics for class variable inheritance. The Axiom of Upwards Compatibility, or the Strict Is-A Rule, says that if A *isa* B (*i.e.* A is a subclass of B, or "all As are Bs), then anything that is true of B must also be true of A. Since a subclass *isa* a superclass, anything true of the superclass—like having a class attribute—must be true of the subclass. Further, any changes to the superclass must be reflected immediately by the subclass: the subclass ***depends*** on the superclass. This solution to class inheritance is illustrated in figure 3.

One nice consequence of this is that it gives us a good way to model delegation in inheritance. [Lieberman] shows that several different ways of mapping prototypes to instances fail, essentially because there is no way to have one instance depend on another in inheritance. However, he does not consider the possibility of mapping prototypes into *classes*. In fact, since class inheritance is fundamentally the same as delegation, this mapping provides the necessary key to showing that delegation really isn't any more powerful than inheritance. The mapping from delegation to inheritance is given formally in the next section.

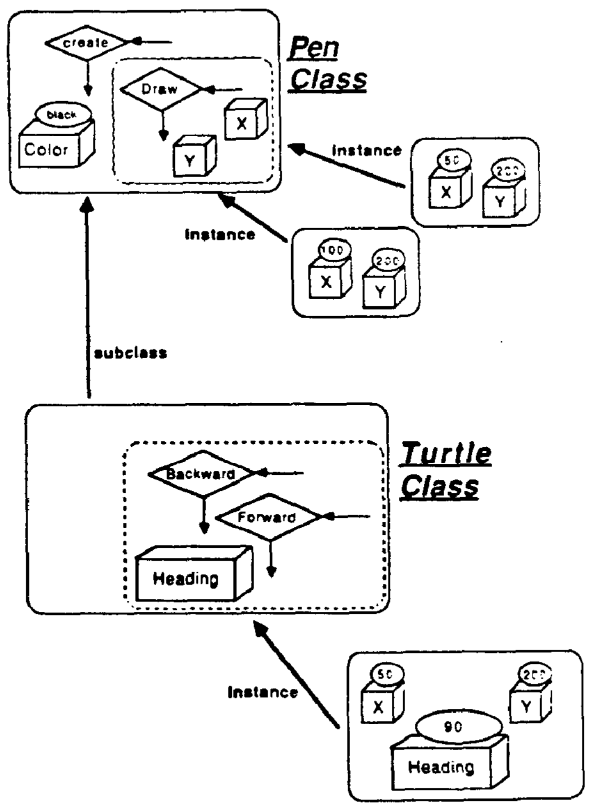


Figure 3. Class Variables.

3. Formal Models

This section contains the formal proofs that inheritance and delegation can be used to model one another. The demonstration that delegation can model inheritance is based on the idea that while delegation allows objects to depend on one another, it does not require it, and the independence of instances in inheritance can be simulated in delegation. While this observation is correct, it has flaws; these are discussed at the beginning of section 3.4. The proof that inheritance can model delegation is based on the observation that strict *isa* inheritance is simply delegation; rather than mapping prototype objects to instances, the proof makes them classes in an inheritance hierarchy without instances. This then captures the full functionality of a delegation hierarchy.

3.1. Delegation

Let $\mathbb{D}$ be a delegation hierarchy. Then $\mathbb{D} = \{ O, X, U \}$, where O is a set of objects, X is a set of attributes, and U is a set of storage for the values of attributes in X.

Each object $o \in O$ is defined by its prototype, $proto(o)$, its attributes, $attributes(o)$, and the values it maintains for these attributes, $V(o) = \{ val_o(x) \mid x \in attributes(o) \}$. If o is a root of the hierarchy, we define $proto(o) = \emptyset$. Since $proto()$ is acyclic, we define $proto^*(o)$, the transitive closure of $proto(o)$, *i.e.*,

$$\{ o, proto(o), proto(proto(o)), \ldots, proto^n(o) \}$$

where n is the least integer such that $proto^{n+1}(o) = \emptyset$ We can also define a recursive relation:

$$\forall o \in O, \text{ if } proto(o) = \emptyset \text{ then } p\_depth(o) = 0$$
$$\text{else } p\_depth(o) = p\_depth(proto(o)) + 1$$

The Axiom of Delegation defines what delegation means on this hierarchy:

Axiom of Delegation

$\forall o \in O$, attribute x is valid for o with value v iff either

(i) $x \in attributes(o)$ and $v = val_o(x)$ or

(ii) Case (i) fails and x is valid for $proto(o)$ with value v

3.2. Inheritance

Let $\mathbb{I}$ be an inheritance hierarchy. Then $\mathbb{I} = \{ C, I, Y, W \}$, where C is a set of classes, I is a set of instances, Y is a set of attributes, and W is a set of storage for the values of attributes in X. C and I need not be disjoint; together, they form the objects in $\mathbb{I}$.

Each class object $c \in C$ is defined by its superclass, $super(c)$, its attributes, $class\_attributes(c)$, and the values it maintains for these attributes, $V(c) = \{ val_c(x) \mid x \in class\_attributes(c) \}$. It also has an $instance\_template(c)$. For simplicity, I will assume throughout that $\bigcup_{o \in I \cup C} class\_attributes(o) \cap \bigcup_{o \in I \cup C} instance\_template(o) = \emptyset$— *i.e.* that it is possible to distinguish between class and instance attributes. This is not, however, strictly necessary. Since $super()$ is acyclic, so we can define $super^*(c)$ in the same manner as $proto^*(o)$, and $c\_depth(c)$ in the same way as $p\_depth(o)$. Note, however, that c_depth is not defined on $I - C$.

An instance object $i \in I$ is defined by its class $class(i)$, and $V(i)$, the values it assigns to the attributes in the instance templates of its class, its class's superclass,...; *i.e.* the instance templates of all the classes in $super^*(class(i))$. That is $V(i) = \{ val_i(x) \mid x \in (\bigcup_{c \in super^*(class(i))} instance\_template(c)) \}$

Class_attributes(c) represents those attributes of c that are not instance attributes of its meta-class, but are simply defined at c. It is entirely possible for an object in $\mathbb{I}$ to have both class attributes, which it defines for itself, and instance attributes, defined by its metaclass. In this case, it acts as a member of both C and I.

The rules for inheritance are described below. The asymmetry of class inheritance (Axiom 1) and instance inheritance (Axiom 2) reflects the different mechanisms used to implement sharing in the two types of objects. In systems where classes are instances, these objects inherit by both the rules for $a \in C$ and $a \in I$; Axiom 3 makes both Axioms 1 and 2 iffs.

First Axiom of Inheritance

$\forall c \in C$, attribute x is valid for c with value v if either

(i) $x \in class\_attributes(c)$ and $v = val_c(x)$ or

(ii) Case (i) fails and x is valid for $super(c)$ with value v

Second Axiom of Inheritance

$\forall i \in I$, attribute x is valid for i with value v if there exists $c \in super^*(class(i))$ such that $x \in instance\_template(c)$ and $val_i(x) = v$

Third Axiom of Inheritance

These are the only rules that apply.

3.3. From Delegation to Inheritance

Definition: Two models, $\mathbb{A}$ and $\mathbb{B}$, are *equivalent* iff for each object a in $\mathbb{A}$, there is a unique corresponding object b in $\mathbb{B}$ such that attribute x is valid for a with value v iff x is valid for b with value v.

Let Ψ be a function which takes a delegation model $\mathbb{D} = \{\ O,\ X,\ U\ \}$, and yields an inheritance model $\Psi(\mathbb{D}) = \{\ C, I, Y, W\}$, as follows:

$i)\ \Psi(\emptyset) = \emptyset$
$ii)\ \forall x \in X,\ \Psi(x) = x$
$iii)\ \forall v \in U,\ \Psi(v) = v$
$iv)\ \forall o \in O$, define $\Psi(o)$ as:

$$super(\Psi(o)) = \Psi(proto(o))$$
$$class\_attributes(\Psi(o)) = \bigcup_{x \in attributes(o)} \Psi(x)$$
$$V(\Psi(o)) = \bigcup_{v \in V(o)} \Psi(v),$$
$$i.e.\ \forall x \in class\_attributes(\Psi(o)),\ val_{\Psi(o)}(x) = val_o(x)$$

$v)\ \Psi(\mathbb{D}) = \{\ C, I, Y, W\}$, where

$$C = \bigcup_{o \in O} \Psi(o)$$
$$I = \emptyset$$
$$Y = X$$
$$W = U$$

Lemma 1. $\forall o \in O$, $x \in class\_attributes(\Psi(o))$ with $val_{\Psi(o)}(x) = v$ iff $x \in attributes(o)$ with $val_o(x) = v$.

Proof: This follows directly from the definition of $class\_attributes(\Psi(o))$. $Class\_attributes(\Psi(o)) = \bigcup_{x \in attributes(o)} \Psi(x) = \bigcup_{x \in attributes(o)} x = attributes(o)$. Since the sets are equal, their membership is the same. By the definition of Ψ, $\forall x \in class\_attributes(\Psi(o)),\ val_{\Psi(o)}(x) = val_o(x)$. □

Lemma 2. $\forall o \in O,\ c\_depth(\Psi(o)) = p\_depth(o)$.

Proof: *By Induction.*

Base Case: $C\_depth(\Psi(o)) = 0$ iff $super(\Psi(o)) = \emptyset$. But $super(\Psi(o)) = \Psi(proto(o)) = \emptyset$ iff $proto(o) = \emptyset$ iff $p\_depth(o) = 0$.

Induction Hypothesis: Assume that $c\_depth(\Psi(o)) = p\_depth(o)$ for all o with $p\_depth(o) \leq k$, for some $k \geq 0$.

Inductive Case: Consider some object $o \in O$, such that $p\_depth(o) = k + 1$. Then $p\_depth(proto(o)) = k$. By the induction hypothesis, this means that $c\_depth(\Psi(proto(o)) = p\_depth(proto(o)) = k$. But by the definition of c_depth, $super(\Psi(o)) = \Psi(proto(o))$, and $c\_depth(\Psi(o)) = c\_depth(super(\Psi(o))) + 1$. So $c\_depth(\Psi(o)) = k + 1$.

□

Theorem 1. For every delegation model, $\mathbb{D}$, there is an equivalent inheritance model, $\Psi(\mathbb{D})$

Proof: *By induction.* To show these models equivalent, we must demonstrate that for any object $o \in O$, an attribute x is valid for o with value v iff x is valid for $\Psi(o)$ with value v.

Base Case: If o is a root of the hierarchy, then $proto(o) = \Psi(proto(o)) = super(\Psi(o)) = \emptyset$. By Delegation, x is valid for o with value v iff $x \in attributes(o)$ and $val_o(x) = v$. By Lemma 1, this is true iff $x \in attributes(\Psi(o))$ and $val_{\Psi(o)}(x) = v$.

Induction Hypothesis: Assume that x is valid for o with value v iff x is valid for $\Psi(o)$ with value v for all o such that $p\_depth(o) \leq k$.

Inductive Case: Consider o such that $p\_depth(o) = k + 1$. Delegation describes two cases:

Case (i): $x \in attributes(o)$, and $val_o(x) = v$. By Lemma 1, this is true iff $x \in class\_attributes(\Psi(o))$, and $val_{\Psi(o)}(x) = val_o(x) = v$.

Case (ii): Case (i) fails, and x is valid for $proto(o)$ with value v. But $p\_depth(proto(o)) < p\_depth(o)$, so the induction hypothesis holds, and x is valid for $\Psi(proto(o)) = super(\Psi(o))$, with value v. Since Case (i) fails, $x \notin class\_attributes(\Psi(o))$; by Inheritance, this means that x is valid for $\Psi(o)$ with value v since it is valid for $super(\Psi(o))$ with value v.

□

3.4. From Inheritance to Delegation

In order to make the same proof work in the other direction, one additional definition needs to be made: Delegation does not allow attributes to be defined by one object, but have their values stored in another. Since this is exactly what instance templates do, any simulation of inheritance by delegation must somehow "fix" this problem. The solution chosen in this proof is one of several; however all share the property that the mapping only holds *up to instance templates*.

Definition: Two models, A and B, are *equivalent up to instance templates* iff for each object a in A, there is a unique corresponding object b in B such that attribute x is valid for a with value v iff x is valid for b with value v or there is some $a' \in \mathit{super}^*(a)$, such that $x \in \mathit{instance\_template}(a')$, and if b' corresponds to a', then $\mathit{val}_{b'}(x) = \#$, where $\#$ is some marker value not in U.

In addition to the restriction that delegation can simulate inheritance only *up to instance templates*, systems allowing only single-parent delegation can only simulate those inheritance systems in which $C \cap I = \varnothing$—all objects are either classes or instances, but not both. If an object in an inheritance hierarchy is both an instance—and so has a *class*—and a class—and so has a *super*—, then in the corresponding delegation hierarchy, it must have two parents: one representing its *class*, the other its *super*. It is, however, possible to extend the proof below to allow for both multiple inheritance and multiple-parent delegation.

Let Φ be a function which takes a delegation model $\mathrm{I} = \{\ C,\ I,\ Y,\ W\ \}$, and yields an inheritance model $\Phi(\mathrm{I}) = \{\ O,\ X,\ U\ \}$, as follows:

$i)$ $\Phi(\varnothing) = \varnothing$

$ii)$ $\forall x \in Y,\ \Phi(x) = x$

$iii)$ $\forall v \in W,\ \Phi(v) = v$

$iv)$ $\forall c \in C$, define $\Phi(c)$ as:

$$\mathit{proto}(\Phi(c)) = \Phi(\mathit{super}(c))$$

$$\mathit{attributes}(\Phi(c)) = \Big(\bigcup_{x \in \mathit{class\_attributes}(c)} \Phi(x)\Big) \cup \Big(\bigcup_{x \in \mathit{instance\_template}(c)} \Phi(x)\Big)$$

$$V(\Phi(c)) = \Big(\bigcup_{v \in V(c)} \Phi(v)\Big) \cup \Big(\bigcup_{x \in \mathit{instance\_template}(c)} \#\Big)$$

i.e. $\forall x \in \mathit{attributes}(\Phi(c))$,
$\mathit{val}_{\Phi(c)}(x) = \mathit{val}_c(x)$,
and $\forall x \in \mathit{instance\_template}(c)$,
$\mathit{val}_{\Phi(c)} = \#$,

$v)$ $\forall i \in I$, define $\Phi(i)$ as:

$$\mathit{proto}(\Phi(i)) = \Phi(\mathit{class}(i))$$

$$\mathit{attributes}(\Phi(i)) = \bigcup_{x \in \bigcup_{c \in \mathit{super}^*(\mathit{class}(i))} \mathit{instance\_template}(c)} \Phi(x)$$

$$V(\Phi(i)) = \bigcup_{v \in V(i)} \Phi(v),$$

i.e., $\forall x \in \mathit{attributes}(\Phi(i))$,
$\mathit{val}_{\Phi(i)}(x) = \mathit{val}_i(x)$

$vi)$ $\Phi(\mathrm{I}) = \{\ O,\ X,\ U\ \}$,
where

$$O = \Big(\bigcup_{c \in C} \Phi(c)\Big) \cup \Big(\bigcup_{i \in I} \Phi(i)\Big)$$

$$X = Y$$

$$U = W \bigcup \#$$

Lemma 3a. $\forall c \in C$, $x \in \mathit{attributes}(\Phi(c))$ with $\mathit{val}_{\Phi(c)}(x) = v$ iff $x \in \mathit{class\_attributes}(c)$ with $\mathit{val}_c(x) = v$, or $x \in \mathit{instance\_template}(c)$ and $\mathit{val}_{\Phi(c)}(x) = \#$.

Proof: $\mathit{Attributes}(\Phi(c)) - \mathit{instance\_template}(c)$
$= \bigcup_{x \in \mathit{class\_attributes}(c)} \Phi(x) = \bigcup_{x \in \mathit{class\_attributes}(c)} x = \mathit{class\_attributes}(c)$.
If $x \in \mathit{attributes}(\Phi(c)) - \mathit{instance\_template}(c)$, $\mathit{val}_{\Phi(c)}(x) = \mathit{val}_c(x)$. If $x \in \mathit{instance\_template}(c)$, then $x \in \mathit{attributes}(\Phi(c))$, and $\mathit{val}_{\Phi(c)}(x) = \#$.

□

Lemma 3b. $\forall i \in I$, $x \in \mathit{attributes}(\Phi(i))$ with $\mathit{val}_{\Phi(i)}(x) = v$ iff there exists $c \in \mathit{super}^*(\mathit{class}(i))$ such that $x \in \mathit{instance\_template}(c)$, and $\mathit{val}_i(x) = v$.

Proof: $\mathit{Attributes}(\Phi(i)) = \bigcup_{x \in \bigcup_{c \in \mathit{super}^*(\mathit{class}(i))} \mathit{instance\_template}(c)} \Phi(x)$
$= \bigcup_{x \in \bigcup_{c \in \mathit{super}^*(\mathit{class}(i))} \mathit{instance\_template}(c)} x = \bigcup_{c \in \mathit{super}^*(\mathit{class}(vi))} \mathit{instance\_template}(c)$.
By the definition of Φ, $\forall x \in \mathit{attributes}(\Phi(i))$, $\mathit{val}_{\Phi(i)}(x) = \mathit{val}_i(x)$.

□

Lemma 4. $\forall c \in C - I$, $\mathit{p\_depth}(\Phi(c)) = \mathit{c\_depth}(c)$.

Proof: *By Induction.*

Base Case: $\mathit{P\_depth}(\Phi(c)) = 0$ iff $\mathit{proto}(\Phi(c)) = \varnothing$. But $\mathit{proto}(\Phi(c)) = \Phi(\mathit{super}(c)) = \varnothing$ iff $\mathit{super}(c) = \varnothing$ iff $\mathit{c\_depth}(c) = 0$.

Induction Hypothesis: Assume that $\mathit{p\_depth}(\Phi(c)) = \mathit{c\_depth}(c)$ for all $c \in C - I$ with $\mathit{c\_depth}(c) \leq k$, for some $k \geq 0$.

Inductive Case: Consider some object $c \in C - I$, such that $\mathit{c\_depth}(c) = k + 1$. Then $\mathit{c\_depth}(\mathit{super}(c)) = k$. By the induction hypothesis, this means that $\mathit{p\_depth}(\Phi(\mathit{super}(c)) = \mathit{c\_depth}(\mathit{super}(c)) = k$. But by the definition of $\mathit{p\_depth}$, $\mathit{p\_depth}(\Phi(c)) = \mathit{p\_depth}(\mathit{proto}(\Phi(c))) + 1$, and $\mathit{proto}(\Phi(c) = \Phi(\mathit{super}(c))$. So $\mathit{p\_depth}(\Phi(c)) = k + 1$.

□

Theorem 2. For every inheritance model, I, with $C \cap I = \varnothing$, there is an equivalent inheritance model, $\Phi(\mathrm{I})$, up to instance templates.

Proof: To show these models equivalent up to instance templates, we must demonstrate that for any object $a \in C \cup I$, an attribute x is valid for a with value v iff x is valid for $\Phi(a)$ with value v, or $x \in \mathit{instance\_template}(a)$ and $val_{\Phi(a)}(x) = \#$.

Instances:

$\forall i \in I$, x is valid for i with value v iff there exists $c \in super^*(class(i))$ such that $x \in \mathit{instance\_template}(c)$ and $val_c(x) = v$ (by Inheritance) iff x is valid for $\Phi(i)$ with value v (by Lemma 3b).

Classes:

Base Case: If c is a root of the hierarchy, then $super(c) = \Phi(super(c)) = proto(\Phi(c)) = \emptyset$. By Delegation, x is valid for $\Phi(c)$ with value v iff $x \in attributes(\Phi(c))$ and $val_{\Phi(c)}(x) = v$. By Lemma 3a, this is true iff $x \in \mathit{class\_attributes}(c)$ and $val_c(x) = v$, or $x \in \mathit{instance\_template}(c)$ and $val_{\Phi(c)}(x) = \#$.

Induction Hypothesis: Assume that x is valid for $\Phi(c)$ with value v iff either x is valid for c with value v or there is a $c' \in super^*(c)$ for which $x \in \mathit{instance\_template}(c')$ and $val_{\Phi(c')}(x) = \#$, for all c such that $\mathit{c\_depth}(c) \leq k$.

Inductive Case: Consider c such that $\mathit{c\_depth}(c) = k + 1$. Inheritance allows two possibilities:

Case (i): $x \in attributes(\Phi(c))$, and $val_{\Phi(c)}(x) = v$. By Lemma 3a, this is true iff $x \in \mathit{class\_attributes}(c)$ with $val_c(x) = v$ or $x \in \mathit{instance\_template}(c)$ and $val_{\Phi(c)}(x) = \#$.

Case (ii): Case (i) fails, and x is valid for $proto(\Phi(c))$ with value v. But $\mathit{p\_depth}(proto(\Phi(c))) < \mathit{c\_depth}(c)$, so the induction hypothesis holds, and either x is valid for $\Phi(super(c)) = proto(\Phi(c))$, with value v, or there is a $c' \in super^*(super(c))$ for which $x \in \mathit{instance\_template}(c')$. $Super^*(super(c)) \subseteq super^*(c)$, so $c' \in super^*(c)$, and the instance template condition holds. Otherwise, since Case (i) fails, $x \notin attributes(\Phi(c))$; by Delegation, this means that x is valid for $\Phi(c)$ with value v since it is valid for $proto(\Phi(c))$ with value v.

□

4. Conclusions and Extensions

4.1. Limitations on the Proof

In discussing the features of inheritance and delegation, I highlighted four ways in which the mechanisms differed. To wit:

(1) *Incremental definition:* inheritance allows it only on classes; delegation, on all objects.

(2) *Sharing of attributes:* inheritance allows sharing of class attributes, but only instance methods; delegation, all attributes of all objects.

(3) *Dependence of instances:* inheritance forbids it; delegation allows it.

(4) *Grouping by type:* inheritance requires it; delegation does not.

It should now be apparent that all four of these distinctions depend on the instance template. It is the instance template which forbids one instance to be defined in terms of the other—all instances are defined in terms of the template. The instance template declares the instance attributes, allowing them to be shared by all instances, but does not store their values. Since instances are required to have their own, private values for all attributes, this disallows sharing of attributes and prevents dependent instances. Finally, the instance template guarantees the structure of all instances of a class, ensuring certain grouping properties of the hierarchy.

But it is precisely this instance template that is lost in the translation from inheritance to delegation; indeed, it is this instance template that delegation lacks. This is both good and bad, in that it allows delegation a flexibility not present in inheritance, but prevents delegation from providing any structural guarantees on the elements in the hierarchy. In the model of inheritance presented above, the class structure provides all that delegation provides; the instance structure adds rigidity.

4.2. The Hybrid Model

Depending on the needs of the application, certain properties of inheritance and delegation may be more or less desirable. For example, grouping properties provide typing guarantees that are useful for compilation and database indexing. On the other hand, the fast copy-creation and small object size of delegation can provide important runtime speedups in a system with rapidly changing objects.

One solution is to provide all the mechanisms of both, and to let the user pick and choose among them. The formal model for inheritance presented above does this, in that the objects called "classes" delegate, while those called "instances" inherit. However, these names and their common associations make it harder to see how rich this system is.

In order to take full advantage of the potential of such a system, objects must be treated simply as objects, rather than as classes or instances. In this model, $C = I$—all objects are both classes and instances, although some may have $\mathit{instance\_template} = \emptyset$.[1] Three new operations can now be defined in terms of the inheritance mechanisms available in the formal model above.

[1] This is in fact what the world looks like after applying Φ to make it delegation, then returning it (through Ψ) to inheritance.

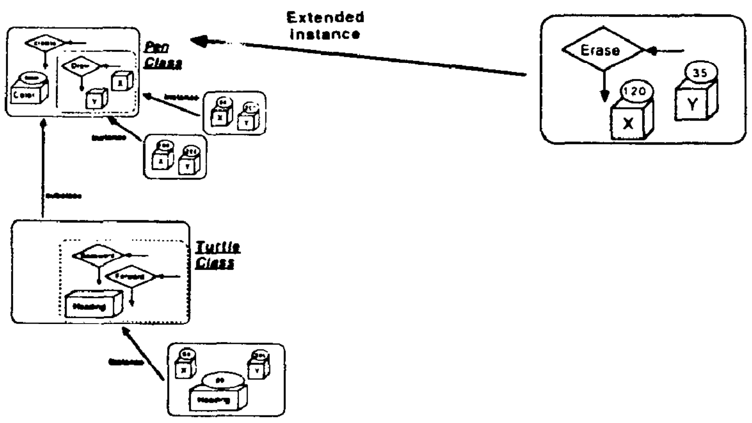


Figure 4. Minimal guarantee.

Suppose, as in figure 4, that we discover an object much like a *pen*, but with an extra *Erase* attribute. This object can be made an "instance" of "class" *pen*; its extra attribute is simply added as a *class_attribute*. This transforms the *instance_template* into a *minimal_template*: rather than guaranteeing the exact shape of an instance, it guarantees a minimal shape. The aspects in the minimal template can still be typechecked and indexed, but not all attributes of the "instance" need be in the template of the class. This avoids the need for creation of extraneous classes every time some object does not precisely fit the definition of its class.

If at some later time many more *pens* are found with an *Erase* attribute, they may simply be added to the hierarchy as extensions to the "prototype" *Erase*-able *pen*. This ***accumulation*** of extensions corresponds to objects delegating to a single prototype, or many subclasses of one class. It is illustrated in figure 5. The new *Erase*-able *pens* may depend on the original one, or not, as the case may be. In either case, the minimal guarantees of the template hold for these objects to the extent that they either possess the attributes or share their prototype's copy.

At some point, this may become insufficient, and a new class may need to be created. In this case, the "prototype" can be ***promoted*** to a class, shifting attributes into the template and copying down values as necessary. The class-instance relationship becomes a class-subclass relationship, and a new instance is created to take the place of the one promoted to class. The result is shown in figure 6.

These extensions to inheritance are natural outgrowths of a new way of looking at objects in a hierarchy.

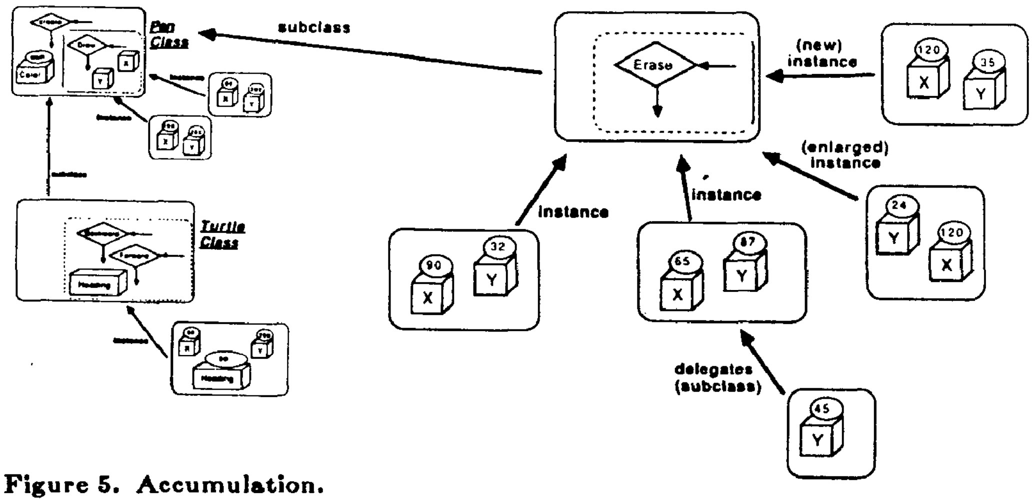


Figure 5. Accumulation.

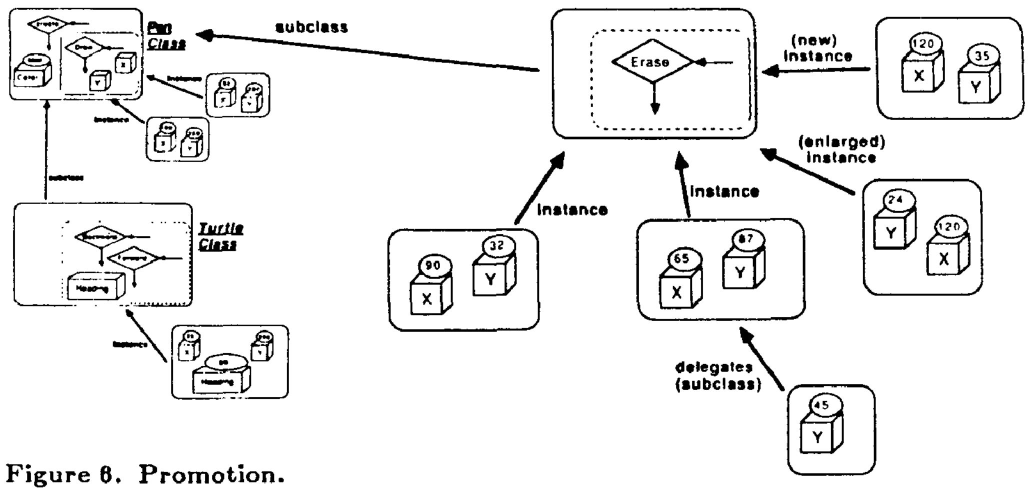


Figure 6. Promotion.

If strict subtyping is observed, the traditional "class-subclass" relationship precisely captures delegation. The class-instance relationship is distinguished by the presence of an instance template, which provides independence of value and typing guarantees. These two relationships combine to allow the hierarchy to take on unusual and useful shapes. By disregarding traditional naming conventions, "classes" can be treated as delegation objects, while instance templates can be used to ensure minimal consistency. The dynamism of delegation and the structured behavior of inheritance can be merged in a flexible but regulated way.

5. Acknowledgements

Many thanks are due to Peter Wegner and Stan Zdonik, without whom this paper could not have been written, and to Scott Meyers, Richard Hughey, Mark Hornick, Maurine Neiberg, and Jim Bloom, for their help and patience.

References

[Borning]
Borning, A. H., "Classes Versus Prototypes in Object-Oriented Languages." *ACM/IEEE Fall Joint Computer Conference,* November 1986.

[Briot & Yonezawa]
Briot, J., and A. Yonezawa. "Inheritance Mechanisms in Object-Oriented Concurrent Languages." Extended abstract. 1987.

[Cardelli & Wegner]
Cardelli, L., and P. Wegner. "On Understanding Types, Data Abstraction, and Polymorphism." *Computing Surveys,* August 1986.

[Goldberg & Robson]
Goldberg, A., and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[LaLonde]
LaLonde, W. "An Exemplar Based Smalltalk." *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* September 1986.

[Lieberman]
Lieberman, H. "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems." *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* September 1986.

[Stefik & Bobrow]
Stefik, M., and D. Bobrow. "Object-Oriented Programming: Themes and Variations." *AI Magazine,* December 1985.

[Touretsky]
Touretsky, D. *The Mathematical Theory of Inheritance.* Morgan-Kaufman, 1986.

[Wegner & Zdonik]
Wegner, P., and S. Zdonik. "Why Like Isn't Like Isa." Brown University Technical Report, March 1987.