Chapter 3

A Shared View of Sharing: *The Treaty of Orlando*

Lynn Andrea Stein

Henry Lieberman

David Ungar

Introduction

For the past few years, researchers have been debating the relative merits of object-oriented languages with classes and inheritance as opposed to those with prototypes and delegation. It has become clear that the object-oriented programming language design space is not a dichotomy. Instead, we have identified two fundamental mechanisms—*templates* and *empathy*—and several different independent degrees of freedom for each. Templates create new objects in their own image, providing guarantees about the similarity of group members. Empathy allows an object to act as if it were some other object, thus providing sharing of state and behavior. The Smalltalk-80™ language,[1] Actors, Lieberman's DELEGATION system, SELF, and HYBRID each take differing stands on the forms of templates

[1]Smalltalk-80™ is a trademark of ParcPlace Systems. In this chapter, the term "Smalltalk" will be used to refer to the Smalltalk-80™ programming language.

and empathy.

Some varieties of template and empathy mechanisms are appropriate for building well-understood programs that must be extremely reliable, while others are better suited for the rapid prototyping of solutions to difficult problems. The differences between languages designed for each of these application domains can be recast as the differences between support for anticipated *vs.* unanticipated sharing. One can even ascribe the ascent of object-oriented programming to its strong support for extension instead of modification. However, there are still many kinds of extension that remain difficult. The decomposition of an object-oriented language into template and empathy mechanisms, and the degree of support for extension provided by the forms of these mechanisms comprise a solid framework for studying language design.

We begin this chapter with the text of our "Treaty," in which we outline the basis for our consensus. In section 3.2, we discuss the differences between *anticipated* and *unanticipated* sharing. Section 3.3 defines more formally the fundamental terms and concepts we have identified. Several languages representing different paradigms are examined as examples of these mechanisms in section 3.4. Finally, section 3.5 describes the larger issues of software evolution which underly the issues raised here; in this context, our conclusions can be seen as a partial solution to the more general problems of sharing.

3.1 *The Treaty*

Mechanisms for sharing knowledge and behavior between objects are among the most useful and also the most hotly debated features of object-oriented languages. The three authors of this chapter have previously published papers in which new sharing mechanisms for object-oriented languages were prominently featured. We met on the occasion of *OOPSLA '87*, in Orlando, Florida, and discovered through discussion that we shared a common outlook that both clarifies the reasons for design choices made in previous languages, and also points the way toward future research in this area. We call this consensus the *Treaty of Orlando*.[2]

WHEREAS the intent of object-oriented programming is to provide a natural and straightforward way to describe real-world concepts, allowing the flexibility of expression necessary to capture the variable nature of the world being modeled, and the dynamic ability to represent changing situations; and

WHEREAS a fundamental part of the naturalness of expression provided by object-oriented programming is the ability to share data, code, and definition, and to this end all object-oriented languages provide some way to define a new object in terms of an existing one, borrowing implementation as well as behavioral description from the previously defined object; and

WHEREAS many object-oriented languages—beginning with Simula-67, and including Smalltalk, Flavors, and Loops—have implemented this sharing through classes, which allow one group

[2]The original treaty text appears in [Power and Weiss, 1988].

of objects to be defined in terms of another, and also provide guarantees about group members, or instances; and

WHEREAS these mechanisms—class, subclass, and instance—impose a rigid type hierarchy, needlessly restricting the flexibility of object-oriented systems, and in particular do not easily permit dynamic control over the patterns of sharing between objects; which dynamic control is particularly necessary in experimental programming situations, where the evolution of software can be expected to proceed rapidly; and

WHEREAS the signatories to this treaty have independently proposed seemingly disparate solutions to this problem, to wit:

[Lieberman, 1986] proposed that traditional inheritance be replaced by delegation, which is the idea that sharing between objects can be accomplished through the forwarding of messages, allowing one object to decide at runtime to forward a message to another, more capable object, and giving this new object the ability to answer this message on the first (delegating) object's behalf; in this scheme, prototypical objects—the "*typical* elephant," for example—replace abstract classes—*e.g.* the *class* elephant—as the repository for shared information;

[Ungar and Smith, 1987] also proposed a prototype-based approach, using a drastic simplification of the Smalltalk model in which a single type of parent link replaces the more complex class/subclass/instance protocol; while this approach does not propose explicit delegation, through "dynamic inheritance" it shares the essential characteristics of allowing dynamic sharing patterns and idiosyncratic behavior of individual objects;

[Stein, 1987] attempted a rapprochement between the delegation and inheritance views, pointing out that the class/subclass relationship is essentially this "delegation," or "dynamic inheritance," and that these new styles of sharing simply make a shift in representation, using what were previously considered "classes" to represent real-world entities rather than abstract groups; this approach gives a different way of providing idiosyncratic behavior and dynamic sharing, through extensions to the class-instance relationship;

WHEREAS the signatories to this treaty now recognize that their seemingly divergent approaches share a common underlying view on the issues of sharing in object-oriented systems, we now declare:

RESOLVED, that we recognize two fundamental mechanisms that sharing mechanisms for object-oriented languages must implement, and that can be used for analyzing and comparing the plethora of linguistic mechanisms for sharing provided by different object-oriented languages: The first is *empathy*, the ability of one object to share the behavior of another object without explicit redefinition; and the second is the ability to create a new object based on a *template*, a "cookie-cutter" which guarantees, at least in part, characteristics of the newly created object.

RESOLVED, that most significant differences between sharing mechanisms can be analyzed as making design choices that differ along the following three independent dimensions, to wit:

First, whether *STATIC* or *DYNAMIC*: When does the system require that the patterns of sharing be fixed? Static systems require determining the sharing patterns by the time an object is created, while dynamic systems permit determination of sharing patterns when an object actually receives a message; and

Second, whether *IMPLICIT* or *EXPLICIT*: Does the system have an operation that allows a programmer to explicitly direct the patterns of sharing between objects, or does the system do this automatically and uniformly? and

Third, whether *PER OBJECT* or *PER GROUP*: Is behavior specified for an entire group of objects at once, as it is with traditional classes or types, or can idiosyncratic behavior be attached to an individual object? Conversely, can behavior be specified/guaranteed for a group?

RESOLVED, that no definitive answer as to what set of these choices is best can be reached. Rather, that different programming situations call for different combinations of these features: for more exploratory, experimental programming environments, it may be desirable to allow the flexibility of dynamic, explicit, per object sharing; while for large, relatively routine software production, restricting to the complimentary set of choices—strictly static, implicit, and group-oriented—may be more appropriate.

RESOLVED, that as systems follow a natural evolution from dynamic and disorganized to static and more highly optimized, the object representation should also have a natural evolutionary path; and that the development environment should itself provide more flexible representations, together with tools—ideally automatic—for adding those structures (of class, of hierarchy, and of collection, for example) as the design (or portions thereof) stabilizes.

and RESOLVED, that this agreement shall henceforth be known as the TREATY OF ORLANDO.

3.2 Anticipated *vs.* Unanticipated Sharing

In this chapter, we distinguish between two kinds of sharing that arise in object-oriented systems; or rather, two kinds of motivations for introducing sharing into an object-oriented system. The distinction between these is an important determiner of preference among object-oriented language mechanisms. One is **anticipated** sharing. During the conceptual phase of system design, before actual coding starts, a designer can often foresee commonalities between different parts of the system, leading to a desire to share procedures and data between those similar parts. This is best accomplished by language mechanisms which provide a means for the designer to write down the anticipated structure to be shared by other components. In traditional Simula-like object-oriented languages, classes serve as the mechanism for encoding anticipated sharing of behavior, which may be utilized by a perhaps unanticipated number of instances.

In contrast, **unanticipated** sharing is less well served by traditional inheritance mechanisms. Unanticipated sharing arises when a designer would like to introduce new behavior into an object system that does not already provide for it, and may not have been foreseen when the original system was programmed. The designer may notice that new behavior can be accomplished, in part, by making use of already-existing components, though procedures and data may have to be added or amended as well. Thus, a sharing relationship arises among components that are used in common for both their original, anticipated purposes, and their new, unanticipated purposes. Obviously, since the new behavior has not been anticipated, being forced to state the sharing relationships in advance puts a restriction on the kinds of new behavior that can be introduced without modifying the previous system. The traditional class-subclass-instance mechanism requires textually distinguishing, in a static way, between those elements intended as common behavior, namely classes, and those expected to be idiosyncratic, the instances.

Supporting unanticipated sharing is important since software evolution often follows unpredictable paths. A language mechanism supports unanticipated sharing best if new behavior can be introduced simply by explaining to the system what the differences are between the desired new behavior and the existing old behavior. Delegation, or dynamic inheritance, accomplishes this by allowing new objects to re-use the behavior of existing ones without requiring prior specification of this relationship.

The examples in [Lieberman, 1986] stress the advantages of delegation in situations where reasonable behavioral extensions to a system are unlikely to be anticipated in the

Figure 3.1: A `dribble stream` records interaction on a previously implemented `terminal stream`.

original design of a system. It is reasonable to want to define a "`dribble stream`" to record interaction on previously implemented `I/O` *input-output* `streams`, but it is unreasonable to require that the implementors of the original `I/O streams` have prepared in advance for the existence of `dribble streams`. On the other hand, it is reasonable to expect that a designer who first implements a `stream` to an interactive terminal build it upon some object representing the abstract notion of "an `I/O stream`", anticipating that there will eventually be some other types of streams, such as `disk` or `network streams`. It would be silly to implement a `disk stream` by delegating to a `terminal stream` for common operations simply because the `terminal stream` happened to have been implemented first. This illustrates the difference between anticipated and unanticipated sharing. It is primarily an issue of software evolution and design aesthetics, and only indirectly a language issue.

The main result of [Stein, 1987] can be rephrased in these terms: because a subclass is defined by stating the differences, in both procedures and data, between it and its superclass, the relation between subclass and superclass is better suited toward unanticipated sharing than the class-instance relation, which limits the differences between an instance and its class to the values of its variables.

3.3 Basic Mechanisms

The arguments over what is fundamental in object oriented programming have existed for as long as the field. Which features—classes, prototypes, inheritance, delegation, message passing, encapsulation, abstraction—are at the heart of object oriented programming, and how these things relate to one another, are not issues that will soon be resolved. Rather than try to settle this debate, we present here two mechanisms which are used in most object oriented languages: **empathy** and **templates**. We claim that they are fundamental in that they *cannot* be defined in terms of one another, and that most object oriented languages *can* be described largely in terms of the ways in which they combine these mechanisms.

The first of these mechanisms underlies both inheritance and delegation. In all languages accepted as object oriented there is some way in which one object can "borrow" an attribute—variable or method—from another. We propose to use the term **empathy** for this behavior:

We say that object $\mathcal{A}$ empathizes with object $\mathcal{B}$ for message $\mathcal{M}$ if $\mathcal{A}$ doesn't have its

Figure 3.2: The `pen` at `(100,200)` *empathizes* with the `pen` at `(50,200)` for its `Y` variable, and for the `Draw` method.

own protocol for responding to $\mathcal{M}$, but instead responds to $\mathcal{M}$ as though it were borrowing $\mathcal{B}$'s response protocol. $\mathcal{A}$ borrows just the response protocol, but not the rest of $\mathcal{B}$. That is, any time $\mathcal{B}$'s response protocol requires a message to be sent to `SELF` (or a variable to be looked up), it is sent to $\mathcal{A}$, not to $\mathcal{B}$; otherwise, $\mathcal{A}$ and $\mathcal{B}$ respond in the same way. For example, in Fig. 3.2, the `pen` at `(100,200)` *empathizes* with the `pen` at `(50,200)` for its `Y` variable, and for the `Draw` method.[3]

[3] Formally, we say that object $\mathcal{A}$ *empathizes* with object $\mathcal{B}$ for $\mathcal{M}$ when the following holds: If $\mathcal{B}$'s behavior in response to $\mathcal{M}$ can be expressed as a function $\psi(\mathcal{B}, \mathcal{M})$—that is, $\mathcal{B}$'s method for $\mathcal{M}$ can be expressed as a function that takes `SELF` as an argument along with $\mathcal{M}$—then $\mathcal{A}$'s response to $\mathcal{M}$ can be expressed using the same function ψ as $\psi(\mathcal{A}, \mathcal{M})$—$\mathcal{A}$'s behavior is derived by using $\mathcal{A}$ wherever $\mathcal{B}$ would have used itself. The symmetry in this behavioral definition of empathy may seem counterintuitive, but such symmetry is

All incarnations of inheritance and delegation include empathy; they differ as to *when* and *how* the relationships are determined. Empathy may be explicit: "Execute `thisObject:thisRoutine` in my environment," as in the ability of CommonLoops to specialize method lookup. It may be by default: "Anything I can't handle locally, look up in `myParent` (and execute in my environment) with `SELF = me`," as in Smalltalk, SELF, and **C++**. It may be dynamic or static, per object or per group. These language choices are responsible for much of the variety of existing object oriented paradigms.

Inheritance, as found in Simula and Smalltalk, is the preprogrammed determination of default delegation paths, by group. It requires the generation of uniform groups of objects. It separates the delegatable (traditionally, methods only) from the non-delegatable (traditionally, the instance variables). The delegatable part is stored in the class; the non-delegatable is necessarily allocated independently for each instance.

But it is perhaps the interaction of delegation with the second fundamental mechanism, that of **templates**, which determines the most interesting and controversial distinctions between types of object-oriented languages. A template is a kind of "cookie cutter" for objects: it contains all the method and variable definitions, parent pointers, *etc.*, needed to define a new object of the same type. If the object may not gain or lose attributes once it is defined, we call the template **strict**.

---

inherent in any behavioral definition. All a behavioral definition can do is say that the behaviors of two objects are similar according to some criterion; you can't tell "who did the implementing" unless you look into the code of the implementation.

In many languages, after the template is copied, or "instantiated," changes to this new object may be permissible, weakening the guarantee of uniform behavior by group. These variations on strict templates are discussed below. In addition, there are languages without templates; however, in these languages, such as Lieberman's DELEGATION, the system provides no inherent concept of "group," or "kind," of object.

In some languages, such as SELF and Actra, the template is itself an object. In others, it is embedded in another, generator object, usually called a **class**. A class is an object of one type which contains a template for objects of another type. Thus, `class elephant` is an object of type `Class`, but contains a template for objects of type `Elephant`. The objects cut from the template embedded in a class are known as its **instances**.

Traditionally, this class-instance relationship is strict: a **strict** (instance) **template** lists exactly those attributes that each object cut from that template must define, and no cookie-cut object can define attributes other than those in the template. Because the template is strict, each object cut from it will have a local copy of each attribute; these attributes cannot be redefined or removed; therefore they will never be delegated. A class thus guarantees the uniformity and independence of its "cookie-cut" instances.

However, this relationship can be relaxed in several ways. For example, a **minimal template** is a cookie cutter in the same sense, but once created, cookie-cut objects can define other attributes as well. An extended instance—one generated by a minimal template, then added to—does not, *a priori*, have a template for its type. Its descendants cannot be strict instances, since there is no template for their type. On the other hand, the extended

	Determination of Empathy			Template Mechanisms	
Language	when	how	for	what	how
Actors	runtime	explicit	per object	none	
DELEGATION	runtime	both	per object	none	
SELF	runtime	implicit[4]	per object	templates	nonstrict
Simula	compile time	implicit	per group	classes	strict
Smalltalk	object creation time	implicit	per group	classes	strict
HYBRID	runtime	both	both	any	nonstrict

Table 3.1: Various languages and their attributes.

instance can be **promoted**, transforming it into a class, of type `Class` but with a template for its original type. This class may then have instances.

Other relaxations in template enforcement create a variety of non-strict templates. In languages where templates are themselves objects, templates are often entirely non-strict. That is, an object may be created from a template, but subsequently go on to add or delete attributes, transforming it from a copy of its template into a new type of object, as in SELF.

3.4 Some Case Studies

In this section, we describe several languages which exemplify three of the language paradigms we have identified. The first paradigm is the least constrained; Actors and DELEGATION are almost purely dynamic empathy systems which support the maximally flexible set of choices. SELF adds to this the concept of template, making grouping of objects possible. However, SELF does not have classes, and remains a fully dynamic and flexible language. The third paradigm is the "classical" style of object-oriented programming found in Simula and Smalltalk. While traditionally this paradigm has been used in a rigid and inflexible manner, we describe one language—HYBRID—which maintains the structure of a class-based system while allowing much of the flexibility of the previous two paradigms. A summary of language features is given in Table 3.

3.4.1 Actors and Lieberman's DELEGATION

The actor systems of Hewitt and his colleagues at MIT represent the most extreme orientation towards dynamic and flexible control. The basic actor model provides only for active objects and parallel message passing [Agha, 1987] and so mandates no particular sharing mechanism. However, actual actor implementations [Lieberman, 1987] have found it most natural to use delegation as the sharing mechanism, since the actor philosophy encourages using patterns of message passing to express what in other languages would require special-purpose mechanisms [Hewitt, 1984].

Along the three dimensions of our treaty, actor systems can be classified as dynamic, explicit, and per-object. Sharing mechanisms in actors are dynamic, since message passing is a run-time operation, invoked without prior declaration. Delegation requires explicit designation of the recipient. Delegation is accomplished through a special message-passing protocol that includes the client (the equivalent of the `SELF` variable in Smalltalk-like languages) as part of the message. Since actor systems have no notion of type, sharing must be specified on a per-object basis. There are no templates, class or instantiation mechanisms defined in the kernels of actor languages. Of course, nothing precludes the use of delegation and object creation operations in actor systems to implement templates, or objects representing classes or sets.

The major conceptual difference between actors and the Simula family of languages arises in what is considered fundamental. In the traditional Simula-like languages, mechanisms of class, subclass and instantiation are considered fundamental. The behavior of the message passing operation is explained in terms of them and their influence on variable and procedure lookup. In actor systems, the message passing operation is considered fundamental. Even variable lookup must be explained in terms of sending messages to an object representing the environment. Thus, sharing mechanisms in actor systems are built on top of message passing.

DELEGATION is an outgrowth of the actors languages. As such, it has no templates. Object creation is independent of the parent: a new object is created by making a new empty object which points to some other object (or doesn't, as is desired), and then filling in the details of what that object should contain. Thus, an object with no attributes but a `name` could be a child of (and therefore delegate to) an `elephant` or an `employee` or a `real`.

Since there are no "class defining objects," there are no "classes" or groups of objects of the same "type." Every object determines its own, unique "type." There is thus no distinc-

Figure 3.3: A DELEGATION hierarchy.

tion between creating a new object, and creating a new "type" of object. In DELEGATION, everything is done on an object-by-object basis. DELEGATION is entirely dynamic; anything can change at any time. Empathy in DELEGATION can be either hierarchical (implicit) or explicit. Figure 3.3 shows a DELEGATION hierarchy.

3.4.2 SELF

SELF was designed to aid in exploratory programming by optimizing expressiveness and malleability. It is essentially a template-based language; however, SELF templates are as much a matter of convention as of language design. New objects are created by cloning an existing ones; the original object—called a prototype—behaves in much the same way as the standard template described above. In Fig. 3.4, a new `elephant` (Fred) has been created

Figure 3.4: A SELF hierarchy.

by copying an existing `elephant` (Clyde). Clyde is thereby functioning as a prototype, or template.

SELF templates are non-strict, so the objects they create can extend or otherwise modify their template-defined properties. Some attributes in the child may simply be delegated to the parent object, while others may be handled locally or delegated elsewhere. The new object may also have additional attributes not defined for the template, creating a sort of "extended instance." New *kinds* of objects are created by making a new template—cloning and then modifying an existing object—so that it has the requisite properties. In this way, a sort of "subclass" behavioral inheritance can be created.[5] Thus, one can take an elephant

[5] It is worth noting that there is no distinction between the concepts of "extended instance" and "subclass" in this kind of language, since any extended instance is also a potential template for a new "type" of object.

and add big ears and the ability to fly. This elephant (Dumbo) is unique: there is no exact template for it.

The patterns of empathy in SELF are determined individually by each object. An object's `parent` slots list the objects it empathizes with. In the example the `elephants` empathize with an object holding shared behavior—`walking`, `eating`, *etc.*—for `elephants`. Since an object may change the contents of its slots whenever it wishes, the patterns of empathy can change dynamically. Finally, since the `parent` attribute is part of a slot, the patterns of empathy are implicit, in the attributes and contents of an object's slots, not explicitly in the code.[6] SELF's non-strict objectified templates, and its individual, dynamic, and implicit patterns of empathy foster exploratory programming.

3.4.3 "Standard Inheritance", and HYBRID

Standard inheritance, exemplified by Smalltalk and all Simula-based languages, consists of class objects, which *contain* templates and therefore can generate instances, and the instances generated by these classes. New objects are simply cut from the template: every variable must be allocated individually for each object, while methods are shared through the template. New class, or generator, objects "inherit" the templates of their superclasses. This is operationally equivalent to delegating part of the template. These classes may themselves be instances. In this case, the metaclass contains a template for an object (the class) which *itself* contains a template.

[6] A limited form of explicit delegation is allowed but rarely used.

Figure 3.5: A Smalltalk hierarchy.

In standard inheritance, all instances of a class fit exactly the template description. Once created, these objects retain their properties forever: each subclass must delegate to its specified superclass(es); each instance remains a member of its class for all time. The objects in Fig. 3.5 reflect this; in order to create Dumbo, the `flying elephant`, a new class—with a single instance—had to be created.

HYBRID is a system which allows the traditionally static and strict relationships of standard inheritance to be dynamic and flexible. There is, after all, no inherent reason why all these relationships must be strict. A HYBRID template, though embedded in a class, behaves more like SELF's templates, allowing objects generated from this class-template to add or delete attributes. Thus, in Fig. 3.6, the unique `flying elephant` is just an extended instance of the `class elephant`. Of course, if `flying elephants` were common, Hybrid does not preclude the creation of a new class—in fact, the language will generate such a class *automatically* from a prototypical instance. In addition, Hybrid inheritance is dynamic, allowing runtime changes to the hierarchy, and instances are not distinguished from classes, allowing them to explicitly delegate—or share—attributes.

Figure 3.6: A Hybrid hierarchy.

3.5 Sharing and Software Evolution

The issue of what kind of behavioral extensions to a system can be accomplished without modifying previously existing code is of central importance. An important principle is that a conceptually small extension to the behavior of a system of objects should be achievable with a small extension to the code. The analysis of alternative mechanisms for sharing should proceed by considering their effects upon the necessity for future modifications of the code to accomplish behavioral extensions.

There are two kinds of changes that we perform in object systems. One is adding new code, or extending the system. The other is editing previously existing code, or modifying the system. These two kinds of changes have very different effects in the programming environment.

Adding new code to extend a system is good. It preserves the previous state of the system; at worst one can simply delete the extension to return to a previous state. Editing code is a much more problematic transformation. It is a destructive operation, both literally and figuratively. As a side effecting operation, editing code destroys irrecoverably the previous state of the system, unless careful backup/undo operations are performed. Perhaps worse is the propensity of editing operations to introduce inconsistencies in a system. Often, behavioral extensions are accomplished by editing several pieces of code in different places, and performing these operations manually leaves the possibility that not all the edits will be performed in a consistent manner.

In fact, one can recast the whole object-oriented enterprise in terms of the extension/modification dichotomy. The true value of object-oriented techniques as opposed to conventional programming techniques is not that they can do things the conventional techniques can't, but that they can often extend behavior by adding new code in cases where conventional techniques would require editing existing code instead. Objects are good because they allow new concepts to be added to a system without modifying previously existing code. Methods are good because they permit adding functionality to a system without modifying previously existing code. Classes are good because they enable using the behavior of one object as part of the behavior of another without modifying previously existing code.

In a conventional language, we might implement a data representation for a geometric shape as a list of points. A display procedure for this representation might dispatch on the kind of shape to more specialized procedures as follows:

```
To DISPLAY a SHAPE:

 * If the shape is a TRIANGLE, call DISPLAY-TRIANGLE.

 * If the shape is a RECTANGLE, call DISPLAY-RECTANGLE,

                        .

                        .

                        .

 * Otherwise, cause an UNRECOGNIZED-SHAPE error.
```

```
Define A-TRIANGLE to be the list of points (100, 100), (-50 200), (150 -20).
```

The kind of shape could be recognizing by appending a tag onto the list of points, or perhaps even by examining the length of the list.

We can now ask the question: What do we have to do to add a new shape to the display procedure? In the conventional system, this involves destructively editing the code to insert a new conditional clause:

```
To DISPLAY a SHAPE:
 * If the shape is a TRIANGLE, call DISPLAY-TRIANGLE.
 * If the shape is a PENTAGON, call DISPLAY-PENTAGON
                       .
                       .
                       .
 * Otherwise...
```

The editing process leaves open the possibility for inconsistent edits, inadvertent deletion of old code, mismatch between protocols for using the data representation in the old and new clauses, etc.

In an object oriented language, by contrast, the representation modularizes the addition of each new object and message so that adding a new object or method can be done without any modification of previously existing code.

```
If I'm a SHAPE object, and I get a DISPLAY message,
```

```
 * I respond with an UNRECOGNIZED-SHAPE error.


Define a TRIANGLE to inherit from SHAPE.


If I'm a TRIANGLE and I get a DISPLAY message,

 * I respond with DISPLAY-TRIANGLE.


Define A-TRIANGLE to be an instance of SHAPE

 * With vertices (100, 100), (-50 200), (150 -20).
```

Now, we can extend the system to know about pentagons simply by adding a new definition which extends the system.

```
Define a PENTAGON to inherit from SHAPE.


If I'm a PENTAGON and I get a DISPLAY message,

 * I respond with DISPLAY-PENTAGON.
```

One way of characterizing the themes common to our three original papers is that we observed that the implementation of unanticipated sharing between objects in Simula-style inheritance systems often required modification of existing code. We were searching for ways of implementing unanticipated behavioral extensions without modifying existing code, and concluded that the solution was to allow more dynamic forms of empathy. At the same time, we wish not to minimize the importance of language mechanisms for traditional, anticipated sharing and believe future languages must seek a synthesis of the two.

The search for ways to accomplish interesting behavioral extensions of object-oriented systems by additive extensions to code is far from over. If we can find any situations where a conceptually simple extension to the behavior of an object system seems to require gratuitous modification of existing code, it's the sign of a problem, and we ought to be looking for a solution. We give an example of one such situation, as a guide for future research.

Occasions arise when we would like to specialize or extend not just a single object, but an entire hierarchy at once. No existing object-oriented language provides a mechanism for this that does not require the modification of previously existing code. Yet conceptually, we should be able to perform these changes by some sort of additive extension.

Suppose we would like to construct a hierarchy of geometrical objects, such as squares and triangles, which all respond to methods like `DISPLAY`. These would all be built on a common base object named `SHAPE`, which might contain variables for the common attributes like a `CENTER` point and perhaps a `BOUNDING-BOX`. Objects like `TRIANGLE` and `SQUARE` inherit from `SHAPE`. If we started out with black-and-white shapes, we could certainly add a `COLOR` attribute to `SHAPE` by simple extension. But this would leave us with the task for reproducing the entire hierarchy of sub-objects emanating from `SHAPE` in new, colored versions. In most present systems, programmers would simply be tempted to add the `COLOR` attribute directly to `SHAPE` rather than creating a new `COLORED-SHAPE` object. This would automatically extend all the geometric objects to colored versions, but at the cost of a destructive editing operation. The previous black-and-white version would be lost, and the editing operation introduces the possibility of errors, such as accidentally sending a color command while running a previously existing black-and-white program, unless careful attention was paid to upward compatibility.

Some languages, like the Flavors object-oriented extension to Lisp, have addressed the issue of sharing of orthogonal features by using the approach of "mixins", using inheritance from multiple parents. This approach lets us create new objects possessing previously unanticipated combinations of behavior from previously existing abstractions. In this approach, a `COLOR-MIXIN` could be created independent of any shape properties, and a new type of object declared to inherit from `COLOR-MIXIN` as well as some specific shape property like `TRIANGLE`. However, this approach still involves all the steps of reproducing all elements of the shape hierarchy in their color versions, or modifications to the code to retroactively mix in the color feature. Thus the mixin feature does not provide true support for smoothly implementing an unanticipated changes such as the black-and-white to color transition.

We don't have, at the moment, a solution to this problem; we state it merely to point out the direction in which we believe object-oriented systems must evolve.

Summary

We have described two sharing mechanisms, *templates* and *empathy*, which hide at the core of object-oriented programming languages. Templates allow two objects to share a common form. They may be embedded inside classes, or may be objects themselves. They may also vary in the degree of strictness they impose on system structure. Empathy allows two objects to share common state or behavior. The patterns of empathy may be determined statically or dynamically, per object or per group, implicitly or explicitly. The decomposition of object-oriented languages into template and empathy mechanisms can shed light on their similarities and differences, weaknesses and strengths.

Bibliography

[Agha, 1987] Gul Agha. *Actors.* MIT Press, Cambridge, Massachusetts, 1987.

[Birtwistle *et al.*, 1973] G. Birtwistle, O. Dahl, B. Myhrtag, and K. Nygaard. *Simula Begin.* Auerbach Press, Philadelphia, 1973.

[Bobrow and Stefik, 1981] D. G. Bobrow and M. Stefik. The loops manual. Technical Report KB–VLSI–81–13, Knowledge Systems Area, Xerox Palo Alto Research Center, 1981.

[Bobrow *et al.*, 1986] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging lisp and object-oriented programming. In *ACM SIGPLAN Notices* [SIG, 1986], pages 17–29.

[Hewitt, 1984] Carl Hewitt. Control structures as patterns of passing messages. In Patrick Winston, editor, *Artificial Intelligence: An MIT Perspective.* MIT Press, Cambridge, Massachusetts, 1984.

[LaLonde, 1986] Wilf LaLonde. An exemplar based smalltalk. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 322–330, Portland, Oregon, September 1986.

[Lieberman, 1986] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 214–223, Portland, Oregon, September 1986.

[Lieberman, 1987] Henry Lieberman. Concurent object-oriented programming in Act 1. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*. MIT Press, Cambridge, Massachusetts, 1987.

[Mercado, 1988] Antonio Mercado Jr. Hybrid: Implementing classes with prototypes. Master's Thesis Technical Report No. CS-88-12, Brown University Department of Computer Science, Providence, Rhode Island, 02912, July 1988.

[Moon, 1986] D. A. Moon. Object-oriented programming with flavors. In *ACM SIGPLAN Notices* [SIG, 1986], pages 1–8.

[Power and Weiss, 1988] Leigh Power and Zvi Weiss, editors. *Addendum to theProceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 23 of *Special edition of SIGPLAN Notices*, May 1988.

[SIG, 1986] *ACM SIGPLAN Notices Special Edition on Object-Oriented Programming Languages*, volume 21, November 1986.

[Stein, 1987] Lynn Andrea Stein. Delegation is inheritance. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 138–146, Orlando, Florida, October 1987.

[Stroustrup, 1986] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[Ungar and Smith, 1987] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242, Orlando, Florida, October 1987.