A Simple and Unifying Approach to Subjective Objects

Randall B. Smith
David Ungar

Sun Microsystems Laboratories, Inc.
2550 Garcia Ave., MTV 29-116
Mountain View, CA 94043

(415) 336-2620
Submission for TAPOS special issue on Subjectivity in Object-Oriented Systems.
{randall.smith,david.ungar}@sun.com

Abstract

Most object-oriented languages are objective: an object always responds to the same message in the same way. Subjective objects more closely match naturally occurring systems, and they provide consistent solutions to a wide range of problems, problems that otherwise must be solved by varied and specialized mechanisms. Applying a *perspective-receiver symmetry* principle in designing the subjectivity semantics of an object-oriented language results in a semantically uncluttered language with a surprisingly wide range of utility. We employ this approach in creating the language Us, a subjective version of Self.

1 Introduction: Subjectivity meets Objects

Object-orientation [MMN] wins converts because it can more closely simulate the real world than procedure-oriented programming, yet most object-oriented languages are stuck in a 19-th century, objective stance. A Self or Smalltalk object, for example, always exhibits the same behavior, no matter what the context. But whenever multiple minds come together in computer systems, whether they be minds of users, creators, or the simulated minds of objects, the system must somehow exist in multiple overlapping but not identical realities. Each user has a legitimate need to see the system in his or her own way; each programmer may need to work in a slightly different version (until changes can be merged), and each abstraction sees the outside of other abstractions, but the inside of itself. An objective stance is not sufficient to the task of simulating a world containing more than one entity, be they people or objects, or even a single entity at more than one time.

Subjectivity is an inescapable aspect of two great natural systems: human language and the physical world. For example, an object in the physical world does not have a unique appearance: what it looks like depends on the direction from which it is viewed. In this century we have learned that all observations of physical parameters are relative to the frame of reference of the observer, and the results reflect the state of observer and observed. The second naturally occurring relativistic system is natural language, in which the meaning of a word is also subjective: factors such as past experience and context can lead to radically different meanings being associated with the same term. Not only do these two systems feature subjectivity, but both natural language and the physical world allow issues of perspective to be changed from within the system: it is not necessary to leave the physical world to change frame of reference, and it is possible to discuss meaning and points of view from within language. Subjectivity is an important aspect of real-world systems — we will show how it is present in a wide-ranging set of computational problems, and how subjectivity, when made a fundamental and dynamic property of a language, can solve these problems.

We have worked out a particular approach to subjectivity and applied it to Self [US]. Self is an unusually pure and simple object-oriented language, and so is a natural choice in our search for a minimalist approach. The resulting language, called “Us,” has been implemented within Self. Currently, Us is a proof-of-concept implementation only. Us has no user community and has had no large programs written in it. As with any language design, Us should be taken as a largely untested proposal, a first stab at making subjectivity a first class and flexible aspect of a pure object-oriented language.

1.1. Subjectivity is everywhere

As a result of the mismatch between the objectivity of our programming languages and the subjectivity of our problem domains, a new ad hoc mechanism must be created every time we bump up against a problem that includes multiple perspectives. Most language environments use one scheme to deal with multiple users, a completely different one to deal with multiple versions of source code, and yet a third to deal with different access rights to state within an object, even though all three problems can be recast in terms of observer dependence. This duplication of work and complexity suggests that it would be profitable to support subjectivity as a fundamental principle of object-oriented systems by embedding it in the language.

The following two examples give a brief taste of how Us can address both encapsulation and group programming with subjectivity. The examples are intentionally introductory, so questions about details will naturally arise: more complete explanations are postponed to later sections.

Example 1: Subjectivity for Encapsulation. To provide access rights within a group of mutually trusting objects, capability-based architectures provide mechanisms that restrict certain operations, such as storing bits on a disk sector, to be performed only within a trusted kernel. A similar effect can be achieved by using perspectives. For example, a perspective can be built that contains extra capabilities on file objects. When objects in the trusted kernel receive a message, they can change to a privately held perspective to send further messages.

```
Class: BankAccount {
  public:{
    transferTo(accountOrPerson, amount);
  }
  private:{
    addAndRecord(amt);
  }
}
```

Figure 1. Perspectives in the physical world can help provide encapsulated access among a group of cooperating individuals. Objective languages must provide interface encapsulation through extra mechanism, such as special private and public declarations, usually created for that purpose alone.

As a more complete example, suppose we wish to simulate an economic system, filled with objects like businesses, people, and bank accounts. Suppose we started to build the bank account object. We might start by thinking of its public interface, and decide to create a `transfer:To:` method that takes two arguments: an amount to transfer, and a person or account object into which to transfer the funds. Our public `transfer:To:` message maintains the "constant total funds" invariant — it will move money from one place to another, without creating an overall gain or loss. So the public interface maintains an important system-wide invariant, thereby making the simulation easier to use correctly.

We turn to the task of implementing this method. We decide to use another message we will give our account objects: addAndRecord:, which adds its single argument to its balance, and records the transaction in some log. If we were to use Self, whose syntax is similar to that of Smalltalk, we could write

```
transfer: amt To: acctOrPerson = (
   addAndRecord: amt negated.        "Subtract amt from our balance..."
   acctOrPerson addAndRecord: amt.   "...and add it to the other account"
   "(When no receiver is specified, the message is sent to self.)"
).
```

Most languages have a way to indicate that certain messages are not part of the public interface, and because the `addAndRecord:` message does not maintain the "constant total funds" invariant, we would like to make it part of a private interface. In Self or Smalltalk, the programming environment can classify methods, but these classifications have no base-level semantics. In Us, we maintain a *perspective* in which such private interfaces are accessible.

A brief word about objects in Self and Us: an object is simply a collection of slots. Each slot contains a name, and a reference to a method or to any other object. When a message is sent to an object, and the message name matches the slot name, that slot is activated. If the slot references a method, the activated slot acts like an element in a method dictionary—the code in the method is run. If the slot references a regular, non-method object, the slot just returns that regular object as the result of activation. We have given our account objects two slots so far: `transfer:To:` and `addAndRecord:`, but we will be adding more, such as a `balance` slot (to keep track of the account's value) and a `log` slot (to record transactions), for example.

Us supports encapsulation by allowing a slot to be visible only from certain perspectives. By moving the `addAndRecord:` slot to a special perspective, the public interface is reduced. The `transfer:To:` method written in Us might become

```
transfer: amt To: acctOrPerson = (
    (  addAndRecord: amt negated.
       acctOrPerson addAndRecord: amt.
    )⊗ pvtPersp
)
```

where `pvtPersp` is itself a message (to self).

The notation "`exp1 ⊗ exp2`" means essentially "evaluate the message sends in `exp1` from the perspective found by evaluating `exp2`." If we were to leave off the change in perspective, the message `addAndRecord:` would be sent from the default perspective, and so the messages would not be understood.

But if `pvtPersp` is a message send to self, where is the corresponding slot named `pvtPersp`? There are several possible locations. Each relevant object (such as each account and each person object) in the simulation could have such a slot, with each of these slots referencing (or even computing) the perspective object. In a simulation with hundreds or thousands of these objects, this would mean quite a bit of replication. Replication can be eliminated by using inheritance, and both Self and Us feature a kind of delegation-based inheritance. Any slot can be designated a "parent" — message lookup will continue looking through the objects in these parent slots, until a matching slot is found. All accounts, for example, would typically have a parent slot referencing a

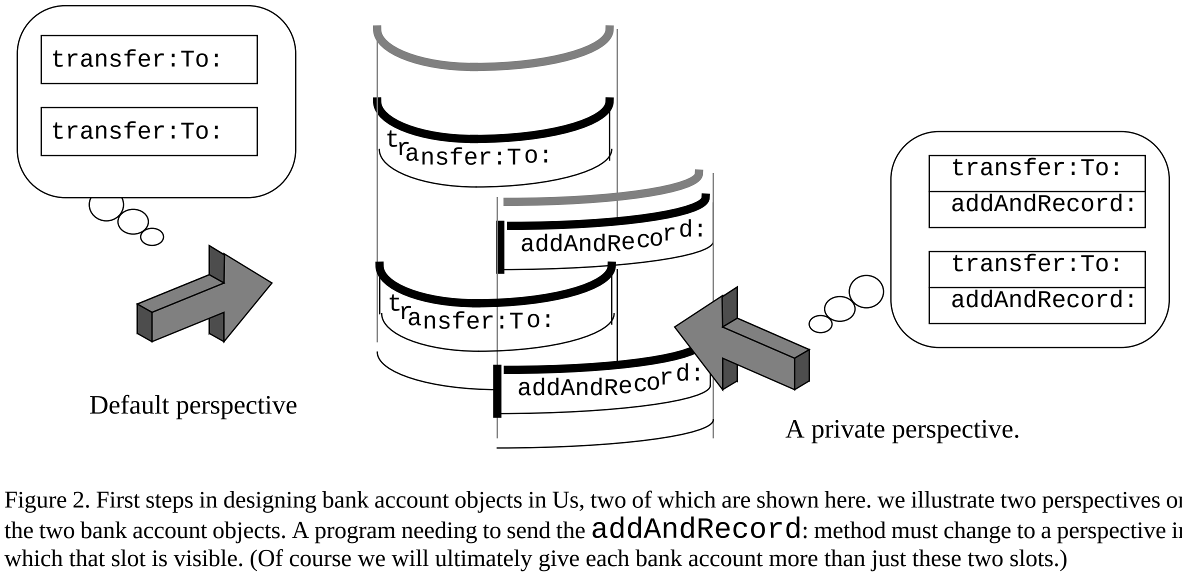


Figure 2. First steps in designing bank account objects in Us, two of which are shown here. we illustrate two perspectives on the two bank account objects. A program needing to send the `addAndRecord`: method must change to a perspective in which that slot is visible. (Of course we will ultimately give each bank account more than just these two slots.)

single object, to act a repository for shared behavior. This common parent would be a natural place to put the `pvtPersp` slot. Other common state or behavior could be put in this parent as well.

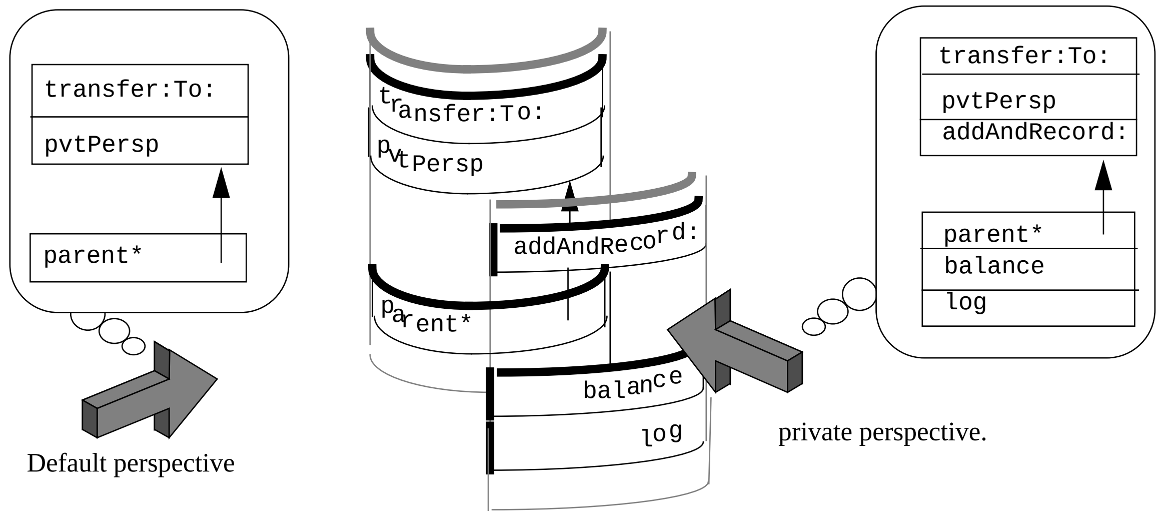


Figure 3. Two perspectives on a bank account object and its inheritance parent. The inheritance parent (referenced through a slot marked with *) holds widely shared behavior, including a reference to the private perspective.

However, the slot `pvtPersp` is publicly available for inspection and possible abuse. Protecting an object from intentional attack is arguably a different problem than the software engineering concerns that arise in building and maintaining a simulation, but it can of course be an issue. We will revisit this example in greater detail, and will discuss the security issue in section 4.2. For now, we hope it is clear how the general notion of perspective can narrow the public interface, and help maintain invariants. Private state can be accessed by changing perspectives in running code.

Example 2: Subjectivity for Group Programming. In the physical world, problems can arise when two or more workers make changes to the same physical system. Two mechanics working on a car have to negotiate how to divide up the task, and how to physically arrange themselves and the system so they can work to minimize interference. The mechanics suffer extra complication due to the collaborative character of their problem, but at least they can adjust their points of view on objects while they work to address interference problems.

In the software world, managing and coordinating the changes of individuals in a programming team is a difficult problem in software engineering. Two Us programmers working on the same program have to negotiate how to divide up the task, and how to associate their work with different perspectives on the system so they minimize interference. The programmers, like the mechanics, suffer extra complication due to the collaborative character of their problem, but at least the perspective mechanism of Us provides a way to reify each individual’s changes as first class objects so they can be manipulated, inspected, and be combined with others in the group. Merging changes into a new perspective is a more straightforward task when perspectives are already objects in the same system.

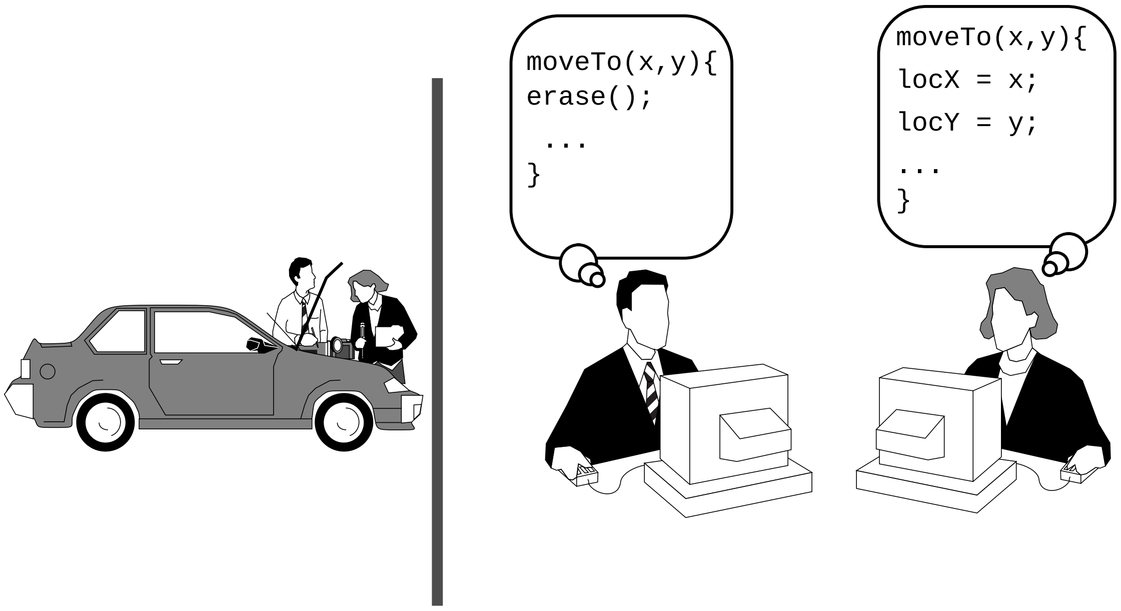


Figure 4. The physical world allows individuals to adopt common or separate perspectives in order facilitate working as a group or as individuals. Computational systems usually do not reify points of view, so merging and unmerging of viewpoints is relatively awkward.

In the language Us, suppose there are two programmers, Tom and Mary, who have made some small changes to the economic simulation. A third user, Fred, uses this simulation quite often: it is part of his standard system. Fred wants to investigate the changes made by Tom and Mary.

Tom and Mary have each placed their changes in a *layer* object. A layer is how we implement perspectives in Us. Specifically, a perspective is an ordered sequence of layers. Each layer has a reference to its “layer parent” through a `layerParent` slot — this slot can be set at runtime, so mixing of perspectives can be a routine event in Us.

Fred first creates two slots in his system, `tomsPerspective` and `marysPerspective` to reference the two sets of changes. Tom and Mary have placed their layer objects (containing just their changes) into files in their home directories. Fred executes the expressions

```
tomsPerspective:   ‘~tom/simulationChangesLayer.us’ readFile.
marysPerspective:  ‘~mary/simulationChangesLayer.us’ readFile.
```

to set these slots. The slot `tomsPerspective` now contains a single layer object, with only those additions, removals, and changes made by Tom. (The layer parent of this layer is currently nil, so in a moment, Fred will change the layerParent slot to create a more useful perspective.)

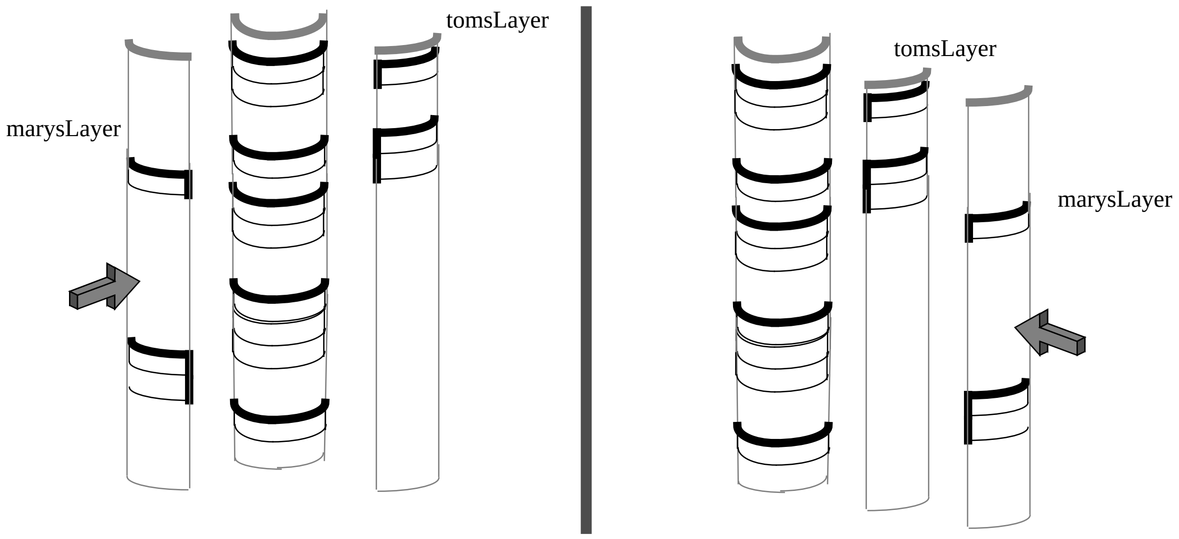


Figure 5. Three layer objects. Layers implement perspectives in Us. Layers can be combined in different orders to create different perspectives. Each layer has a `layerParent` slot that designates the next layer in the chain. The configuration of layers in these diagrams suggests the layer parent relationships. At the left, we show how `marysLayer` and `tomsLayer` might share a common layer parent. In that case, the slots in `tomsLayer` are not visible when `marysLayer` is the current perspective. At the right we show how adopting Mary's layer as a perspective would appear after evaluating `marysLayer layerParent: tomsLayer.`

Now in Us, as in Self, the routine way to make new objects is to copy a prototype. To run the unchanged simulation, Fred evaluates

```
simulation copy run.
```

This creates a window and starts some animated simulation going. The above expression is equivalent to

```
(simulation copy run) ⊗ here
```

where "`here`" is the globally inherited slot containing the current default layer. The notation "`⊗ here`" means "from the current perspective." To try Tom's version, Fred now sets the layer parent slot so that this layer will inherit all the changes contained in the `here` perspective:

```
tomsPerspective layerParent: here
```

then executes the expression

```
(simulation copy run)⊗ tomsPerspective.
```

Because the `layerParent` slot in Tom's layer has been set to "`here`," the simulation is being run from a perspective in which Tom's changes have been combined with Fred's default perspective on the system.

Similarly, Fred evaluates

```
marysPerspective layerParent: here
```

to mix the system perspective into Mary's layer, then evaluates

```
(simulation copy run)⊗ marysPerspective.
```

to try Mary's version.

Fred now has three simulations running on his screen at once (the original version, plus Tom and Mary's versions) and he can compare them side by side.

Fred might now quit all three of his running simulations and turn to the task of merging the changes. He creates two new slots: `tomThenMary`, and `maryThenTom`, to represent two approaches to merging the changes:

```
tomThenMary: (tomsPerspective copy layerParent: marysPerspective copy).
(simulation copy run)⊗ tomThenMary.
maryThenTom: (marysPerspective copy layerParent: tomsPerspective copy).
(simulation copy run)⊗ maryThenTom.
```

The two running simulations enable side by side comparison to investigate the commutativity of the two changes. More directly, Fred can simply look at the two layers:

```
marysPerspective    print.
tomsPerspective     print.
```

or he could employ facilities to compare the two layers, such as tools that flag changes to the same slots, or tools that guide the user through a merging process.

The general problem of merging of changes for Self-like objects is discussed in [U]. Generally speaking, there are interactions between the receiving system and the individual changes being made that the simple layer approach may not capture by itself. Perhaps the most difficult problem is how to find the "new host object" for a slot that was created in a different system. For example, a new slot S was added to object O on a layer L in system W. When layer L is brought into a new version of the system, W', how can object O be found? W' may not contain anything remotely like O. Thus, in addition to simply creating a layer of changes, the slots in the layer must come with extra information. Despite such complications, we believe that the power of having layers available as objects provides a good foundation for attacking the programming team problem. In the above example, the object-ness of the layers was the key that enabled Fred to inspect, try out, and merge the changes made by the various programmers.

Coordinating changes among a group of programmers involves time-scales of days or even months, can involve hundreds of large objects, and is normally visible to the user, whereas providing a protected layer of capabilities may involve only a few objects changing perspectives on the millisecond time scale, and normally happens deep in the system, far away from the user. Despite the orders-of-magnitude differences, each of these examples suggest the same weakness in the object-oriented model: a lack of context.

A skeptical reader may think that, since subjectivity is a complication, it would be better to stay with purely objective systems. After all, when we design a computer language we have an opportunity to create our own reality—why not make this reality consistent with our more naive but intuitive physics? Well, additional complexity is a genuine concern of ours, but subjectivity may not be an overwhelming complication for humans; after all people exhibit greater facility with manipulations in the naturally occurring but inherently subjective domains of language and the physical world than they do with manipulations of purely objective computer languages. Furthermore, pure objectivity may indeed be adequate for very simple software, but we believe there are a number of problem types for which subjectivity is a natural solution. They are each addressed in current languages by ad hoc mechanisms that provide relatively awkward solutions.

1.2. Us: toward Uncomplicated Subjectivity

Us can be thought of as a subjective version of the language Self. Us has a model of computation based on a “subjective object.” In this model, there is no single “true” state and behavior for an object: rather, the state and behavior of an object depends on a “perspective” which we reify as an object. Our work builds upon related approaches of object-oriented systems like PIE [GD], CLORIS [HO2], and RPDE[3] [HO1]. Our primary reservation about subjectivity is its potential to complicate, confound, and confuse. Consequently, we believe it is crucial to take a conceptually minimal approach to subjectivity. We also believe that a conceptually minimal approach can lead to a more flexible system. Our goal is to attain a kind of symmetry between the role of the message *receiver* and the role of the message sender’s *perspective*. In fact, we do not achieve perfect perspective-receiver symmetry, but we will explain where and why we depart from strict observance of this principle. Nevertheless, it is adherence to a minimalist esthetic that most distinguishes Us from prior formulations of subjectivity in computation.

The next section briefly reviews relevant prior work. Section 3 provides a fuller elaboration of our simple object model that provides for subjectivity. In section 4 we offer more evidence that subjective objects can be useful in many situations, and show how our specific model would apply.

Objects have come to be accepted as a metaphor that can provide genuine utility beyond the procedural paradigm. We believe that subjectivity should also be considered a part of the story, and that providing for it at the language level can save repeated awkwardnesses at all levels of the system.

2 Relation to previous work

The idea of extending object-oriented systems to include notions like contexts, views, perspectives and the like is a thread running back through the past decade (with non-object-oriented precursors in the late 1970’s). Perhaps most notable in recent years has been the work centering around RPDE[3] and its follow-on work. Ossher and Harrison show how class hierarchies can be combined [OH] and even restructured [HO2] by taking different perspectives (or different “subjects”) on the system as a whole, and describe how access can be granted or denied to instance variables and methods alike, keeping their discussion at a fairly language independent level. These authors point out the issues of new object initialization and the richness of perspective combination strategies. We will be comparing our approach on these subjects in section 3.

By extending message dispatch to include some sort of perspective as well as receiver, our language Us is doing a kind of “double dispatch.” Languages with multiple dispatch [BKKMSZ], [BDGKKM] can similarly be thought of as including a kind of subjective element. Our approach can be seen as choosing a particular semantics for double dispatch, and effectively applying it at every message send with an implicit argument. We also have a separation between inheritance attributes and perspective layering attributes, as discussed in section 3, that will not fit easily into the multiple dispatch paradigm. However, languages with multiple dispatch are almost automatically going to exhibit the kind of perspective-receiver symmetry we are after.

The database community speaks of views or schema that allow the various users of a database to see the same records differently, thereby protecting privacy or supporting the viewers' differing needs. Views and schema have been discussed in terms like "semantic relativism." (for a survey see [HK], and application to OODB in [TYI], [HZ], [WKOST]) Such work overlaps our own in that data structures look different in different contexts. Concepts such as "context" and "world view" appear even earlier in applications of artificial intelligence and in the language KRL [BW], but one of the more central earlier works in the object-oriented language area is PIE, [GD], in which layers can be placed over Smalltalk objects in order to allow system changes to be independently "layered" and installed. PIE was aimed at the programming change problem, and was not really intended to be used for switching perspectives on a per-message send level. Multiple world views played a part in Amber, another language extension of Smalltalk designed at Xerox PARC [BHMPZ]*, and the notion of an object being a "figment of its viewer's beliefs" is a feature in Alan Kay's sketch of a language he called Rainbow, [K].

Our approach to subjectivity is broadly similar in spirit to much of this previous work, but differs in its emphasis on a minimalist approach to pushing subjectivity into the act of message sending. We hope that the issues surrounding subjectivity can be thrown into sharp relief by relatively simple semantics of Self, our starting point. More fundamentally, we believe that subjectivity can be simplified by establishing a symmetry between the receiver of a message and the perspective in which a message is sent. We wish to bring the thread of research into this area closer to the fabric of practice by making subjectivity more comprehensible.

3 Conceptual sketch

Before sketching our formulation of subjectivity we take a moment to introduce our terms: The "attributes" of an object are those elements that can influence its state and/or behavior. Thus in a language like Smalltalk, the attributes include methods and the various kinds of variables (instance, class, global, and pool). For Self an attribute is simply a slot, which has a name and references some object, possibly a method. For the sake of simplicity our formulation is a "pure" object-oriented system in which everything happens by sending messages to objects, although the general approach should apply to just the object-centered part of any system.

Subjective programming is a bit like the third dimension. We can make an analogy that starts with one dimension (procedural languages), progresses to two (object-oriented languages), and extrapolates to three (Subjective languages). Just as a square is an extension of a line segment, we will start by explaining how object oriented programming with inheritance can be considered a kind of subjective version of procedural programming. A line becoming a square is in some sense similar to how a square becomes a cube: when we take the same extension technique that transforms procedural programming to objects, and apply it to objects, we get a kind of subjective programming. We will start at quite a general level and become more specific, finally describing Us, our subjective version of the language Self.

Procedure invocation in a simple procedural language (Fortran, say) can be thought of as a kind of message passing in a degenerate case in which there is only one message receiver. The message name is looked up in a kind of giant virtual dictionary, and a resulting "method" (procedure) is invoked. For example the Fortan expression `sqrt(2.0)` sends

* Ambers worlds had quite different semantics than Us layers, and its approach was somewhat idiosyncratic to Smalltalk.

the sqrt message to the (implicit) world, with argument 2.0. Since there is only one receiver (the entire world), the `sqrt` message always runs the same code (the square root routine).

Generalizing from procedure- to object-oriented programming imposes another indirection in method lookup: in O-O programs, there exists more than one potential receiver (just as in 2-space there is more than one possible y coordinate), so one must send a message *to some object* by specifying the receiver. Each object needs its own virtual dictionary of message-to-method mappings. For example the message `displayOn: aBitmap` will run different code depending on whether its receiver is a square or a circle. Just as two coordinates are needed to specify a point in 2-space, so are two objects: the name of the message *and* the receiver are needed in order to specify some code to run.

Object-oriented programming has some degree of subjectivity in it already, because the interpretation of the message

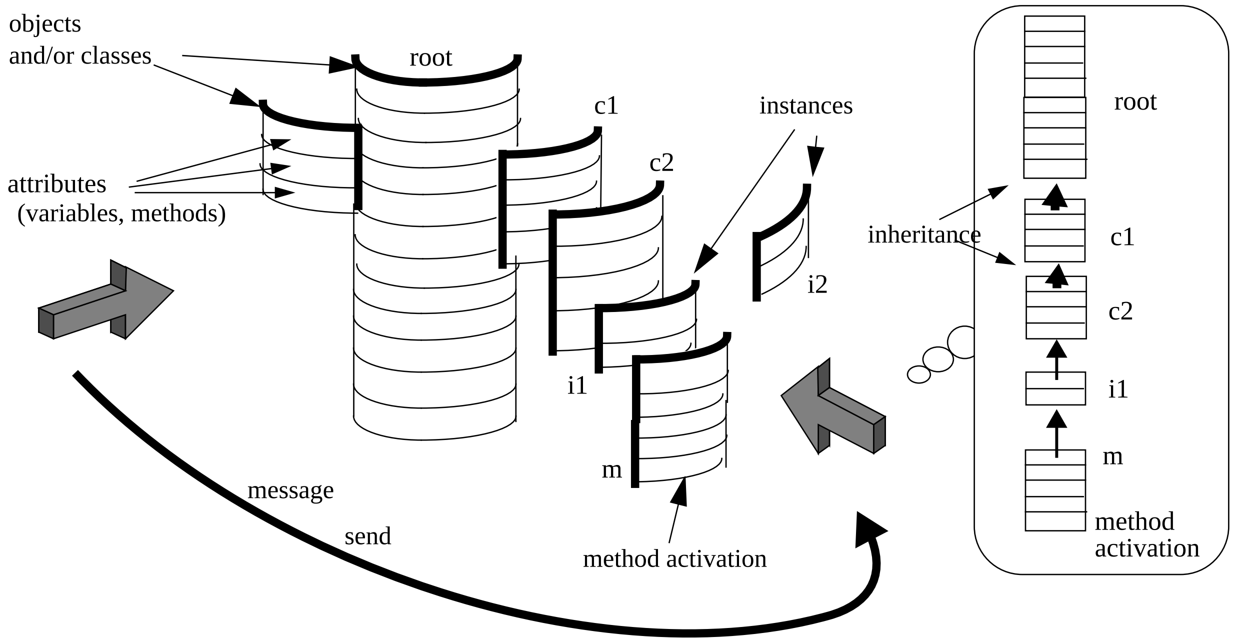


Figure 6. Object-oriented programming with inheritance can be considered points of view through layers on an underlying single "root." The root is a generic set of behaviors and globaly available state. The attributes in the center-most object (or class) are those shared most widely. Refinement and extensions to this core behavior are layered on top as one moves outward through inheritance children towards a specific object. Finally, a specific method activation makes state available through temporary variables or specific argument variable bindings.

is carried out in the receiver's frame of reference. In languages like Self (or C++), in which the reserved word *self* (or *this*) can be syntactically elided when self is the message receiver, this sense of subjectivity is even stronger. The message receiver can "go without saying" in such languages, providing a kind of background context upon which computation is implicitly played out (see Figure 6 on page 11). Thus if you stand on your head, you can think of object-oriented programming as message sending to only one object, which can be viewed from many different perspectives.

However, in object-oriented programs, all objects that send message X to object Y will get the same result. Of course, a world populated by objects is richer than the simple procedural system, but it is still an "objective" world in that a reference to an object gives access to a particular set of behaviors, regardless of the reference holder. To attain full subjectivity, we propose that yet another level of indirection be applied to message lookup: somehow, a perspective or “point of view” that can be specified by the message sender should be a participant in the lookup algorithm. Before a message can be sent to an object, the system must consider the perspective from which the message is sent, just as in a 3-D world, a third coordinate must be taken into account when locating points.

For the sake of uniformity, this perspective should itself be an object. If object X has a reference to object R and perspective object P, we say X can send any message to R from the P point of view. A syntactic extension to an objective OOP language will be needed to allow the specification of this extra perspective argument.

In subjective programming then, an object reference is somewhat more vague than in objective programming. In objective programming, an object reference is conceptually unambiguous, whereas in subjective programming an object reference must be combined with a perspective in order to become concrete enough for message lookup. Subjectivity arises when the message lookup depends on the message, the receiver, *and* the sender’s perspective, so that the same object has different manifestations from different perspectives.

It is of course possible to imagine further generalization in the same direction. Perhaps a fourth element could also have a voice, forming a four-way collaboration in message lookup. Such mind-boggling generalizations are entertaining, but we limit our discussion to the sender’s perspective, receiver, and message, as we believe many problems have natural solutions through this particular mechanism.

3.1. Perspective as layer

Central to our formulation of perspective the concept of a *layer.* A layer can be thought of as metaphorical sheet of glass. Each layer may have another layer as a *layer parent*, and the layers therefore form a hierarchy. A layer considered together with its layer parents is a perspective. In Us, a reference to a layer serves as a reference to a perspective. So we use the terms “layer” and “perspective” somewhat interchangeably. For example, one might say “Changing the layer parent of layer L changes the slots visible from the L perspective.”

The chain of layer parents is intentionally analogous to the chain of inheritance parents, according to the perspective-receiver symmetry principle. One must keep in mind that a layer comes with its whole chain of layer parents and the entire chain determines what is visible from a particular layer, just as a Self object comes with its whole chain of parents and the entire chain determines what is “contained” in the object.

Each object has exactly one *piece* on each layer.

$$\text{piece} = P(\text{object}_n, \text{layer}_m)$$

A piece is a (possibly empty) collection of attributes (variables and/or methods) and is properly considered that part of an object which is associated with a particular layer (see Figure 7 on page 13). A piece can be thought of as a metaphorical bit of paper stuck to a layer. By bending our sheets of glass (layers) into cylindrical sections, we can visualize all the pieces for a single object filling a horizontal disk-shaped region (See Figure 8 on page 14.) An object reference can then be thought of as a reference to this disk, the collection of all pieces for the object.

Our metaphor has an observer in a particular perspective looking horizontally through several curved sheets of glass. The pieces for a particular object will then line up to form the illusion of a discrete packet of attributes, containing state and behavior (See Figure 7 on page 13). The resulting packet is an object as viewed from a particular perspective.

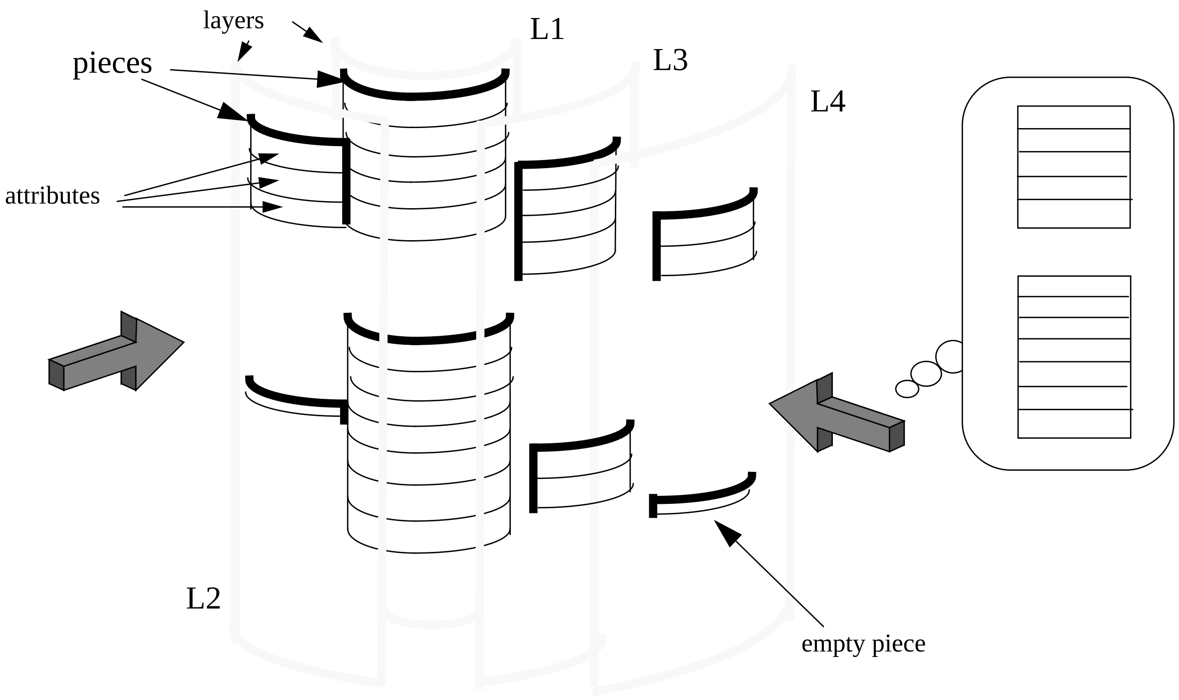
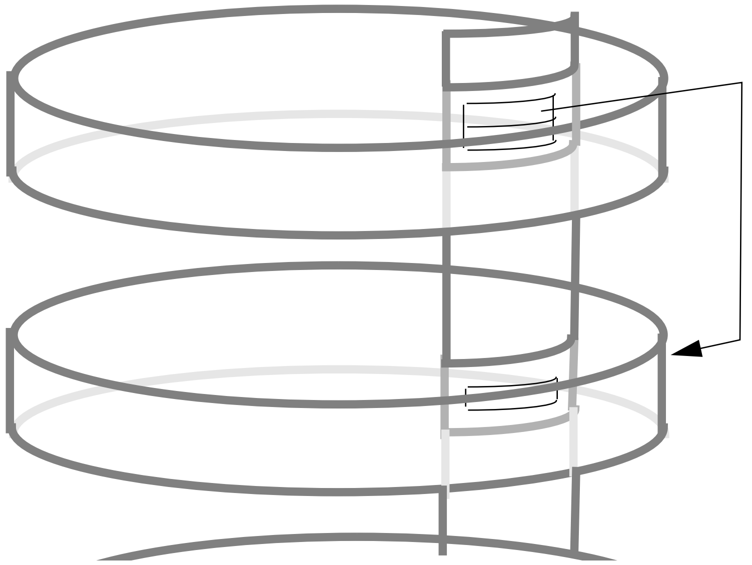


Figure 7. Layers and pieces. Here we have two objects: each object has four pieces because there are four layers. A perspective is a point of view on the entire system, and is represented as a layer. Layers are arranged into an inheritance hierarchy orthogonal to the normal object inheritance hierarchy. Each layer has exactly one piece per object. For the language Self, attributes are slots which can act as variables or method-dictionary entries, because they can contain either methods or non-method objects. The similarity between this figure and Figure 6 on page 11 is the basis for the perspective-receiver symmetry principle discussed in the text.

For the sake of simplicity, we have chosen to formulate a model of subjectivity in which object identity is invariant under change of perspective. In other words, if an object exists in any perspective, it will exist in every perspective, if two references refer to the same object in any perspective, they will do so in every perspective, and if two references refer to different objects in one perspective, they must refer to different objects in every perspective. In order to maintain this invariant, every object must have a (possibly empty) piece on every layer. Outer layers tend to be sparse — most of their pieces will in fact be empty. Typically, if an outer layer were to be removed from its parent layer chain and used alone as a perspective, most messages sent to most objects would result in some sort of “message not understood” error.

Figure 8. An object reference. An object can be pictured as a cylindrical disk in a system in which a layer is a vertical cylindrical section slicing through all disks. At the intersection of each layer and each object disk is a “piece.” Pieces contain zero or more named attributes which reference objects in the system, possibly methods.



The attributes within a piece in one layer can mask (or override) attributes in its layer parent. By structuring the layers in a hierarchy with this kind of overriding semantics, the resulting system exhibits a kind of perspective-receiver symmetry, which we describe below. We believe that minimalism is important in language design, and employing familiar semantics for perspective inheritance enables the programmers to use their objective object-oriented language experience in understanding and designing subjective software.

3.2. The perspective-receiver symmetry principle

The perspective-receiver principle is illustrated by the similarity between Figure 6 on page 11 and Figure 7 on page 13 (and by the metaphor of higher-dimensional Euclidean spaces). If there is only one perspective available in the entire system, clearly the system reduces to objective object-oriented programming. What if there are many perspectives objects, but only a single receiver for all messages sent? Receiver-perspective symmetry states that if in the entire system, only one object is used as a receiver while many perspectives are employed, then we will have the same semantics as if there were only one perspective and many receivers: a conventional objective object-oriented language with inheritance.

Applying the perspective-receiver symmetry to a language with inheritance will mean in particular that perspectives can be arranged in an inheritance hierarchy, that the semantics of overriding is the same between parent and child in both the perspective and inheritance direction, and that any given perspective can have many children, as represented in Figure 7 on page 13. We have achieved this much with our use of layers and their layer parent attribute.

Message lookup and inter-perspective invariants: Messages are sent from a single perspective. In our approach, the response of an object to a message sent from layer L is completely independent of pieces outside of L’s layer parent chain. Hopefully this will seem intuitively reasonable based on the kind of geometrical metaphor we describe. However, it is not the only possibility, and in particular is different than the “merge” semantics described in [HO2], in which a message is sent from all perspectives at once (as though they were merged together), but evaluated locally within each perspective. Our approach is in keeping with receiver-perspective symmetry: a message lookup proceeds along a narrow and well-defined path through the layer and its parent chain, similar to the way a lookup path in an objective system wends its way through the inheritance links.

However the price of this symmetry is that invariants that involve attributes in multiple layers cannot be maintained automatically. Consider a point object that stores x and y in one perspective and rho and theta in another. Assume that we want to maintain the invariant that the x and y values for this point will be the same in either perspective. In Us, the method that assigns to x must also explicitly change perspectives and set rho and theta, just as if the polar coordinates were in a different object. With “merge” semantics the update message would automatically be sent in the polar perspective as well as the cartesian one. However, as Harrison and Ossher point out, merge semantics are not intended to solve all such invariant maintenance problems: for example, the rho, theta perspective might be using the point in such a way that the constraint y=2x is maintained. The x,y perspective may not be trying to maintain any such constraint, or worse, it may be trying to maintain a conflicting one. So we prefer that any cross-layer constraint that needs to be maintained be made explicit in the code. The simple, symmetric semantics of Us seem to provide a sufficiently flexible framework for helping to resolve such problems (to the extend they can be dealt with!), but this is an area that could merit further study.

The receiver-perspective symmetry is at least weakened by the lookup algorithm which composes the layer hierarchy first, and then constructs the normal inheritance hierarchy. This ordering may seem intuitively reasonable, and is essentially forced upon us if we are to allow normal inheritance links to be first class attributes which can, in the layer direction, be inherited or overridden by layer children. Indeed in general, a piece may have no inheritance link attribute at all. If such a piece were encountered in a lookup scheme that prioritized the inheritance links over the layer parent chain, the lookup would stop immediately. Still, this symmetry violation is troubling, and may be a manifestation of a deeper problem, that of reusing base-level attributes to carry essentially reflective information about inheritance.

The perspective-receiver symmetry results in another difference from Harrison and Ossher merge semantics. Visibility of attributes of an object’s pieces within a layer parent chain follows the usual semantics of overriding. Hence we allow the formation of a layer chain (L1, L2) to give different results than (L2, L1). Switching the order of classes (i.e. C1 inheriting from C2 vs. C2 inheriting from C1), in an inheritance hierarchy can make a radical difference, so our symmetry principle allows layer ordering to matter. This may seem intuitively reasonable, but it differs from the merge semantics mentioned above, which only allows merges if order switching is irrelevant.

Layer parent links vs. inheritance links: The distinction between the layer parent and a regular inheritance link (e.g. a *parent* slot in a Self object or the superclass variable in a Smalltalk class) can be confusing. Why have two separate but orthogonal hierarchies? It seems like a violation of Alan Kay’s language design principle to have two things that are similar be made distinct. However, one of our goals is to achieve a simple uniformity by reifying layers as objects, so that it is possible for a layer to be used as a message *receiver* in certain situations. As we have seen in our implementation of Us, it is quite common to send messages to layers (to recombine them into specific layer chains, for example).

Reusing the inheritance link as the layer parent link would create problems. There is a difference between the attributes in the pieces a layer holds on behalf of other objects (all of which might be empty for a newly created layer, say), and the attributes (like `layerParent`) that reflect the state of the layer itself. Another more graphical way to think of this: a layer, because it is an object, is represented not only as a vertical cylindrical segment through the disks in Figure 8 on page 14, but is itself also a disk somewhere in that figure. The pieces contained in a layer's slice and those contained in its disk need not be the same for the system to treat layers and disks in a similar way.

Any object as layer? If every layer is an object, should not every object be a layer? Our implementation of Us does not go that far. It would not be really difficult to impose this kind of strong symmetry property on the system, but even if we built things this way, there would be some objects that would tend to be used in a layer-like role, and others that would be used in a receiver-only role, and so the purpose of the grand symmetry might well remain rather theoretical. However, we suspect there are some elegant designs and deep issues in this direction, and unearthing them might yield some surprising flexibility or utility.

Creating new objects: When obtaining new objects either by instantiation or by copying, another issue arises: should *all* the pieces for the new object be created, even those in layers that are not in the sending layer chain? One might take the position that the new object will have pieces only in those layers that are part of the sending perspective chain. However, we have already mentioned the need for each object to have a piece on each layer, though in practice most of these may well be empty. We have chosen to have the default way to copy produce shallow copies of every piece. This makes it sensible to pass out references to objects that take a different perspective on the system, but does not cure all problems, as the initialization of all the layers may need to be addressed as well. Initialization is an important but special case of maintaining invariants across layers, which we have previously discussed. Using this copying policy and applying the perspective-receiver symmetry principle means that a copy of a layer object should come with a fresh copy of each individual piece.

When does the perspective change? The system needs a default policy for answering this question. There are perhaps many possibilities, and we will mention two, starting with the "rubber band policy." Under the rubber band policy, if a method activation in perspective L1 sends a message through perspective L2, the system swings over to perspective L2 to perform the look up, but upon finding the method to activate, returns to perspective L1 to install and execute the activation. The rubber band has pulled the activation back to the launching perspective. The opposite policy (the one we employ) might be called the "minimal motion policy." Under this scheme, when the message is sent from an activation in perspective L1 through perspective L2, the system swings to the new point of view to perform the lookup, but stays there as that method is activated. Further messages will cause the more activations in the same perspective L2 (unless there are further explicit perspective changes), and the system will finally return to L1 only when the activated context in L2 returns. We find the minimum motion policy useful for most applications of subjectivity, some of which we discuss in the next section, and within this policy the effect of the rubber band policy can be simulated, though we have not found an elegant solution. The minimal motion policy is also in keeping with the perspective-receiver symmetry as applied to the language Us: in Us, as in Self, if a message is sent to object O1, and all subsequent messages are sent without an explicit receiver being specified, all the consequent method activations will be in the context of O1. Under the minimal motion policy, if a message is sent through perspective L2, and subsequently no perspective is explicitly invoked, the messages will be looked up and activated in perspective L2.

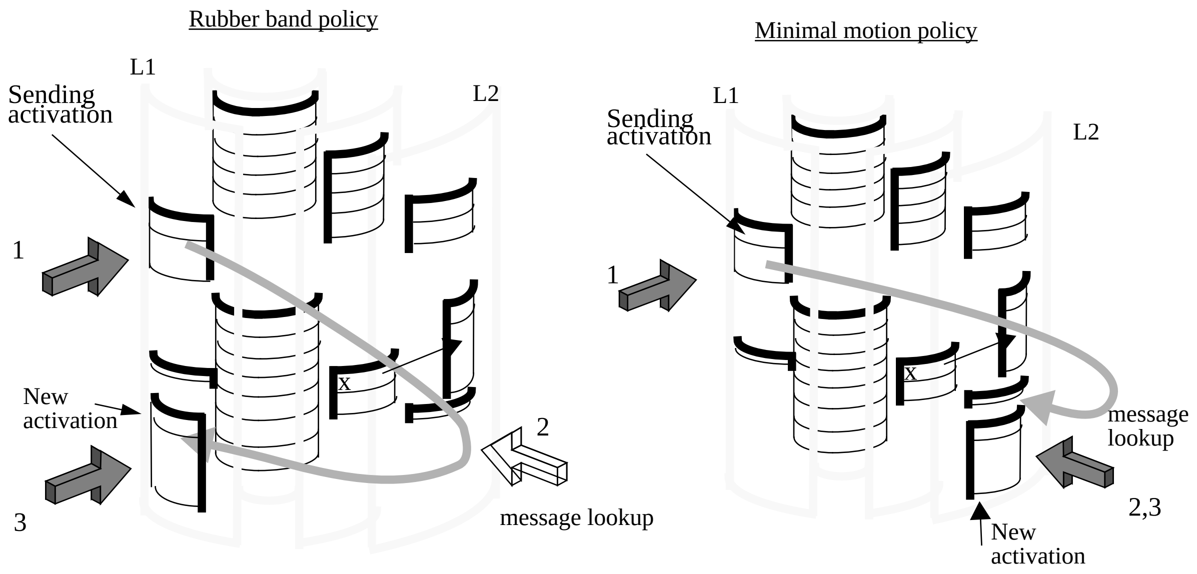


Figure 9. Two possible meanings for sending a message from one perspective through another. In general, when a message is looked up and a method is found, a new activation is created and installed as part of the lexical space of available attributes. In the “Rubber band” model, shown at left, when x is sent and slot x is found in the perspective L2, the resulting method activation is created *in the sending perspective*, L1. In the “minimal motion” policy, shown at right, the new activation is created at the site of the perspective L2. The system will only return to L1 when the new activation returns. The minimum motion policy, which we employ, is useful and in keeping with perspective-receiver symmetry.

Multiple inheritance: Self supports multiple inheritance, and if we believe in perspective-receiver symmetry, perhaps we should allow for multiple layer parents too. Our Us implementation currently only supports single inheritance in the layer parent dimension. While this is a simplification of sorts, multiple layer parents are conceptually reasonable: a point of view can derive from several other viewpoints. Such a “mixin” approach to layer combination could, for example, allow one to temporarily inherit from the debugging layer and a colleague’s programming changes layer, in order to help debug his code.

Perspectives for removing attributes: Is it possible to use a layer to *remove* an attribute from an inner layer? This kind of attribute masking can be supported in the normal inheritance domain by some kind of “anti-attribute” attribute (sometimes called attribute “underriding”). For symmetry, a subjective system could introduce two directions for underriding of attributes: one for the normal inheritance direction, and one for the layer composition direction. Another solution is to have a attribute masking path that searches for attributes to cancel by following the subjective lookup path: inward through layers then upward through inheritance links.

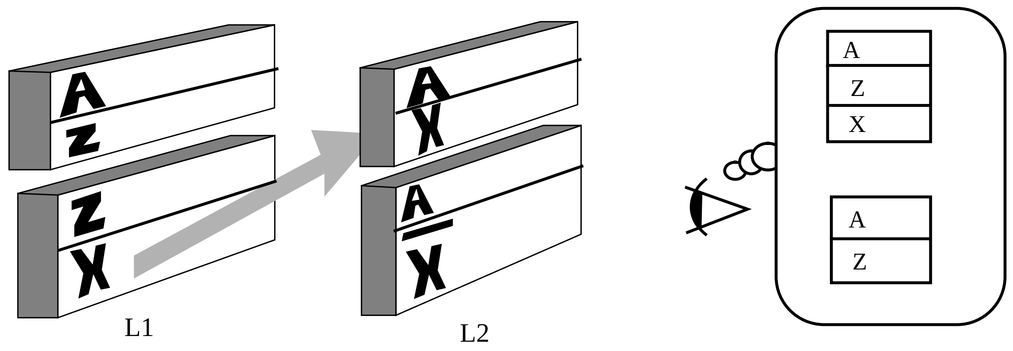
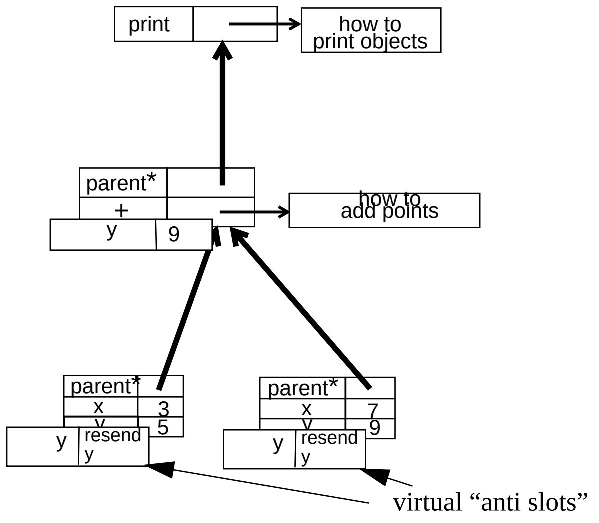


Figure 10. Using perspectives to move an attribute to another object. This can be achieved by some sort of anti-attribute mechanism with which an outer layer can mask the existence of an attribute on an inner layer. If the attribute can hold a method, then a “call-next-method” or “super” resending facility can be used (See Figure 11 on page 18).

However, our approach is to use a normal inheritance resend (sometimes called “super” or “call-next-method”) invocation to achieve the anti-attribute effect. This is illustrated in Figure 11 on page 18.

Again, for symmetry, we provide a layer-wise resend in addition to the normal resend for the inheritance direction. Another possibility is to provide a single resend which searches for the next invocation by following the subjective lookup path (first through layers, then up through inheritance links). However, this kind of resend could not be used to achieve the anti-attribute effect.

Figure 11. Layers and pieces, applied to Self. Here are two Cartesian point objects. Normally points have x and y slots, with a parent link for inheritance of shared operations such as addition. The point parent in turn may have a parent, inheriting even more generic slots that may provide, for example, operations for printing. In the situation at right, a layer has been added in which a “y” slot has been added to the point parent, and essentially deleted from the individual points, through use of resend (analogous to “super” or “call-next-method”). Thus, these points will share a common y value from this perspective,. In the original perspective however, each point still retains its own y.



3.3. Us: subjectivity applied to Self

We have applied this piece-layer approach to the language Self, and we call the resulting language Us in honor of the multiplicity of perspectives. We begin this exposition with a brief review of Self.

Self objects are simply a collection of slots. As mentioned earlier, each slot has a name and a reference to some other object. A slot can refer to a method, or it can refer to an ordinary non-method object, in which case a slot acts like a variable. When a message is sent to an object, the object looks for a slot name to match the message name, and the result of activating the slot’s method is returned (if the slot contained a method), or if it contains a non-method, that non-method object is simply returned. If no match is found, the lookup is passed on to the objects referenced in *parent* slots (a parent slot is a special slot bearing a mark recognized by the virtual machine). Self has no class/instance distinction: any object can inherit directly from any other object (object inheritance by implicit delegation).

In Self, everything happens by message sending. There are no variables as such, since there is no code in Self that can tell if a slot contains a method or non-method object.

A subjective language will require a syntactic extension beyond the objective object-oriented language, to allow explicit specification of the perspective. The syntax for any language can be extended for subjectivity by performing the transformation

$$\texttt{expression} \xrightarrow{\textit{to add subjectivity}} \texttt{expression} \otimes \texttt{perspectiveExpression}$$

Meaning that the messages in `expression` will be sent from the `perspectiveExpression` perspective. For example, in (objective) Smalltalk or Self, you might extract the contents string for a letter by evaluating

```
documents letterToHome contentsString
```

but in a subjective version you might see

```
(documents letterToHome ⊗ documents version3) contentsString
```

The implication here is that the `documents` object has many pieces out there with `letterToHome` slots: we are sending from a perspective which will select some appropriate version. The use of the ⊗ marker is meant to evoke a sense of the direct product space of receivers and layers. Just as *self* can go without saying in some languages, it is important to avoid a recursive infinity-of-perspectives overflow by allowing the current perspective to go without saying, and we introduce the reserved word *here* which can be elided from the syntax.

```
something ⊗ here <==> something
```

So in Us, the following are equivalent:

```
message <==> self message <==> self message ⊗ here.
```

Of course, the equivalence set for `message` is infinite, including things like

```
message ⊗ (self here ⊗ (here ⊗ (self here ⊗ (here ⊗ here))))
```

and so on. But we have a kind of fixed point in this implicit infinite regress because

```
here ⊗ here
```

is identical to

```
here.
```

(In loose terms, this means that the current view of the current view is, in fact, the actual current view. This could be argued as being philosophically tautological, since by "current view" we mean the perspective that goes without saying.)

It seems that, like death and taxes, unbounded meta-regressions will always be with Us. Unfortunately, if everything is subjective, and if perspectives are objects, then one must face this idea of perspectives on a perspective (and perspectives on perspectives on...). To bring this down to earth, one might have a perspective, `Pdave` representing Dave's view of the system and another, `Prandy`, for Randy's. But, suppose views of the system change with time. Then we might have a meta-perspective, `Pmon`, that when used to view `Pdave` or `Prandy`, yields Monday's version, and another `Ptues`, that yields Tuesday's. One might then find expression like

```
    (document ⊗ (Pdave ⊗ Pmon)) browse.
    (document ⊗ (Prandy ⊗ Ptues)) browse.
```

When `Pdave` is sent (to self, which is elided) it will be done from the `Pmon` (Monday) point of view. The resulting perspective is used to specify which `document` slot is found.

We allow perspectives to be combined by assigning to the `layerParent` slot, which is used to control "inheritance" in the layer dimension.

```
L1 layerParent: L2   "Set L2 to be the layer parent of L1"
```

A common cliche is

```
receiver message ⊗ (layer copy layerParent: here).
```

which essentially attaches "layer" to the end of the current chain of layers. The copy prevents interference between concurrently active threads, each of which may be trying to re-parent layers. Again, we are building on parallel functionality between the receiver space and perspective space, because in Self, parent slots can be assignable (a feature called "dynamic inheritance.")

Because a parent slot can be moved around among layers, perspectives in Us have the potential to express alternate views on the inheritance hierarchy. This is potentially useful for situations in which an object may want to be viewed as having different kinds of behavior in different contexts.

The current Us implementation is "cohabitant" with Self: any Self object can be viewed as an Us object with a single piece on the root layer. As long as there is only the one root layer, Us acts like a slow version of Self. For every object, there is an Us interpreter which traps when sent a message and figures out which slot to activate. Methods are activated in the context of the interpreter itself, so further message sends will also be interpreted. An Us interpreter is created for the resulting object, and returned: the per-object Us interpreters are only created as needed. Of course much more efficient implementations should be possible, but for now we are able to carry out small experiments, and the implementation has forced us to be specific about the semantics.

4 Subjectivity as a unified solution to many problems

In designing programming languages, each additional concept burdens every person who must learn the language. Therefore, a concept merits consideration only if its inclusion will add to the utility of the language in many situations.

Subjectivity is such a radical departure from conventional OOPL design that it must meet a high standard of ubiquitous utility. In this section, we illustrate our formulation of subjectivity by showing how it can be applied in many situations.

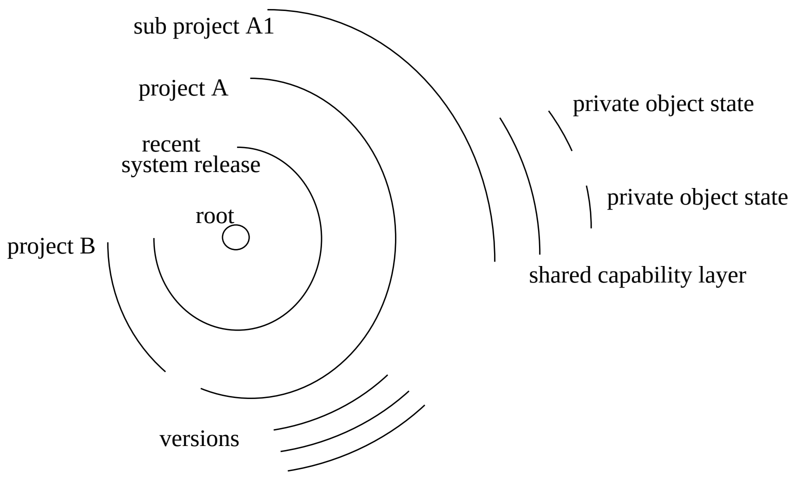


Figure 12. Many uses for perspectives. This is an overhead view of the layers in a system. Typically the outermost layers are more ephemeral. Layers from inner realms are more widely available, and are usually installed earlier. Layers on the outside tend to be more recent, and can even come and go with a single method activation.

4.1. Encapsulation

As illustrated earlier, subjectivity can be used to express encapsulation. We revisit our economic simulation example in light of the “perspective as layer sequence” picture, to see more about how the particular framework of Us would be used.

In order to encapsulate private attributes such as `balance` and `log`, account objects would embed them in a private layer. The account would retain a reference to a private layer and would use that layer as the perspective for the object’s implementation. Of course such a private layer can have non-empty pieces for objects anywhere in the system, even

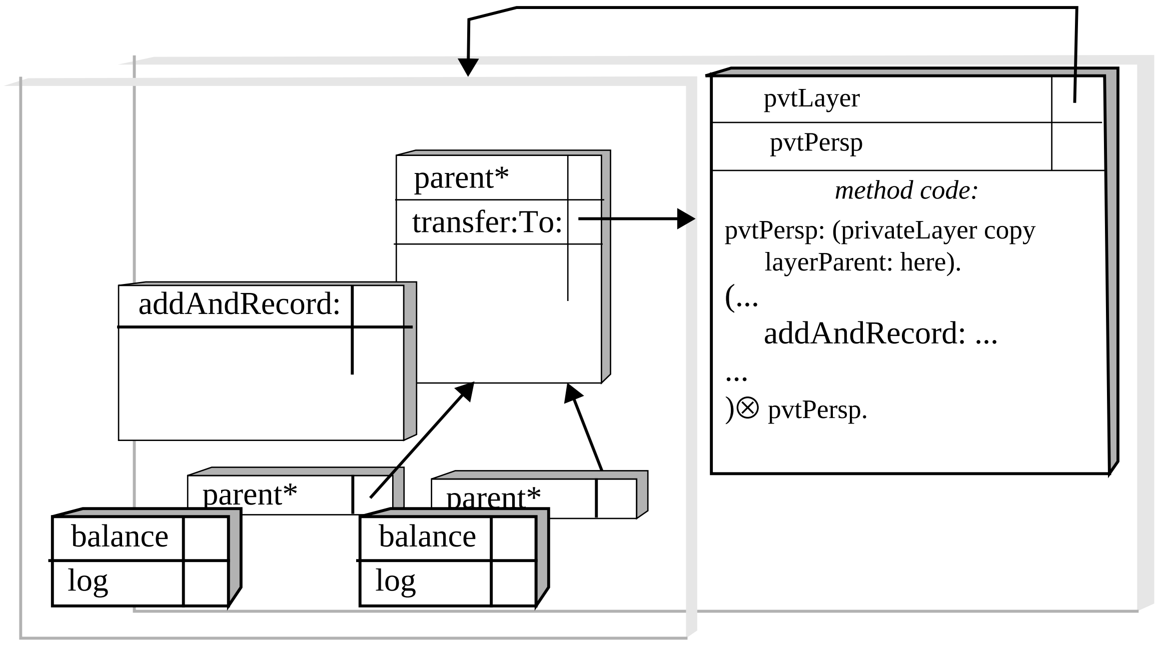


Figure 13. A layer used for encapsulation of account objects: a fuller elaboration of the earlier example. Here are two account objects sharing a common parent. In the `transfer:To:` method code, the private layer is needed so that the `addAndRecord:` slot will be visible. For encapsulation, the wise thing to do is to copy the private layer and append it as a layer child of here (the current perspective). This enables all changes to the current perspective to remain in force while the encapsulated code is accessed and run.

for those objects way up near the inheritance root of the hierarchy. This can be useful for a group of objects who wish to cooperate and share state in a private layer. Even though these objects may have only the inheritance root parent in common, they can still share private state and behavior by having a piece for this root in a commonly shared layer.

Encapsulation is perhaps the application which sees the most rapid changes in perspectives. Typically, the layer with the encapsulated slots would be copied, then appended as a layer child of the current perspective. In this way the privacy layer is merged onto the current perspective whether it happens to be a standard default perspective, or a temporary test of somebody's changes. Copying is a protection against a concurrently active thread's interference with the re-parenting of our layer. Since copying is "shallow," and an Us layer actually contains (considered as an object) just a few slots, copying is not as heavyweight as it might seem.

Conflict may emerge in such situations: a layer may have a critical slot unintentionally overridden by a private layer. However, we doubt this will be a major problem for Us, since we find that name collisions are not common in Self. We would not expect Us to have more or fewer slot names than Self, Us would simply group the slots into layers.

4.2. Encapsulation and Security

Our discussion in the previous section has been about how Us supports modularity and encapsulation for software engineering concerns, such as the narrowing of the public message interface, the maintainability of code, and the reusability of objects. Security is, we believe, a different though related concern. The separation is clear in Smalltalk, for example, where instance variables are normally considered encapsulated state. But in most Smalltalk systems it is easy to “break into” an object and gain a reference to the object stored in an instance variable. Encapsulation in Smalltalk seems to work well for software engineering issues, even while failing to provide perfect security. Nevertheless, it is interesting to speculate on how security might be addressed by the layer mechanisms in Us.

Suppose the account objects we have been modeling wish to make their `balance` and `log` slots secure — inaccessible from the outside world. As a first step, the accounts may wish to “hide” the reference to their private layer by placing it in a slot within a method, as done in Figure 13 on page 22. This means that a reference to an account does not automatically yield a reference to the perspective, because no message sends can access the internals of a method — sending a message to an object can only cause a method to run. Of course, programmers actually can modify methods and install new method objects in method-containing slots, but this requires reflective capabilities. In a full-blown Us implementation, reflective capabilities might be jealously guarded in private layers.

But even this is not sufficient to guarantee security. For example, while the accounts’ private layer is in effect (while it is part of the current perspective), the `addAndRecord:` method could be running, and might need to do some routine operation — concatenate two strings, for example. If you know this fact, and can reprogram the code for string concatenation, you can catch the current perspective while the string concatenation method is called from the supposedly private perspective. Thus security may require extremely careful or very limited use of private perspectives, and/or private copies of public operations.

There may be other moves one could take within the Us model to protect a private layer, but our sense is that the game would be a long one with many moves, counter moves, and counter counter moves. Security does not emerge in an elegant way from this model, though a more complete study of such issues would be an interesting avenue for further research.

4.3. Multiple views

The MVC (model-view-controller) paradigm is a classic user interface mechanism that lets an object have multiple views. In this design, the model represents the data, and the view is responsible for creating a graphical manifestation. By separating the view from the data, the problem of multiple graphical representations is neatly solved: many views may share the same model.

However, a separated view object must respect the encapsulation boundaries of the model. Our dictionary, for example, holds keys and values in internal structures that can be accessed more quickly by a direct route than through the normal dictionary protocol.

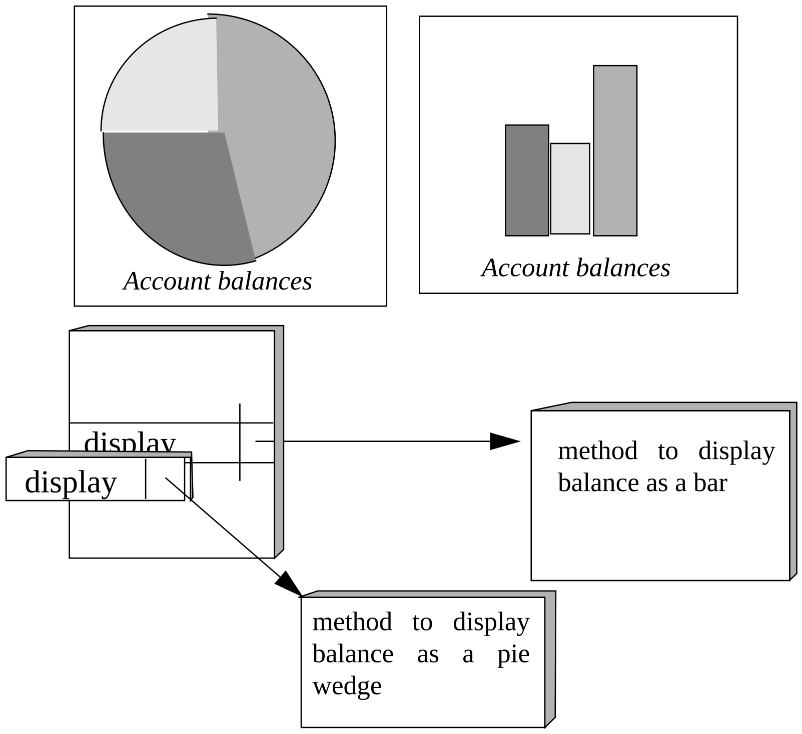

Account balances
Account balances
display
display
method to display balance as a bar
method to display balance as a pie wedge

Figure 14. The method to display the accounts balance data as a pie chart is in a separate layer, but will contain code that runs in the context of the receiver. The various viewing mechanisms can be separated into layers, while still allowing methods to access encapsulated state, since they will run in the context of the receiver.

With subjectivity, views can be both separate and integrated. Instead of separating viewer from viewee with object identity, a subjective language like Us can use multiple perspectives. The code to view an object in a particular way can be part of that object, yet in a different perspective. Multiple perspectives can be used to obtain multiple views. Although each view is separated by dint of being on its own *piece*, each view is at the same time inside the object it views, so that the view can be as tightly coupled to the object as needed.

4.4. Reflection

In the last few years, many researchers have proposed that the operations on objects as objects per se be formally separated from the operations on an object that it provides to stand for something in the problem domain. The domain of objects as objects is called the reflective domain. For example, in Smalltalk, which does not provide formal reflection, each object responds to `instVarAt:`, but in 3-KRS[M] each object instead has an associated meta-object containing the corresponding methods for the object's instance variables. Likewise in Self, each object has an associated *mirror* object that implements its reflective behavior. (In Self, mirrors serve only for structural, not behavioral reflection.)

However, we have found the meta-object approach can at times be awkward, because confusion can arise as to whether to pass in an object or its meta-object as an argument to certain methods. In such situations, sometimes we Self programmers yearn for the “one reference does it all” days of Smalltalk programming. Worse yet, there is more than one kind of reflection; for example structural reflection provides access to structure, and behavioral reflection, which allows for behaviour modification. Modularization would suggest that each object would need a different associated meta-object for each kind of reflection, and this multiplicity would further exacerbate the problem.

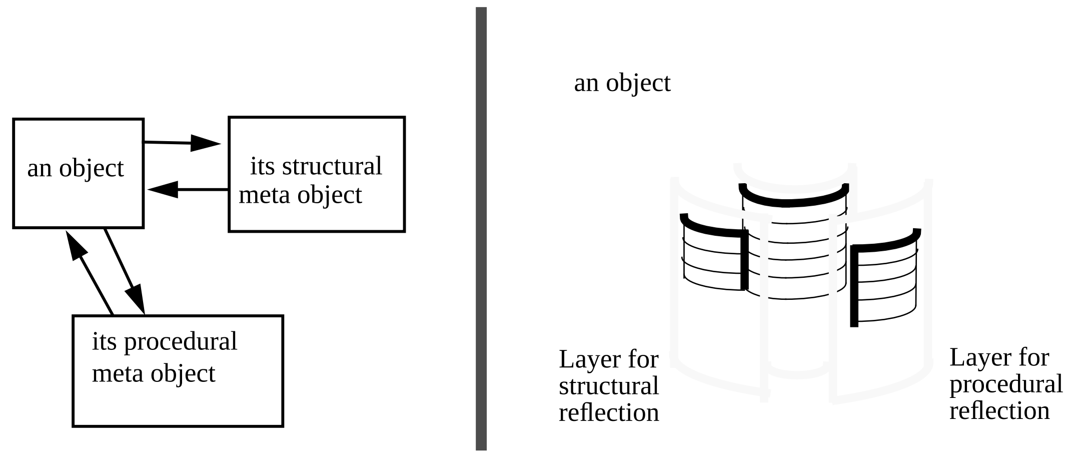

an object
its structural meta object
its procedural meta object
an object
Layer for structural reflection
Layer for procedural reflection

Figure 15. In most reflective architectures, there is a separate meta object for every object. With layers, reflection can be provided by perspectives. Users of reflective and nonreflective layers pass around a single object reference.

Subjectivity offers a way out. Instead of dealing with a cluster of different objects, we can provide different perspectives, one for non-reflective, ordinary computation, and one for each kind of reflection. Since such a system would always pass around pointers to the same kind of object, the reuse barriers erected by meta-objects would disappear.

4.5. Debugging the user interface

When writing user interface code in a language whose environment is written in itself, like Smalltalk or Self, bugs can be especially annoying because the errors may break the debugging tools. For example, every time a Smalltalk programmer changes the `display` method for a component of the browser, he might break the very browser he needs to repair the damage, resulting in a crash which would force him to recover from the last backup copy of the system. However, if the programmer follows the discipline of putting changes into a separate layer, the debugger can insist on switching to an underlying “safe” layer in which the user interface elements are known to function. Here we benefit from the minimum motion policy, in that the debugger catches a few messages that start it operating, then quickly switches to its safe perspective. Once switched, the debugger runs in the safe perspective until the initial activation returns. Once a set of changes has been tested, it can be moved from the experimental perspective to the standard one. Or, other users can change their perspectives to the new one in order to preserve both versions of the system.

4.6. Summary of Examples

Although many modern programming systems include facilities for multiple users, multiple programmers, privacy, reflection, and multiple graphical views, they need different facilities for each. A subjective language provides one facility that can be reused to provide each of these capabilities. In many cases the resultant structure is less awkward and more general. It would seem that the effort expended to embed subjectivity in the programming language and the extra learning required of users may well pay off in a simpler system overall.

5 Conclusions

In the beginning, there was Algol-60, and the target of a subroutine call was uniquely determined by the name of the subroutine. Then came Simula-67, and the run-time type of one parameter joined with the message name to select the code: objects added a new dimension to the computational domain. But the two-dimensional domain of receiver and message fails to capture many common situations, and is relatively naive compared to what we know about naturally occurring systems. The third dimension, subjectivity, is provided by including a sending perspective.

The quintessential manifestation of subjectivity is that all references to an object need not give access to the same state and behavior. We have described a general recipe for how to achieve a particular kind of subjectivity starting with an object-oriented system with inheritance: apply the piece and layer concepts so that the layer domain functions in a way similar to the ordinary inheritance domain. This perspective-layer symmetry provides a conceptual economy we feel is crucial in reducing the complexity that is unleashed by introducing a new dimension into message sending.

We have applied this principle to Self, creating the language Us. The principle leads us to represent perspectives as layers, arranged in a hierarchy similar to but distinct from the inheritance hierarchy. An object then becomes a collection of pieces scattered across layers, and the symmetry principle gives rise to particular semantics for attribute combination among the pieces on various layers. The symmetry principle also leads us to implement a “minimal motion policy” for the semantics of method activation.

Our effort is largely one of design, with our current implementation too limited to really be put to the test. But we believe we have shown how the subjectivity in Us can be applied to a wide range of problems. We also demonstrate that the subjective approach has the potential to simplify systems by collapsing solutions to problems as disparate as encapsulation, multiple user interface views, and version control into one mechanism.

6 References

[BDGKKM] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common Lisp Object System Specification X3J13 document 88-002R. Sigplan Notices, 12, 1988

[BHMPZ] S. Ballard, C. Hibbert, P. McCullough, A. Purdy, and F. Zdybel, “Amber Language Specification,” unpublished internal report, Xerox Palo Alto Research Center, 1989.

[BKKMSZ] D. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, “CommonLoops: Merging Lisp and Object-oriented programming,” in Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Portland OR, ACM, Sept. 1986, pp 17-29

[BW] D. Bobrow, T Winograd, et. al., “Experiences with KRL-0: One Cycle of a Knowledge Representation Language,” in Proc. Fifth International Joint Conf. on Artificial Intelligence (Aug 1977), pp 213-222.

[GD] I. Goldstein and D. Bobrow, “A Layered Approach to Software Design,” Technical Report CSL-80-5, Xerox Palo Alto Research Center, Dec 1980.

[HK] R. Hull and R. King, “Semantic Database Modeling: Survey, Applications, and Research Issues,” Technical Report TR-86-201, Computer Science Dept., University of Southern California, April 1986.

[HO1] B. Hailpern and H. Ossher, “Extending Objects to Support Multiple Interfaces and Access control,” IEEE Trans. on Software Engineering, Vol. 16, no. 11, Nov 1990, pp 1247-1257.

[HO2] W. Harrison and H. Ossher, “Subject-Oriented Programming (a Critique of Pure Objects),” in Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Washington, D.C., ACM, Oct 1993, pp 411-428.

[HZ] S. Heiler and S. Zdonik, “Object Views: Extending the Vision,” in Proc. 6th International Conf. on Data Engineering, Los Angeles, IEEE, Feb 1990, pp86- 93

[K] A. Kay, “New Directions for Novice Programming in the 1980’s,” unpublished internal report, Xerox Palo Alto Research Center, 1982.

[M] P. Maes, “Concepts and Experiments in Computational Reflections,” in Proc. ACM Conf on Object-Oriented Programming Systems, Languages, and Applications, Orlando, FL, ACM, Dec 1987, pp 147-155.

[MMN] O. Lehrmann Madsen, B. Møller-Pedersen, K. Nygaard, *Object-Oriented Programming in the Beta Programming Language*, Addison-Wesley, 1993.

[OH] H. Ossher and W. Harrison, “Combination of Inheritance Hierarchies,” in Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Vancouver, B.C., ACM, Oct 1992, pp 25-40.

[OH2] H. Ossher and W. Harrison, “Support for Change in RPDE^{3}in RC 15866 (#69975), Research Report, IBM Research Div., T.J. Watson Research Center, Yorktown Heights, NY, May 1990.

[TYI] K. Tanaka, M. Yoshikawa, K. Ishihara, “Schema Virtualization in Object-Oriented Databases,” in Proc. 6th International Conf. on Data Engineering, Los Angeles, IEEE, Feb 1990, pp 23- 30.

[U] D. Ungar, “Annotating Objects for Transport to Other Worlds,” in Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Austin TX, ACM, Oct. 1995, pp 73-87.

[US] D. Ungar and R. B. Smith, “Self: The power of simplicity,” in Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications, Orlando FLA, ACM, Oct. 1987, pp 227-242.

[WKOST] S. Watari, S. Kono, E. Osawa, R. Smoody, and M. Tokoro, “Extending Object-Oriented Systems to Support Dialectic Worldviews,” in Selected Technical Reports, May 1990, Sony Computer Science Laboratory, Inc., Tokyo, pp 61-71.