Classes versus Prototypes in Object-Oriented Languages*

Alan Borning

Abstract

Smalltalk uses classes to describe the common properties of related objects. Unfortunately, the use of classes and metaclasses is the source of a number of complications. This paper discusses prototypes as an alternative to classes and metaclasses. In a prototype-based language, copying rather than instantiation is the mechanism provided to the user for making new objects. Inheritance constraints are proposed as a way of representing object hierarchies and supporting the automatic updating of related objects when edits are made.

1 Introduction

The Smalltalk-80 language [8], as well as a number of other object-oriented languages, uses *classes* to describe the common properties of related objects. Unfortunately, classes and the class-instance relation are the source of a number of complications. First, for an object to have a distinct message protocol, a separate class must be created for it. If, as in Smalltalk, classes themselves are objects, then to allow different classes to understand different initialization messages, each class must itself be an instance of a different class (called a metaclass in Smalltalk). Metaclasses add to the complexity of the language; a recent study [14] on difficulties encountered in teaching and learning about Smalltalk indicates that metaclasses are uniformly regarded as the single worst barrier to learnability by both teachers and students. Second, the emphasis on classes in the programmer's interface is at odds with the goal of interacting with the computer in a concrete way. When designing a new object, one must first move to the abstract level of the class, write a class definition, then instantiate it and test it, rather than remaining at one level, incrementally building an object. This problem is most apparent in systems for graphical or visual programming.

The alternative suggested in this paper is the organization of the programming environment around *prototypes* rather than classes. A prototype is a standard example instance; new objects are produced by copying and modifying prototypes, rather than by instantiating classes.

The remainder of the paper is organized as follows. The following subsections list some sources of complexity in the current Smalltalk metaclass-class-instance mechanism, and then describe a small gedanken experiment in language design, in which prototypes are used instead of classes. This very simple language has several limitations, and a more realistic design is presented in Section 2. Following this is an enumeration of the advantages and disadvantages of the proposal. The final section provides comparisons and references to related work.

1.1 Some Sources of Complexity

One source of the complexity surrounding classes in Smalltalk is the interaction of message lookup with the role of classes as the generators of new objects, which gives rise to the need for metaclasses. Another source is the use of classes for several different functions.

*Originally published in *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, Dallas, Texas, November 1986, pages 36–40.

In Smalltalk, when an object receives a message, the interpreter goes to the object's class and looks in its method dictionary for a method for receiving that message. If a method isn't found, the interpreter looks in the class's superclass, and so on up the class hierarchy. Classes are themselves objects, and to make new objects, one sends appropriate messages to classes. In general these messages will vary from class to class. For example, to make new points, one wants to send an *x:y:* message to the class Point, so that the x and y coordinates for the new instance can be passed as arguments. Given the way message lookup is done, this requires that the class Point be an instance of a different class from (say) class Rectangle, which should not understand the *x:y:* message, but rather a different initialization message specific to rectangles. This pragmatic need for class-specific initialization methods was satisfied by the introduction of metaclasses: each class is an instance of a separate metaclass. However, as noted above, this design decision has had unfortunate consequences for the teachability and learnability of the language.

In regard to the use of classes for several different functions, some of the roles that classes play in Smalltalk are as follows:

- generators of new objects
- descriptions of the representation of their instances
- descriptions of the message protocol of their instances
- elements in the description of the object taxonomy
- a means for implementing differential programming (this new object is like some other one, with the following differences . . .)
- repositories for methods for receiving messages
- devices for dynamically updating many objects when a change is made to a method
- sets of all instances of those classes (via the *allInstances* message)

While some of the above items are related, it is clear that classes in Smalltalk are playing multiple roles.

1.2 A Gedanken Experiment in Language Design

To help unravel the complexity, let's do a small gedanken experiment in language design. Considerations of space and time efficiency are to be ignored for the moment, to avoid needlessly intertwining semantic and implementation considerations.

Suppose that objects are completely self-contained, so that an object consists of *state* and *behavior*. One can send messages to an object asking it for information, asking it to change its state, or asking it to change its behavior. The only way to make a new object is to make a complete copy of an existing object, copying both state and behavior. Once the copy is made, there is no further relation between the original and the copy. (Creating new objects by copying eliminates the need for metaclasses, since creation and modification messages are sent to prototypes or other individuals rather than to classes.)

This is a clean model, and would be easy to teach about. It handles object creation, modification, and representation. What is missing? First, there is no notion of classification of kinds of objects, either by message protocol or by representation. Second, there is no way to update a whole group of objects in a similar manner at one time (the equivalent of such actions as adding new methods to a class in Smalltalk). These are both important, and so the model needs to be augmented to support classification and updating.

2 A Proposal for a Prototype-Based Language

In this section a proposal for a more realistic language is presented, in which the simple model described above is augmented to support object classification and updating. Many of the ideas used here have arisen from the author's work on constraint-oriented languages and systems [1, 2, 3], where a constraint describes a relation that must hold. In this proposal, constraints are used to express inheritance relations among objects. However, the set of inheritance constraints used here is limited and straightforward to maintain, and a general-purpose constraint representation and satisfaction mechanism is not required.

In this proposed language (as in the language described in the gedanken experiment), an object has state and behavior. The state of an object is represented by a set of named *fields*. We will on occasion be interested in an object's field names, and this list of names can be accessed separately from the contents of the fields. There are two components of an object's behavior. The first component is a *method dictionary*, which is similar to that in Smalltalk, except that there may be several methods for receiving a given message. (The way in which one method is chosen, or several are combined, is discussed in Section 3.2.) The second component is a *protocol* that describes the set of messages the object declares that it can understand, the protocols required of the arguments to the messages, and the protocols of the results returned by the messages.

New objects are produced by copying other objects. Thus, to make a new point, one would make a copy of the prototypical point, with new values substituted for the x and y fields. Once a prototype has been copied, there would be no hidden relation between the prototype and the copy; any further relations that were desired would be explicitly represented using constraints.

2.1 Inheritance Constraints

The proposed language does not include a general constraint mechanism. Rather, there is a fixed set of *inheritance constraints*—constraints on an object's field names, methods, and protocol—built into the language. These are as follows:

inherits-field-names(x,y) This constraint holds if every field name of y is also a field name of x.

inherits-behavior(x,y) This constraint holds if all of the methods in y's method dictionary are also in x's method dictionary.

inherits-protocol(x,y) This constraint holds if the protocol of y is a subset of the protocol of x, i.e., if every message that y can understand is also understood by x, and if each object returned by x in response to one of these messages also obeys the corresponding protocol declared in y.

In general these three constraints are independent. For example, an object x might inherit the protocol of y but not its methods (x would implement the necessary methods in completely different ways). If an object y does not use all of its fields, then x can inherit the behavior of y but not all its field names. Of course, the constraints are not totally independent—for example, if x inherits behavior from y, all of the field names used by y must also be field names of x.

Nevertheless, it will often be the case that these three constraints will be applied together, and so a *descendant* constraint is defined as follows:

descendant(x,y) $\equiv$ inherits-field-names(x,y) $\wedge$ inherits-behavior(x,y) $\wedge$ inherits-protocol(x,y)

2.2 Object Creation Messages

There are two messages available for creating new objects: *copy* and *descendant*. The *copy* method makes a complete copy of the receiver and returns it. There is no further relation between the receiver and the copy. The *descendant* method makes a copy, and also sets up a one-way descendant constraint between the original and the copy.

2.3 Examples of Use

In place of the class Point would be a prototype point. The prototype point would have x and y fields, each initialized to 0. (Alternatively they could be left as *nil*.) It would understand messages such as *+*, *printOn:*, and so forth. To make a new point, one would evaluate

```
point x: 4 y: 6
```

which would make a descendant of the prototype point, and then set its fields to 4 and 6. The code for *point x:y:* is as follows:

```
x: newx y: newy
  ↑ point descendant setx: newx sety: newy
```

The message *setx:sety:* is defined as in Smalltalk:

```
setx: newx sety: newy
  x ← newx.
  y ← newy.
```

The following messages would build a new kind of object, *threepoint*, and define an addition method for it.

```
threePoint ← object descendant.
threePoint hasFields: 'x y z'.
threePoint hasMethod:
  '+ p ↑ threePoint x: x + p x y: y + p y z: z + p z'
...
```

Naturally, there would be user interface support for the creation and modification of prototypes. This could be done using a browser, which could have much the same appearance and functionality as the browser in the current Smalltalk system.

3 Implementation

The *descendant* method should be implemented as a primitive, and the primitives for *new* and *new:* eliminated. It might also be useful to implement *copy* as a primitive. To make new variable-length objects, one would make a copy or descendant of an appropriate prototype, and then grow or shrink the copy as needed. It might be useful to combine copying and growing in *descendant:* and *copy:* methods.

To test the scheme, it could be implemented using the present Smalltalk bytecode set by simulating the new primitives. Classes would be given an additional field named *prototypes* that points to the collection of prototypes for that class. (Usually, a class would have a single prototype, but multiple prototypes are possible.) Below is the Smalltalk code for simulating *Object copy* and *Object descendant*, along with some auxiliary methods.

```
copy
  "if I am a prototype, need to copy my class; otherwise just
   make a simple copy"
  self isPrototype
    ifTrue: [↑ self class copy prototype]
    ifFalse: [↑ self simpleCopy]
```

descendant
```
self bePrototype. "make sure that I'm a prototype"
↑ self simpleCopy
```

simpleCopy
```
"return a complete copy of me, taking account of
 substructure. IdentityDictionary is a
 dictionary in which == is used for key comparison"
↑ self copyWithDict: IdentityDictionary new
```

copyWithDict: dict
```
  | copy f |
(dict includesKey: self) ifTrue: [↑ dict at: self]
"make the shell of the new copy, then fill it in"
copy ← self class new.
dict at: self put: copy.
1 to: self class instSize do:
  [:i | f ← (self instVarAt: i) copyWithDict: dict.
      copy instVarAt: i put: f.
↑ copy
```

bePrototype
```
"make me into a prototype, if I'm not already"
  | newSelf |
self isPrototype ifTrue: [↑ self].
newSelf ← self class newSubclass prototype.
1 to: self class instSize do:
  [:i | newSelf instVarAt: i put: (self instVarAt: i)].
self become: newSelf.
```

isPrototype
```
↑ self class prototypes includes: self
```

These general methods would be overridden for classes, and for primitive objects such as numbers.

3.1 Classes

For objects such as points and rectangles, the field names, method dictionary, and protocol will be the same for many objects. In an implementation, then, it is reasonable to group these together into a class. Further, rather than having multiple methods in a given dictionary, the methods could be partitioned among sub- and superclasses, as in Smalltalk. Thus, there would be classes as well as prototypes. However, classes in general won't have global names, there won't be any metaclasses, and the user will usually interact with a prototype rather than a class.

3.2 Multiple Inheritance

An object can have *descendant* constraints that relate it to several parents, i.e. multiple inheritance is supported. Each of the three constraints that compose the *descendant* constraint (*inherits-field-names*, *inherits-behavior*, and *inherits-protocol*) establishes the correct relation when multiple inheritance is used. The only difficult question is which method (or methods) to execute in response to a given message, if there are several conflicting inherited methods. However, this problem arises

equally in class-based systems, and the same sorts of choices are applicable. For simplicity, for the present the rules described in the Smalltalk multiple inheritance implementation [4] are to be used.

4 Evaluation

In this section, a (doubtless biased) listing of the benefits and drawbacks of the proposed scheme as opposed to that in the standard Smalltalk-80 language is presented.

4.1 Benefits

Some benefits of this scheme over the current one are:

- The initial explanation of the language is much simpler. One can just talk about objects as having state and behavior.
- Even at a deeper level, there are fewer concepts—metaclasses are no longer needed. It is simpler to explain how message lookup occurs when teaching about the language.
- Fields are automatically given default values in newly created objects (by copying the contents of the corresponding field in the prototype).
- A prototype-based language would provide better support for concrete, visual programming systems.
- Any object can be given individualized behavior. This could be useful for example in debugging, when one might want to set a halt in the method for one particular object, and not for all instances of a class.
- The semantics of inheritance are described in terms of constraints; sharing is regarded as simply an implementation technique. This distinction might be useful when building distributed systems, systems that run on multiprocessors without shared memory, object servers, or the like. In such systems, sharing would *not* be the only technique used to implement inheritance.

4.2 Drawbacks

- The use of prototypes seems natural for things like windows, but unnatural for such basic objects as integers. What is the prototypical integer? The decision here is that *all* integers are prototypes, that is, that a change to the methods of any integer affects all integers. Alternate possibilities are that 0 or 1 is the prototypical integer, or else that the prototype is a special integer whose value is undefined. None of these possibilities seems completely natural.
- A related drawback is that the concreteness of prototypes may be inappropriate for describing such standard data structures as stacks or queues. Rather than talking about a prototype stack, one wants to talk about stacks in general.
- There is a danger of inadvertently modifying a prototype. One can of course inadvertently modify a class in Smalltalk, but this seems less likely since it has a different message protocol from its instances.
- There is an efficiency problem in regard to making new objects by copying. For example, when building a new rectangle the system will make a copy of the prototype rectangle, probably only to replace its *origin* and *corner* fields immediately with new values.

The first drawback listed above applies to the extreme situations in the language. This seems to be analogous to the situation in Smalltalk itself. Objects and messages are a great idea most places, but they seem bizarre for things like integers ("3 + 4 means sending the message +4 to the object 3??"). However, the benefits of uniformity are such that making 3 be an object that understands messages is the right choice in Smalltalk.

Regarding stacks and similar data structures, the explicit existence of classes in this scheme may a help, since one can still talk about classes if one wants.

Regarding inadvertently modifying a prototype, I believe that the solution to this is not to introduce a different message protocol for prototypes, but to introduce some form of protection. For example, one might make prototypes read-only except in particular environments. One should also be able to designate some messages (e.g. *setx:sety:* for points) as being private, and allow this message to be sent only by *self*.

Regarding the efficiency problem, an obvious step is to code the *descendant* primitive efficiently. Beyond that, it appears that code sequences like *point descendant setx: newx sety: newy* will occur frequently. Therefore, in addition to the *descendant* primitive, for each object the system might automatically compile a method for e.g. *descendantWithx:y:*, which would accomplish the same thing, but more efficiently. This would call a new primitive that makes a clone of the receiver and then sets all of the instance fields in the clone. (Since this message exposes the object's representation, it is best regarded as a private initialization message.)

5 Related Work

The idea of prototypes is not new, and discussions of prototypes from a variety of perspectives appear in the literature. (However, the idea of using constraints to establish and maintain inheritance relations does appear to be new.)

One sort of system in which prototypes have often been used is systems for visual or concrete programming. In such applications, prototypes are more useful than classes, since it is more straightforward to display them for viewing and manipulation by the user; their concreteness also makes them valuable for less experienced users. Examples of systems of this sort that have been built in Smalltalk include ThingLab [1, 2], Programming by Rehearsal [9, 10], the Alternate Reality Kit [16], and Animus [5, 6].

Languages in the Actor family are general-purpose programming languages that use prototypes. Rather than inheritance, the Actor languages use a more general concept of *delegation*, in which any object may be delegated to handle a message for another; Lieberman [13] provides a useful and readable discussion of both prototypes and delegation. LaLonde [12] describes an exemplar-based Smalltalk (an exemplar is the same as a prototype); this language allows a given class to have multiple exemplars, an idea that has been borrowed and used in the design described here. In Biggertalk [11], an object-oriented language implemented in Prolog, instances are like classes in all respects, except that they cannot be further refined. Finally, prototypes are often used in artificial intelligence representation languages [7] to store default or typical information.

The language proposed in Section 2 does not include type declarations. However, if type declarations were to be added, protocols would be the logical entity to use in the declaration and checking of type. An object-oriented language that does have strong typing, along with a separation of protocol and implementation, is Trellis/Owl [15].

Acknowledgements

Thanks to Dave Robson, Randy Smith, Adele Goldberg, Tim O'Shea, and other members of the System Concepts Laboratory at Xerox PARC, and to David Notkin and Andrew Black at the University of Washington. This research was sponsored in part by the Xerox Corporation, and in part by the National Science Foundation under grants MCS-8202520 and IST-8604923.

References

[1] Alan Borning. *ThingLab—A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford, March 1979. A revised version is published as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).

[2] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[3] Alan Borning. Constraints and functional programming. In *Proceedings of the Sixth Annual IEEE Phoenix Conference on Computers and Communications*, pages 300–306, Scottsdale, Arizona, February 1987. IEEE.

[4] Alan Borning and Danial H. H. Ingalls. Multiple inheritance in smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237, Pittsburgh, Pennsylvania, August 1982. American Association for Artificial Intelligence.

[5] Robert Duisberg. Animus: A constraint based animation system. In *CHI'86 Proceedings*, pages 131–136, Boston, April 1986.

[6] Robert A. Duisberg. *Constraint-Based Animation: The Implementation of Temporal Constraints in the Animus System.* PhD thesis, University of Washington, 1986. Published as UW Computer Science Department Technical Report No. 86-09-01.

[7] R. Fikes and T. Kehler. The role of frame-based reasoning in representation. *Communications of the ACM*, 28(9):904–920, September 1985.

[8] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[9] Laura Gould and William Finzer. Programming by rehearsal. Technical Report SCL-84-1, Xerox Palo Alto Research Center, May 1984.

[10] Laura Gould and William Finzer. Programming by rehearsal. *Byte*, 9(6):187–210, June 1984.

[11] E. Gullichsen. Biggertalk: Object-oriented prolog. Technical Report STP-125-85, MCC, November 1985.

[12] W. R. LaLonde, D.A. Thomas, and J. R. Pugh. An exemplar based smalltalk. Technical Report TR-94, Computer Science Department, Carleton University, Ottawa, Canada, 1986.

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

[14] Tim O'Shea. The learnability of object-oriented programming systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, page 502, Portland, Oregon, September 1986. ACM.

[15] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to trellis/owl. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 9–16, Portland, Oregon, September 1986. ACM.

[16] Randall B. Smith. The alternate reality kit: An animated environment for creating interactive simulations. In *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages*, Dallas, Texas, June 1986. IEEE.