The Stripetalk Papers: Understandability as a Language Design Issue in Object-Oriented Programming Systems

T.R.G. Green, A. Borning, T. O'Shea, M. Minoughan, and R. Smith
Rank Xerox EuroPARC, Cambridge, U.K.

Foreword

This paper (originally just titled "Understandability as a Language Design Issue in Object-Oriented Programming Systems") grew out of work done at Rank Xerox EuroPARC during the summer of 1988, in which we designed a series of experiments to compare the understandability of various features of object-oriented languages. A major question was the understandability of prototype-based vs. class-based languages. The work was quite preliminary, but we thought we'd try sending it to ECOOP'89. The paper was rejected — perhaps the program committee wanted running implementations or actual results of experiments or some similar unreasonable demand. However, an earlier and shorter draft (known as "The Stripetalk Paper") circulated in the prototypes community in the following years: a kind of cult classic, perhaps, for an extremely small cult.

The paper that follows is basically the same as the rejected ECOOP paper, and is presented as an historical document. Perhaps readers will still find the space of language alternatives it presents of interest.

Abstract

Language features that influence the learnability and understandability of Smalltalk (O'Shea 1986) have motivated us to explore alternative language designs. We present a language space whose dimensions correspond to the ways of dealing with these identified learnability-influencing features. For example, one dimension represents how new objects are created in a language; we consider prototype-based copying and class-based instantiation. We establish the positions of some existing languages in this space and locate several new ones. Some of these languages may hold promise for improved learnability and understandability, and we propose a mechanism for testing them with comparison studies. An important observation from the current explorations has been that it is necessary to consider language features in the context of the tools provided by the environment; the effectiveness of a particular linguistic feature is enhanced or diminished by the degree of environmental support provided for it.

Keywords: Understandability, usability, object-oriented programming, language design.

1 Introduction

Our starting point is that object-oriented languages are significant members of the family of programming languages, and that Smalltalk is among the most mature of such languages and environments. Smalltalk also has a mature user community: there are effective communities of Smalltalk users with adequate teaching experience, and a number of teaching and reference texts are now available (e.g. Kaehler and Patterson, 1986; Pinson and Wiener, 1988). However, beginners experience difficulties in learning and understanding the language (O'Shea 1986). We believe that some changes to the language would result in significant gains in both learnability and understandability.

This paper starts from Smalltalk and explores how some of the possible alternative design decisions affect learning and usability. We do this by listing the linguistic features that seem to us to be particularly important in establishing the flavour of Smalltalk, and then envisaging a language in which an alternative decision was made. In this way we can generate a sort of giant cube of hypothetical languages. We are looking for languages at the extreme points on the cube, rather than slight variations, so as to better explore the space of possibilities. A dominant theme, but by no means the only theme, is whether prototypes and copying could make a useful alternative to classes and inheritance, as suggested by Lieberman (1986), Borning (1986), and others. Some of the languages already exist, others are designed as interesting and reasonable languages in their own right, and finally some are extreme designs that are interesting for doing comparative experiments, but not as languages that one would realistically expect to be used for programming.

We are building on our previous work in this area, in particular the empirical studies by Tim O'Shea on Smalltalk learnability (O'Shea, 1986), and the Deltatalk proposal by Alan Borning and Tim O'Shea for a simplification of Smalltalk-80 (Borning and O'Shea, 1987). In the present experiments, however, we are proposing more than the modest Delta changes.

The changes that we are proposing are not solely linguistic. An effective programming system demands the support of adequate programming tools, and Smalltalk, with its various browsers, notifiers and inspectors, does an unusually good job in that respect. Some of our design proposals would reduce the quality of the Smalltalk programming system unless additional features were introduced into the environment; for example, dispensing with the class inheritance system would mean that mass updates to objects, normally performed by editing the class definition, would become very long-winded unless some tool were provided. So as part of discussing linguistic proposals we believe that it is necessary to mention their implications for the programmer's tools.

It is important to recognise that we are not so much interested in particular languages as in the learnability and understandability of various language features. Therefore, we are relatively unconcerned with implementation problems, execution speed, and memory usage. Implementation techniques are rightly an active area of research, and some of the new languages we present could create problems that would severely tax the creativity of any implementor. However, we believe that gaining insight into the way that particular features contribute to understandability and learnability would be the most salient contribution for future designers of systems and languages. Restricting ourselves to languages that are realisable on current hardware is not the best way to explore the space of language features.

Our approach stands in contrast to that of Wegner (1987) who maps out the design space of object-based languages according to different aspects of computational behaviour, rather than understandability issues. The two approaches are equally valid views and have some similarities; it is simply that the motivation behind each is different. However, we would argue that Wegner's view that an object-based language is object-oriented only if its objects belong to classes and behaviour is shared via inheritance through the class hierarchy is unnecessarily exclusive since the functionality provided by classes and inheritance can be provided by a variety of systems, including systems in which state and/or behaviour is held locally to the object, such as prototype-based systems.

2 Making Evaluative Comparisons

Outlining the design choices is of little use unless some basis for choice can be found. Many techniques for comparing the learnability, understandability, and usability of languages have been reported in the literature (Soloway and Iyengar, 1986; Olson et al., 1988) but standard laboratory studies are extremely expensive in time and effort. We propose therefore to restrict ourselves to a single aspect of evaluation, viz. understandability, especially since the understandability of Smalltalk emerged as a significant problem in the studies reported by O'Shea (1986). The discussion of particular languages in the following sections is therefore to be read in the context of our concern with "understanding how simple things are done in the language by watching demonstrations."

We do not, of course, believe that understandability means nothing more than that; nor that understandability, even in the widest sense, is the only way in which to evaluate language designs; but we do believe that it is an important test, on which existing OOPS designs could do better.

The conventional experiments on programming language comprehensibility have frequently relied on paper and pencil techniques. This is inappropriate to a language with as powerful an environment as Smalltalk and the related designs we discuss here. We therefore propose to use a 'Watch and explain' game. For this game, we intend to devise written materials discussing each language and video tapes showing how operations are performed. Recent evidence suggests that watching videos is an excellent way to explain powerful environments (Payne et al., in prep.).

The subjects will be given some material explaining one of the languages, and then asked to read the written examples or to view the tape. They will then be asked questions of the form "what happened in the machine when the user did x" or "what would happen if instead the user typed y". From this data we hope to glean information about ways to improve the understandability of object-oriented languages.

The environments in each case will be Smalltalk-like, except as changes are required by particular features of different languages. (Of course, the environment is critical in learnability and usability; we just don't want to take on too much for the first experiments.)

3 Linguistic Features

There is some indication that the following linguistic features contribute to the understandability of an OOPS, from empirical studies (O'Shea and Borning, 1986) and also through anecdotal teaching experience (e.g. see the 'learnability' panel session in OOPSLA 1986).

These features are:

- presence or absence of metaclasses
- classes or prototypes for generating new objects
- explicit representation or not of abstract message protocol (separate from concrete implementation)
- kind of mass updating facility: shared classes, dependency links, indirection, retrieval and edit, none
- technique for handling references to overridden inherited methods: none, "super" as in Smalltalk, "boxed methods"
- presence or absence of delegation
- presence or absence of assignment statements
- updating and browsing built into the language or provided by the environment.

Here is a brief discussion of each of these items.

3.1 Presence or absence of metaclasses

A metaclass is simply a class whose instances are themselves classes. Any system in which every object holds its behaviour in a class, and in which classes are objects, will have metaclasses. Metaclasses were identified by O'Shea (1986) as a severe hurdle for learners of Smalltalk, and it is important to investigate the effects of dispensing with them.

An alternative design decision would be to preserve metaclasses but to make their use less difficult. Smalltalk learners encounter metaclasses early, because class-specific instantiation of new objects requires a separate metaclass for each class. If that requirement for an early encounter could be avoided, learners might find that they were mature enough in their use of an OOPS to comprehend metaclasses readily when they did come to them.

It should be noted that it is possible to use classes to hold behaviour and for classes not to be objects, but this solution has generally been less favoured than that which makes everything in the system be an object, thus enhancing consistency.

3.2 Use of classes or prototypes for generating new objects

Although OOPS grew up around the notion of class, alternatives are available. If classes are used, new objects are created by sending a "new" message to a class. In contrast, if prototypes are used, new objects are created by copying and modifying existing ones.

One disadvantage of classes, pointed out in Borning (1986), is that to change the message protocol of one object in a class-based system, i.e. to have instance-specific behaviour, it is necessary to create a new class for that object. In this case a prototype-based system has a clear advantage in that an individual object can be modified directly.

A widely-held belief is that although classes are useful, building a class hierarchy and getting it right is difficult. Lieberman (1986) and Borning (1986) have independently argued that it is difficult for the programmer designing new objects (or rather a class hierarchy for new objects) to start from the most abstract level and proceed to the more concrete, and that prototype systems have a usability advantage because they allow the programmer to operate at the concrete level. Note that this applies to both the sharing of behaviour through inheritance as well as the determination of what comprises the state of an object (i.e. what its instance variables are), and that the two are independent. That is, it would be possible to have a prototype-based system which used classes to share behaviour between objects. The idea that it is better for the programmer to be able to work at the concrete level seems intuitively reasonable, but at present there is no strong empirical support to indicate that this is the case. If it were the case then this would be an argument in favour of a system which used prototypes to make new objects and in which all behaviour for each object was held locally. (It may be that in the implementation of such a system behaviours would be shared between objects for space efficiency, but in such a way that this would not be visible to the user.)

Two kinds of prototype-based system can be envisaged for making new objects. In the 'Platonic' system, all similar objects, say all cows, are made by first copying from a prototype 'ideal' cow and then changing attributes (colour, age, name, etc.) as required. The 'ideal' cow serves only as a template for building new objects. In the 'Aristotelian' system there are only 'real' cows. To make a new cow, any already-existing cow may be copied. We have investigated the Aristotelian system, which is inherently simpler.

3.3 Explicit representation or not of abstract message protocol

An object's message protocol is what it presents to the outside world. In theory one should be able to substitute one object for another as long as it obeys the correct message protocol, irrespective of its implementation. This is important, for example, if one wants to change the concrete representation of an object — as long as the protocol remains the same, no changes to the users of the object are needed. An important part of the design task in object-oriented programming is deciding on the message protocols, since it is these protocols that specify how the objects may be used. However, in Smalltalk, for example, there is no explicit representation of protocol as an entity distinct from the object's implementation. There are such entities in such statically-typed object-oriented languages as Emerald. ("Statically typed" means that the type of every expression can be determined and checked for consistency at compile time. This allows type errors to be detected at compile rather than run time.) In more conventional languages such as Ada and Modula-2, the package or module specification is analogous to a representation of abstract message protocol; it is widely accepted by designers and users in this school that this is a Good Thing, assisting the understandability of inter-relationships within the program as a whole.

3.4 Kind of mass updating facility: shared classes, dependency links, indirection, retrieval and edit, none

In a class-based system such as Smalltalk, one can add or edit a method to some class and all instances of that class and of its subclasses will be immediately affected. This capability is not automatically available in a prototype-based system. Although it is likely to be more important for experts than for novices, informal evidence has indicated that even novices experience problems when this type of facility is not provided (Fischer and Lemke, 1988).

There are however a number of ways to support mass updating in a prototype-based system. Updating is an example of a feature which can be provided either by having a language with a shared information structure, such as a class hierarchy or shared ancestors, or by having an environment with specialised support for mass updating. Some alternative methods of updating are as follows.

First, we could use dependency links, in which the behaviour of an object B could be declared to be dependent on that of object A. (This is a simple variety of one-way constraint.) If a method in A were changed, and if B had the same method as the old version in A, then B's method would be changed in the same way. Also, if a new method were added to A, and if B didn't already have a method with the same selector, than the new method would also be added to B. This updating could be done automatically, or only upon an explicit request by the user.

Second, we could use indirection. B could have methods that said "look in the corresponding method in A"; as a result updating would be automatic. To run the method in B, we could either copy the method from A into B at runtime, or use delegation (Subsection 3.6).

Third, instead of performing mass updating through the use of shared information, we could use some sort of retrieval mechanism to retrieve all the objects that should be updated, and (in effect) update each of the retrieved objects, either automatically or interactively. This option relies on a programming environment with smart features, and is discussed below.

3.5 Technique for accessing inherited overridden methods: none, "super" as in Smalltalk, or "boxed methods"

Smalltalk includes a construct ("super") that allows a programmer to access inherited methods that have been overridden in the subclass. However, this construct is known to be confusing to novices (O'Shea, 1986), and has some odd properties. Simply eliminating it may well be acceptable in a language for novices, but probably wouldn't be in a language for experts, due to the loss in modularity. Without "super" or a similar mechanism, such as "resend" in Self, one would either need to copy down the code from the overridden method, or else split off an auxiliary method from the overridden one which could then be accessed both from the original method and from the method in the subclass. The copying technique is not satisfactory, since now there are two copies of essentially the same code, introducing an updating problem (the need to keep both copies updated). The splitting method also causes problems, since adding a subclass might require changes in the superclass; one would prefer that making a subclass could be done in a more modular way.

In a system in which behaviour and state are completely self-contained, "super" as such doesn't make sense. To support something like it, one possibility is to include inherited methods duplicated in the child (under a different selector). Another is to put a "box" around the portion of the new method that was inherited, so that it is identified as a unit for purposes of updating, whether by the programmer or by the system.

3.6 Presence or absence of delegation

Lieberman (1986) has proposed that delegation be used to support inheritance of methods in a prototype-based system, and also for a number of advanced programming applications (he gives a dribble stream as an example). We can and should separate delegation and prototypes: all four combinations of presence or absence of delegation and prototypes vs. classes make sense. In addition, we could still provide delegation for the advanced applications, while not using it routinely for inheritance.

3.7 Presence or absence of assignment statements

Assignment statements are ubiquitous in most imperative programming languages, and are also known to be a cause of confusion for novices (Soloway and Ehrlich, 1984, Gilmore, 1986). They are particularly strange in object-oriented languages as they are not consistent with the message-passing metaphor. The language Self does away with assignment statements, instead handling state change requests via automatically provided methods (Ungar and Smith, 1987).

3.8 Updating and browsing built into the language or provided by the environment

The choice of language features has a direct impact on what tasks the user will require to be supported by the environment. Some means of browsing related objects should be provided. Users should be also be provided with a 'power tool' for changing many objects at once — in Smalltalk for example all objects belonging to the same class can be changed simply by changing the class definition. There are of course many other possibilities for specifying the group of objects to be changed. These include retrieval by abstract message protocol, by ancestry, by concrete representation (presence of given instance variables), by arbitrary user-defined mark (or 'stripe' — see Subsection 4.1), by behaviour, or some combination of these.

4 A Cube of Languages

Figures 1 and 2 show a matrix of languages which touches on the corners of our "cube" of design choices. Some of these languages or variants listed above are already defined and in some cases implemented. These are Smalltalk (Goldberg and Robson, 1983), Deltatalk (Borning and O'Shea, 1987), CLOS (Bobrow et al., 1987), Self (Ungar and Smith, 1987), Emerald (Black et al., 1986, Hutchinson, 1987), and Trellis/Owl (Schaffert et al., 1986). The remaining 'languages', Stripetalk, Prototalk+Workbench, Delegation talk, and Sharetalk, are briefly described below. Stripetalk and Prototalk+Workbench are discussed in more detail than the others, since they are among the languages we plan to study.

	Sec. 3.1 Metaclasses present	**Sec. 3.2 Classes or prototypes**	**Sec. 3.3 Separate rep of protocol**	**Sec. 3.4 What is shared**
Smalltalk	yes	classes	no	classes
Stripetalk	no	prototypes	[yes]	nothing
Prototalk+ Workbench	no	prototypes	[yes]	nothing
Deltatalk	no	classes	no	classes
CLOS	yes	classes	no	classes
Delegation-talk	no	prototypes		prototypes
Sharetalk	no	prototypes		behaviours
Self	no	prototypes	no	prototypes
Emerald	no	neither	yes	none
Trellis/Owl	no	classes (“types”)	yes	classes

[brackets] indicates arbitrary choice; space indicates no choice yet.

Figure 1: The Language Matrix (Part 1)

	Sec. 3.5 How to handle super	**Sec. 3.6 Delegation present**	**Sec. 3.7 Assignment**	**Sec. 3.8 Updating mechanism**
Smalltalk	super	no	yes	shared classes
Stripetalk	boxed methods	[yes]	[yes]	search and replace
Prototalk+ Workbench	boxed methods	[yes]	[no]	workbench/ ancestry
Deltatalk	super	no	yes	shared classes
CLOS	super-ish	no	yes	shared classes
Delegation-talk	delegation	yes		shared prototypes
Sharetalk	super-ish			shared behaviours
Self	super-ish (resend)	yes	no	shared prototypes
Emerald	don't	no	yes	none
Trellis/Owl	super	no	yes	shared classes

Figure 2: The Language Matrix (Part 2)

In the language designs, we of course strived for a set of choices representing reasonable combinations of the chosen features. When it was necessary to make an arbitrary choice, we used the criterion of selecting extreme points in the space of languages.

4.1 Stripetalk

Stripetalk uses prototypes. Updating a single object is easily handled by sending that object an edit message. Mass updates are handled by retrieving all objects matching some description, and applying the same edit to all of them. To support retrieval, objects can be marked with zero or more 'stripes' (hence the name). Stripes would simply be symbols; they have no intrinsic meaning. One could thus retrieve all objects with the "point" stripe, or with the "fix me" stripe, or with both the "window" and "fix me" stripes. When a copy of an object is made, the copy will of course start with the same collection of stripes possessed by the original. The user can subsequently add or delete stripes from the copy.

We have also considered retrieval mechanisms that allow a variety of forms of specification. The strongest candidate for usability is a graphical template-based interface to support a broad range of retrieval requests. The template is partially filled in, and objects that matched the template are retrieved (a blank in a field matches anything). This is similar to query-by-example for databases. The template might specify for example a list of stripes such that objects with all of these stripes would be retrieved.

We consider Stripetalk to be somewhat radical in that sharing is not part of the language semantics, yet mass updating is supported by the environment. That all objects hold their own state and behaviour is, we consider, closer to the way objects exist in the real world. Since one of the claims for the benefits of object-oriented languages is that they are easier to use because they allow people to deal with objects as they do in the real world, then this would appear to be a significant advantage.

There were a number of arbitrary choices made in our language matrix for Stripetalk: it should have a separate representation of protocol, and it should support delegation and assignment.

4.2 Prototalk+Workbench

Prototalk+Workbench again uses prototypes. To support updating, the environment maintains a programmer's workbench database of where objects and methods come from. Thus if an object B is created by copying A, this is in the database. Similarly if method M2 is produced by copying method M1, this is recorded as well. (These links are permanent.)

When the user edits a method he or she is queried as to whether or not copies of that method should also be updated. The choices are:

- don't update any copies
- update all copies without asking
- ask

Choosing the "ask" option gives breadth-first enumeration of copies. For each copy, the choices are:

- update this copy and keep asking
- update this copy and all copies of it without asking any more
- don't update this copy; keep asking
- don't update this copy; stop asking for copies of it.

If a new method is added to or deleted from an object, the user is asked in a similar fashion whether to add or delete the corresponding method from copies of the object.

Access to overridden inherited methods is handled by the boxed-code approach. This seems to be the most logical choice for this language, since it fits in well with the programmer's workbench database — the idea of the workbench is to provide environmental support for identifying and updating portions of the code that determines an object's behaviour.

Arbitrary choices for Prototalk+Workbench are: separate representation of protocol, support for delegation, and no assignment.

4.3 Delegationtalk

This language uses prototypes. Shared prototypes and delegation are used for mass updating, and to handle access to overridden inherited methods. This may be thought of as Lieberman's (1986) language, translated into Smalltalk syntax.

4.4 Sharetalk

This is a minimal-change approach to introducing prototypes. Prototypes rather than classes are used, but objects are not completely self-contained. Rather, the behavioural portions are explicitly shared and updated, and come in hierarchies (like classes). So this ends up similar to Deltatalk, but using prototypes rather than classes.

4.5 CLOStalk

CLOS, being an embedded language, is very different in syntax from the other languages, making experimental comparison perhaps more difficult. However, it is possible to design a language variant with a Smalltalk-like syntax but that uses CLOS's approach to object creation, which — unlike Smalltalk — allows easy initialisation of new objects, without having to create a separate metaclass for each class. In CLOStalk, then, there are still classes and metaclasses. However, at least as far as beginners are concerned, all classes have the same protocol. To handle initialisation, the message "new" takes an optional list of keyword-parameter pairs. This list is passed on to the newly created instance for initialisation. For example, if we write

Point new x: 10 y: 20

we are sending the message "new" to the class Point with arguments "x: 10 y: 20". This pair of arguments can then be passed on to the new instance of Point as arguments to an "initialise" message.

5 Discussion

The most interesting consequences for understandability appear to flow from the decision to use prototypes rather than classes. Both our prototype-based designs seems promising. Stripetalk is in the 'shock-a-Smalltalk-wizard' category: simple, distinctly different from existing languages, and not easy to evaluate at present. In Stripetalk the set of objects retrieved for updating is arbitrary and under user control. In contrast, in Prototalk+Workbench the set of objects retrieved is rigidly specified (by ancestry).

This rigid specification in Prototalk+Workbench is ideologically pure, but will not be accepted by expert users, since it fixes the updating links once and for all. A better scheme for experts would be to have explicit behavioural dependence links in the database. These would start out the same as the original-copy links but could later be changed as the system evolves. One could also cut the links if one wanted to make independent objects.

What are the conditions of easy understandability in an object-oriented programming system? At present, of course, we have few grounds on which to make predictions. Typical criteria that are discussed from the armchair include the following.

- Concepts to be acquired gradually. Smalltalk requires learners to tackle the concept of 'class' before doing anything else much, and require the concept of 'metaclass' at an early stage of learning. Our feature set offers two alternatives design decisions: the learning of metaclasses can be postponed if CLOS-like initialisation is used; and classes themselves can be avoided if prototypes are used.

- Difficult concepts to be avoided. Present evidence singles out the metaclass as a prime example of a difficult concept (O'Shea, 1986).
- Fewer concepts to be learnt. This criterion (also termed 'brute elegance') is extremely important, yet it is hard to assess in a particular case. For instance, it is not clear whether the prototype-based schemes will turn out to ask more or less of learners than the class-based schemes. The ideas built into the language are fewer, but to compensate, they will have to learn to use the environmental tools more effectively. A lot will depend on investigating the possibilities for mass update and browsing.
- Premature commitment to be avoided during design. A characteristic of certain programming languages is the need to make decisions before the programmer is ready to do so. This has been noted in the case of Smalltalk by Goldstein and Bobrow (1981) whose PIE environment attempts to allow incremental, coordinated design of object hierarchies. We believe that the prototype-based systems may be another route to this goal.

Which of these criteria are genuinely important and yet also practicable? How should a language designer choose between one ideal and another? Our hope is that by investigating the consequences of a wide range of design decisions in the manner we have described, we shall be able to report that certain conditions must be met while others have less impact.

Acknowledgements

Many thanks to Rank Xerox EuroPARC to providing the environment in which these ideas could initially be developed, and for agreeing to allow this paper to be published years later.

References

Black, A., Hutchinson, N.C., Jul, E. and Levy, H. (1986) Object structure in the Emerald System. Proc. OOPSLA 1986 Conference on Object-Oriented Programming Systems. New York: ACM.

Bobrow, D.G. and Goldstein, I. (1981) An experimental description-based programming environment: four reports. Technical Report CSL-81-3. Xerox Palo Alto Research Center, Palo Alto, California.

Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S., Kiczales, G. and Moon, D.A. (1987) Common Lisp object system specification. ANSI X3J13, Document 87-002, American National Standards Institute. Washington D.C.

Borning, A. (1986) Classes versus prototypes in object-oriented languages. Proc. Fall Joint Computer Conference, ACM/IEEE, November 1986.

Borning, A. and O'Shea, T. (1987) Deltatalk: an empirically and aesthetically motivated simplification of the Smalltalk-80 language. Proc. ECOOP 1987, European Conference on Object-Oriented Programming.

Fischer, G., & Lemke, A.C. Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication. Human-Computer Interaction, vol. 3 no. 3 (1987-1988) pp 179-222, Lawrence Erlbaum Associates, Hillsdale, NJ, USA.

Gilmore, D.G. (1986) Structural visibility and program comprehension. People and Computers; designing for usability. Proc. 2nd. Conf. British Computer Society Human Computer Interaction Special Interest Group. University of York, September 1986.

Graube, N. (1988) Reflexive architecture: from ObjVLisp to CLOS. Proc. ECOOP 1988, European Conference on Object-Oriented Programming, pp 110-127.

Hutchinson, N.C. (1987) Emerald: an object-based language for distributed programming. Ph.D. thesis, University of Washington.

Kaehler, T. and Patterson, D. (1986) A taste of Smalltalk. New York: Norton.

Lieberman, H. (1986) Using prototypical objects to implement shared behaviour in object-oriented systems. Proc. OOPSLA 1986 Conference on Object-Oriented Programming Systems. New York: ACM.

Olson, G. M., Sheppard, S. and Soloway, E. Empirical studies of programmers: second workshop. Ablex.

O'Shea, T. (1986) Why object-oriented systems are hard to learn. Proc. OOPSLA 1986 Conference on Object-Oriented Programming Systems. New York: ACM.

Payne, S., Chesworth, L., Waterson, P. and Hill, E. (in preparation) Display animation as an orientation for exploratory learning.

Pinson, L. J. and Wiener, R. S. (1988) An introduction to object-oriented programming and Smalltalk. Reading, Mass.: Addison-Wesley.

Schaffert, C., Cooper, T., Bullis, B., Kilian, M., Wilpolt, C. (1986) An introduction to Trellis/Owl. Proc. OOPSLA 1986 Conference on Object-Oriented Programming Systems. New York: ACM.

Soloway, E. and Ehrlich, K. (1984) Empirical studies of programming knowledge. IEEE Trans. on Software Engineering, vol. SE-10, no. 5, September 1984.

Soloway, E. and Iyengar, S. (1986) Empirical studies of programmers. Ablex.

Ungar, D. and Smith, R.B. (1987) Self: the power of simplicity. Proc. OOPSLA 1987 Conference on Object-Oriented Programming Systems. New York: ACM.

Wegner, P. (1987) Dimensions of object-based language design. Proc. OOPSLA 1987 Conference on Object-Oriented Programming Systems. New York: ACM.