An object-oriented implementation of a graphical-programming system

G. S. Cunningham, K. M. Hanson, G. R. Jennings, Jr., and D. R. Wolf[*]
Los Alamos National Laboratory, MS P940
Los Alamos, New Mexico 87545 USA
email: cunning@lanl.gov

ABSTRACT

Object-oriented (OO) analysis, design, and programming is a powerful paradigm for creating software that is easily understood, modified, and maintained. In this paper we demonstrate how the OO concepts of abstraction, inheritance, encapsulation, polymorphism, and dynamic binding have aided in the design of a graphical-programming tool.

The tool that we have developed allows a user to build radiographic system models for computing simulated radiographic data. It will eventually be used to perform Bayesian reconstructions of objects given radiographic data. The models are built by connecting icons that represent physical transformations, such as line integrals, exponentiation, and convolution, on a canvas.

We will also briefly discuss ParcPlace's application development environment, VisualWorks, which we have found to be as helpful as the OO paradigm.

1 Introduction

In this paper we discuss the importance of OO concepts for software development in the context of a graphical programming tool. The graphical programming tool that we have built allows a user to instantiate data-transform icons and connect these transforms with lines to define a "data-flow" diagram on a canvas that appears on a workstation screen. The icons represent transforms that map input data to output data, e.g. the line integral transform of the input data, exponentiation of the input data, the addition of two data inputs, etc. The data-flow diagram represents a measurement system.

Our goal is to use the graphical programming tool in conjunction with a 3D radiographic object modeling tool that is still in the process of being built. The object-modeling tool will allow a user to lay down simple 3D shapes and then twist, warp, and deform them to create novel shapes. Furthermore, the user will eventually be able to identify parameters of the created object that are subject to uncertainty for Bayesian inference and hypothesis testing.

We believe that these tools will be useful to scientists and engineers for orchestrating Bayesian inference and hypothesis testing of geometric object parameters [12, 14] given real radiographic data [6]. The general problem for which these tools are intended is the determination of an object of unknown shape and distribution, described by a user-defined parameterization, given limited data generated by a well-characterized, user-defined measurement system.

[*]Supported by the United States Department of Energy under contract number W-7405-ENG-36.

The software described in this paper was written in an OO programming language, Smalltalk-80 [5], in the context of ParcPlace's supporting environment, VisualWorks [11]. We have found that the OO concepts of abstraction, inheritance, message-passing, encapsulation, polymorphism, and dynamic binding, are important in realizing our goal of a flexible and powerful software solution. In the discussion of the graphical programming software, we will provide examples of how each of these concepts was important.

The rest of the paper is organized as follows: the next section will define basic OO concepts. Section 3 will describe the graphical programming tool, with an emphasis on the software's structure and the importance of the OO concepts discussed in Section 2. Section 4 will describe Smalltalk-80 and VisualWorks and the impact both had on our software development.

2 The Object-Oriented Paradigm

The OO paradigm has recently attracted attention because of its promise for code re-use and ease of maintainence, in addition to the natural and intuitive language it promotes for discussion of software problems and their solutions [15].

2.1 OO concepts

Software development using the OO paradigm [3, 13] includes the same three phases that are used in other software-engineering methodologies: analysis (OOA), design (OOD), and programming (OOP). In OOA, OOD, and OOP, the concept of *class* and *object* is critical. A class is an *abstraction* of an object. A class is defined as a set of *methods*, or functionality, and *attributes*, or data. An object is a particular instance of a class, in the sense that it has specific values for its attributes. For example, a Car class may have a color attribute, while a Car object may have a red instance of the color attribute. The concept of *inheritance* allows the software developer to organize classes into a hierarchy, wherein *subclasses*, which are lower in the hierarchy, inherit methods and attributes from *superclasses*, which are higher in the hierarchy.

The "world outside" of an object can communicate with that object only through *messages* sent to it that request some method to be performed. The implementation of the method is entirely up to the object that possesses it, and is of no concern to the outside world. Thus, attributes are *encapsulated* by their methods so that internal data representations and implementations of data retrieval are unimportant to the outside world. That is, if the value of a certain attribute of object #1 is desired by object #2, object #2 has to send a message to #1 requesting the value of that attribute. Object #1 can then implement the retrieval of that attribute value in any way it desires.

If many objects can respond to the same message with potentially different implementations of the method associated with that message, we say that the system is *polymorphic*. For example, text strings, PostScript documents, and raster images may all know how to print, but will certainly have different implementations of the method associated with the same message `print`.

Finally, *dynamic binding* is the capability provided in some programming languages to omit typecasting, i.e. assigning a specific type such as integer, float, etc. The type of object bound to a message is determined at run-time rather than at compile-time.

2.2 OO Analysis and Design

Software engineering experts seem to be split over whether the OOA, OOD, and OOP phases should be distinct and sequential, or whether it is more desirable to use a recursive approach [17]. We have used a recursive approach, in part because many of the OOA tools require a detailed problem statement, which was difficult for us to create at the beginning of the project. We found that "use-cases" [8], which document a typical user's interaction with the final product, were the most helpful OOA tool for defining the scope of our problem, but we did not persist in their use long enough to prove them as valuable in a formal analysis or design.

Our approach was to rapid prototype a set of classes and then to use OOA tools (Object International's ObjecTool) to display the software's structure and discuss it. The next stage of rapid prototyping was based on these discussions. With the help of a consultant in OO technology we decided that we should use the Coad and Yourdon OOA/OOD methodology [3], which emphasizes the static nature of classes contained in the inheritance tree, in conjunction with some elements of Rumbaugh's methodology [13], which contains state diagrams for describing the dynamic nature of classes, although we have made little use of Rumbaugh's methodology thus far.

The OO concepts discussed in this section are important in all three phases of software development: OOA, OOD, and OOP. In our limited experience with OOA and OOD, we have found that the OO paradigm fosters creativity by making it easy to put aside implementation details. The OO concepts are important because they allow software developers to discuss project goals using a natural language - one which revolves around objects, their responsibilities or behaviors, and their attributes. However, we have more experience concerning the importance of these concepts in OOP, and so the next section will be focussed on illustrating their importance in OOP through examples drawn from our experience in building a graphical programming tool.

3 Discussion of the software

The graphical programming tool that we will discuss in this section is part of a larger project that is described in Section 1. The tool allows a user to construct measurement models for radiographic systems by graphically connecting transforms to define a data-flow diagram. The data-flow diagram represents the measurement process. For example, a simple system model might consist of a sequence of transforms including: 1) a line integral transform that takes line integrals of a radiographic object's attenuation profile in order to determine the pathlength of photons traveling through the object, 2) an exponentiation transform of the pathlength data to determine the average probability that a photon travels through the object unscattered along the given paths, and 3) a spatial convolution transform that describes the detector's blur function.

The graphical programming tool operates as follows. The user is presented with a window, or canvas, on which appear buttons that allow the user to add items to, or delete items from, the canvas. The user can add or delete `Transforms` and `Connections`. `Transforms` map input `Data` to output `Data` and are represented on the screen with a 32x32 bit-mapped icon. The user can specify the direction of the flow of data by connecting one `Transform` to another using a `Connector`, which is represented on the screen as a line segment between the two `Transform` icons that it connects. The user can move the `Transforms` on the canvas by clicking and dragging the cursor when the cursor is in the region of the screen owned by the `Transform`'s icon. The user can delete `Transforms`, and any `Connections` to the `Transform` are also automatically deleted. The user can break `Connection` lines so that the `Connection` is represented with connected line segments rather than just a single line segment between the two `Transform` icons it connects.

The `Transforms` are "living" objects, and the user can interact with them in several ways. The user can see a description of a `Transform`, and change the parameters that define it. For example, a `SetOfParallelLine-Integrals` has parameters that describe the angle(s) and separation of lines, a `Convolution` has parameters that describe the filter function, etc. The user can also message the `Transform` to display its output. This message is forwarded to the `Transform`'s `output` attribute, which is messaged to display itself. The fact that the `Transform` objects are alive distinguishes this graphical programming tool from one that allows a user to construct and visualize a script that contains a sequence of actions to be executed in a certain order [9].

In this section, we describe the `Transform`, `Data`, and `Connection` classes with an emphasis on demonstrating the utility of OO concepts.

3.1 Data class hierarchy

The `Data` class hierarchy is shown in Fig. 1. The classes whose names start with `Abstract` are so named because they are never instantiated, but provide a repository for methods and attributes contained by their subclasses. The letter `C` is contained in the names of all of the classes in this hierarchy because they all contain an attribute that points to external "C" programming language data structures, kept external to the Smalltalk programming environment for efficiency of execution reasons. This class hierarchy is intended to capture the structure and responsibility of vector and image data that will be manipulated by C programming language subroutines.

The subclasses of `AbstractCVector` include `Vector` and `Matrix` objects that store values or coordinates of float or integer types in C memory. `VectorOfValues` objects can perform pointwise transformations of themselves, using the exponential, logarithmic, square root, and trigonometric functions. They can multiply themselves pointwise by another `VectorOfValues`, subtract themselves, add themselves or copy themselves. A `FloatMatrixOfValues` object can multiply itself by a `FloatVectorOfValues` object. A `VectorOfValues` object can dot product itself with another `VectorOfValues` object. `FloatVectorOfValues` objects can be told to `display`. This message produces a line plot of the data. `FloatMatrixOfValues` objects can also be told to `display`. At present, this produces an `ImageAnalysisManager` that displays gray-scale views of the data and provides some tools for manipulation of the display. Each of these behaviors can be elicited by sending a single message to a `VectorOfValues` object.

In the graphical programming tool that we have described, `Data` are passed around by `Connectors` and manipulated by `Transforms`, and are naturally viewed as objects. Message-passing proves invaluable for making code that involves `Data` objects comprehendable. Most functionality can be elicited with a single command, e.g. `aFloatVectorOfValues display`, in which `display` is the message sent to the object `aFloatVectorOfValues`, `aFloatVectorOfValues exp`, or

`aFloatVectorOfCoordinates translateBy:aCoordinate`,

in which `translateBy:` is the message sent to the object `aFloatVectorOfCoordinates` and the object `aCoordinate` is the argument of the message.

Several other OO concepts are illustrated in this class hierarchy. Inheritance is used to indicate that all `CVectors` must have an address (pointer to C memory) and size attribute. The size attribute may be a single number, as for a `FloatVector`, or it may contain x and y components, as for a `FloatMatrix`. Although the message sent to each in order to retrieve its size is the same, the methods are implemented differently, an example of polymorphism. All subclasses of `AbstractCVector` possess the accessing methods, `at:anIndex` and `at:anIndex put:aCoordinate`. Sending the message `at:anIndex` to a `CVectorOfCoordinates` returns the (x, y) coordinate pair at the location specified by `anIndex`. The implementation uses the fact that the data are stored in C memory as $x00, y00, x01, y01, ...$, that is, as (x, y) pairs row-wise in one contiguous block.

On the other hand, sending the message `at:anIndex` to a `CVectorOfValues` returns the value at the location specified by `anIndex` (either an integer or float), another example of polymorphism.

The implementations for accessing, displaying, and manipulating are hidden from the user and so the data is encapsulated. If, at a later time, we wish to change the memory allocation protocol, we may also have to change the implementation of the `at:anIndex` method in some or all of the subclasses of CVectorOfValues, but we won't have to change how every other object elicits the same old behavior.

`FloatMatrixOfCoordinates` and `IntegerMatrixOfCoordinates` share all methods and attributes except the type of pointer to C memory that they have. The C memory is allocated, and the pointer to it defined, in the instance creation method

`FloatMatrixOfCoordinates new:aSize withAll:aCoord`.

Note that we are able to avoid typecasting the address attribute that was inherited, an example of dynamic binding.

3.2 Transforms

The `Transform` classes are relatively simple at this early stage in our software development. We have written classes for several categories of `Transforms`, including `MultiInputSingleOutput` (`Add`, `Multiply`, `Subtract`), `SingleInputSingleOutput` (`Exponential`, `Log`, `SqRt`, `Sin`, `Cos`, `LineIntegral`, `ParallelLineIntegral`) and no-input single-output (`Parameter` and its subclasses).

All `Transforms` inherit `output` and `dataSet` attributes. The attribute `output` is generated from `dataSet` using the subclass-specific method `generateOutput`, which first calls `generateDataSet` to get the current `dataSet` and then computes `output`. Finally, all `Transforms` know how to `displayOutput` (this message is forwarded to `output` which knows how to `display` itself).

All subclasses of `Transform` except `Parameter` share the method `generateDataSet`. `generateDataSet` queries the Connections that are connected to the `Transform` in order to find the ones which deliver input to it. These `Connections` are told to `generateOutput` and the return object is stored in the `dataSet`. `Parameters` do not depend on a `Connection` object to deliver their data, and so they over-ride this inherited method.

Subclass-specific transformations are in the method `generateOutput`, so that sending this message to any `Transform` will result in `output` being computed as a transformation of the input object, `dataSet`, an example of polymorphism. Again, note that we can decide to change the implementation of the transform in order to increase speed, etc., but that this does not affect the "rest of the world".

All transforms are implemented by telling items in `dataSet` to compute the transform associated with the `Transform` object that contains `dataSet` as input. In this way, a `LineIntegral` can take line integrals of all kinds of `Data` objects, since it just tells the `Data` object to take line integrals of itself. The responsibility lies with the `Data` object. This mechanism is called double dispatching, which is a way to implement dynamic binding. Another example of double dispatching is the following implementation of `aFloat addSelfTo:aNumber`. `aFloat` doesn't know the type of `aNumber`. `aFloat` could have logic that first determines what the type of `aNumber` is and then pick an implementation that is consistent with `aNumber`'s type (equivalent to a case statement in C), or it could simply say `aNumber addSelfToFloat:aFloat`. Since `aFloat` knows that it is a Float object, it can pick the proper implementation for `aNumber` to do the addition properly.

3.3 Connections

`Connections` merely transport `Data` from one `Transform` to another, which is a trivial responsibility in the context of the environment we are operating in at present, wherein `Data` is stored in memory on a single CPU computer. However, `Connections` might be very useful in a distributed environment or in an application where `Data` is stored in a database or even in files. Note that, even though a `Connection` merely passes along `Data` from one `Transform` to another, even this simple activity exhibits the important concept of dynamic binding. Since `Connections` don't know what type of `Data` they are passing, the reference to the passed object cannot be typecast. Similarily, the `Connection` doesn't know what type of `Transform` it is getting `Data` from and passing it to. Thus, the attributes that hold these input and output `Transforms` cannot be typecast. For example, the user might use a `aConnection` to connect `aFloatVector` to a `Convolution` or a `aFloatMatrix` to a `Convolution`, but this is not determined until the application is run, so that the input to `aConnection` cannot be typecast to either the `FloatVector` or the `FloatMatrix` class at compile-time.

3.4 Advantages of OOP

3.4.1 Ease in understanding software

The notion of objects is natural and intuitive since we think in terms of objects. Objects have responsibilities or behaviors that they must be able to perform and attributes that describe their current state. Behaviors of one object are elicited by another object through messaging. Objects are passed as arguments of messages, meaning that tremendous information can be passed into an object with a minimal amount of syntax. For example, the `generateOutput` method of aTransform might contain the single line `dataSet lineIntegralSpecifiedBy:aLineIntegralDescription`. The object `dataSet` might be a very complex 3D parameterization of a radiographic object. `aLineIntegralDescription` might be an attribute-rich object (like a C structure) that contains a long list of specifications that define lines along which dataSet is to compute its integral. Alternatively, `aLineIntegralDescription` might be a method-heavy object that produces the set of line integrals programmatically. `aLineIntegralDescription` might also be able to answer questions about what kind of set of line integrals it is so that `dataSet` could take advantage of particular regularities (like an equally spaced set of parallel lines, e.g.). Message-passing with object arguments encourages the programmer to write code that is compact and easy to comprehend.

Since many objects may have the same methods, the same message can be sent to different objects with a different implementation in each case (polymorphism), resulting in code that more closely parallels a natural language description of the software's function. Furthermore, the implementation is not important to the "outside world" (the data is encapsulated) and so internal data representation, accessing, and computation can be modified easily. Finally, inheritance trees put generic functionality higher in the tree so that only methods and attributes that differentiate a class from its superclass are contained in the class description. Inheritance makes it easier to comprehend classes, organize them, and re-use them.

3.4.2 Ease in extending software

The notion of objects puts an emphasis on responsibility, meaning that it is easy to determine where new functionality should be put – it will be a method or set of methods belonging to some particular class or set of classes. The inheritance tree allows the programmer to incrementally add responsibility and test it by subclassing (rapid prototyping). Encapsulation makes it easier to use the new functionality since the implementation details are hidden. For example, when we wanted to make line plots of `FloatVectorOfValues`, we created a new method called `display` for the class `FloatVectorOfValues`. Now, whenever an instance of `FloatVectorOfValues` is told to `display`, it knows how to do it. This new functionality was engaged immediately from other objects in the application. We didn't have to worry about all of the setup particular to `aFloatVectorOfValues`, e.g. the number of points in `aFloatVectorOfValues`, because this information is stored as attributes of `aFloatVectorOfValues`, and is used by the method `display`. We have found rapid prototyping to be a very important tool for testing ideas and stimulating new ones.

4 Programming language and environment

In this section, we will briefly discuss the impact that the programming language and environment has had on our software development. Before choosing Smalltalk-80 and the VisualWorks environment, we looked at one other option: using C++ as the programming language, InterViews and/or DEC's VUIT for building graphical user interfaces (GUIs), and CenterLine's ObjectCenter environment for code development.

We feel that the C++ language is lacking in many respects. The language is essentially C, which we feel does not encourage modularity. C++ enforces typecasting and dynamic binding is only allowed through the use of virtual classes. We also felt that InterViews and VUIT were inadequate. In fact, we could not find a good, portable GUI class library. InterViews and other packages are X-Windows based, but we are interested in transparent portability to PCs, which don't support X-Windows. DEC's VUIT allows MacDraw-like construction of GUIs, and generates the stubs that are associated with them, but, again, it is based on X-Windows. Code development environments, like CenterLine's ObjectCenter, still seemed immature to us. Finally, and most convincingly, while ParcPlace's VisualWorks melds the solutions to all of our requirements into a seamless environment that works on multiple platforms, using C++ would have required us to use several platform-specific tools that were not designed to work with one another.

4.1 Smalltalk-80

Smalltalk-80 is a pure OOP language, incorporating all of the desirable characteristics described above: classes, objects, inheritance, encapsulation, messaging, polymorphism, and dynamic binding. ParcPlace's class library contains several hundred useful classes and tens of thousands of methods. We have reaped many benefits by building our class hierarchy underneath the ParcPlace library through subclassing, especially as regards the GUI.

4.2 VisualWorks

ParcPlace's VisualWorks is a development environment that greatly enhances the programmer's productivity. VisualWorks includes a tool for building GUIs with MacDraw-like commands. Many components are provided, such as buttons, knobs, switches, sliders, text editors, etc. The graphical editor allows the programmer to define his own components and re-use them easily. The graphical editor also builds "stubs" for the methods connected to messages initiated by user interactions, e.g. when the user pushes on a button.

VisualWorks includes a debugger that allows easy access to the last several messages that were sent before an error occurred, or a user-written program "halt" or "notify" was encountered. Inspectors can be used to display the values of the attributes of the objects to which the last messages were sent. The ability to do incremental compilation of individual methods and the integration of the code editor, compiler and debugger mean that a seamless environment is provided for code development.

Furthermore, once the code is written, it can be run on a number of other platforms without modification. Smalltalk-80 and VisualWorks are suppported on DECStation, Sun SPARC, NeXT 3.0, Macintosh, MS-Windows, Sun Solaris 2.0, OS/2 2.0, IBM RS/6000, HP9000/700, and Sequent. Finally, ParcPlace provides the C Programmer's ObjectKit (CPOK) for integrating C and C++ code into your application. On some computing platforms, C/C++ code can be dynamically linked into the SmallTalk application.

4.3 Summary of Smalltalk-80 and VisualWorks

It has been suggested [4] that the OO paradigm may not be as important as the development environment in terms of boosting productivity. However, we feel that VisualWorks and Smalltalk-80 work hand-in-hand and that it is difficult to separate the contributions of one from the other.

The incremental compilation provided by VisualWorks is useful primarily because the piece of code being compiled (and debugged) is usually a method of an object rather than an arbitrary chunk out of a stream of code. Thus, when debugged and compiled, it truly provides an incremental extension to the functionality of the system and can be used immediately.

We found that when we initially began to use Smalltalk-80, debugging an entire application (collection of classes) took more time to do initially than debugging an equivalent piece of C code. The reason for this is that we were still learning the entire class library, which is inherited by our classes. Debugging is actually the primary means by which we learned the class library provided by ParcPlace. Once the steep part of the learning curve is overcome, though, the programmer finds that debugging OO systems is very efficient and sensible. Again, the browsers and inspectors for the code and data, which are part of the debugger, are much more useful because the code (methods) and data (attributes) are already organized via the class definitions and inheritance tree.

The main contribution that the VisualWorks environment makes, independent of the language, is the seamlessness with which the tools can be used together. The debugger contains a panel with a code editor and incremental compiler, a panel with an inspector on the object of current interest, and a panel containing a list of the most recent messages before the error or halt. Other browsers can be open simultaneously for a more thorough browsing of related classes. The ease with which a programmer can find errors, fix them, recompile, and restart the application profoundly enhances productivity. The environment encourages and rewards rapid prototyping and reverse engineering (recursive approach) for designing, implementing, and evaluating a system.

5 Conclusions

We have found that OO concepts play an important part in thinking about our software project and in building a solution. Since the world around us is filled with objects and their interaction, the OO paradigm meshes well with our thought processes. The basic human instinct to categorize and organize is well-suited to the construction of inheritance hierarchies. Abstraction and inheritance encourage the programmer to organize his understanding of the software and make changes to it that fit in the existing structure sensibly. The *real* ability to incrementally extend functionality using inheritance is evidenced by the dramatic increase in the availability of source code for purchase from software vendors (in the form of class hierarchy libraries) instead of executables.

Message-passing and polymorphism make the code easy to comprehend and prototype quickly since compact, readable software modules are encouraged. For example, ParcPlace recommends that Smalltalk-80 methods be no more than 5-7 lines long [16]. Encapsulation makes the code easy to extend since implementation specifics are hidden inside of objects that must fulfill their responsibilities via appropriate behavior, but can do the dirty details of that behavior in any way they like.

Our software resembles other applications that currently exist, some of which are also OO. For example, graphical programming tools like Khoros [9], AVS [1], IRIS Explorer [7], and VIVA [2], allow a user to place icons (that may represent data transforms) on a canvas and connect them to other icons, creating a data-flow diagram. There are also abundant applications for free-hand drawing and solid object modeling, as well as class libraries and OO graphical object descriptions [10]. However, our goals beyond the immediate project include having control over and access to the software environment so that it is readily modifiable and extendable for new projects and goals. Toward this end, we conclude that the choice of a robust OO programming environment is necessary for us to create a comprehensive class library that will be useful in our future efforts.

References

[1] Advanced Visual Systems Inc., 300 Fifth Ave., Waltham, MA 02154, e-mail address: avs@avs.com, ftp to avs.ncsc.org [128.109.178.23].

[2] Birchman, J.J., Tanimoto, S.L., Rowberg, A.H., Choi, H.S., and Kim, Y., "Applying a Visual Language for Image Processing as a Graphical Teaching Tool in Medical Imaging," Proc. SPIE, vol. 1653, pp. 379-390, 1992.

[3] Coad, P., and Yourdon, E., *Object-Oriented Analysis,* Prentice-Hall, 1991.

[4] Coggins, J., "Strategic Significance of Object-Oriented Design," Proc. SPIE, vol. 1898, pp. 690-701, 1993.

[5] Goldberg, A., and Robson, D., *Smalltalk-80: The Language,* Addison-Wesley, 1989.

[6] Hanson, K.M., "Bayesian reconstruction based on flexible prior models," *J. Opt. Soc. Am. A,* vol. 10, 1993, pp. 997-1004.

[7] IRIS Explorer Center (North America), 1400 Opus Place, Suite 200, Downers Grove, IL 60515-5702, e-mail address: infodesk@nag.com, ftp to swedishchef.lerc.nasa.gov.

[8] Jacobson, I., *Object-Oriented Software Engineering: A Use-Case Driven Approach,* Addison-Wesley, 1992.

[9] The Khoros Group, Room 110 EECE Dept., University of New Mexico, Albuquerque, NM 87131, e-mail address: khoros-request@chama.eece.unm.edu, ftp to pprg.eece.unm.edu [129.24.24.10].

[10] Koved, L., and Wooten, W.L., "GROOP: An Object-Oriented Toolkit for Animated 3D Graphics," OOPSLA 1993, Washington DC, pp. 309-325.

[11] ParcPlace Systems, 999 East Arques Avenue, Sunnyvale, CA 94086-4593, phone: (800) 759-PARC.

[12] Raloff, J., "Brain Warping," *Science News,* vol. 144, 1993, pp. 392-394.

[13] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modeling and Design,* Prentice-Hall, 1991.

[14] Szeliski, R., and Lavallee, S., "Matching 3-D Anatomical Surfaces with Non-Rigid Deformations using Octree-Splines," SPIE Geometric Methods in Computer Vision II, 1993, vol. 2031, pp. 306-315.

[15] Taylor, D. A., *Object-Oriented Technology: A Manager's Guide,* Addison-Wesley, 1990.

[16] Notes for "Introduction to Smalltalk", a course offerred by ParcPlace Systems.

[17] OOPSLA 1993, Washington DC, "Experience Reports" sessions. No published summary.