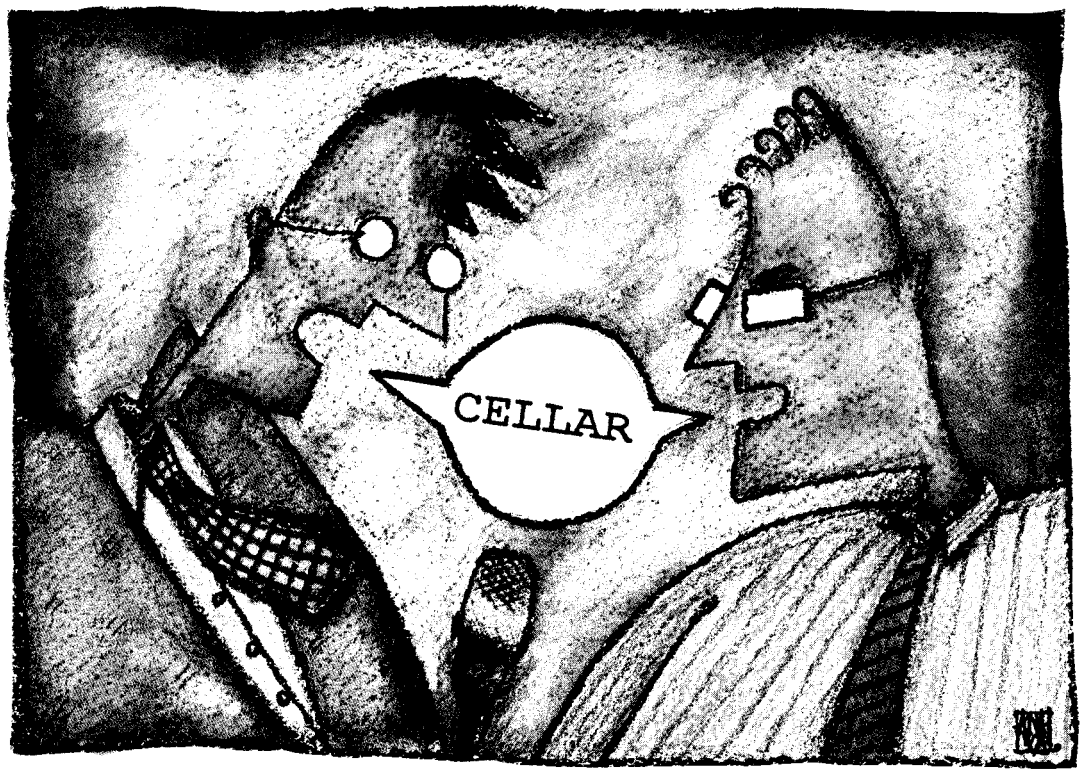
Extended Objects



Despite the name of this column, some installments have focused less on practice than on practical philosophy. The last two—"Hat Racks for Understanding" and "Cooperative Software"—advocated a set of ideals or goals that could guide the design of the next generation of applications software.

It is time to talk about these ideals in actual practice. Suppose someone set out to build software that facilitated understanding, that acted as a cooperative partner in the problem-solving process. What would that software look like? What internal structures would it have?

Programmers at the Academic Computing Department at the Summer Institute of Linguistics (SIL) in Dallas, Tex., have been building such software for the past three years. The result is called CELLAR: Computing Environment for Linguistic, Literary, and Anthropological Research (a name which reflects the project's original motivation more than its nature).

One of the key technologies in CELLAR is a set of fundamental extensions to the object-oriented approach to software development. Having decided what would be necessary for excellent applications support, the designers decided that these things should be implemented in the innermost parts of the system. This makes things like multiple views of information available throughout an application, at any level of granularity.

ILLUSTRATION: ANNA CICA

Project Vision

This work is driven by a vision for usable high-functionality software, a philosophy of training by immersion, and several key insights. Among them:

- Supportive applications require a conceptual model of the application domain. An environment for building supportive applications should have an excellent domain-modeling facility.
- Supportive applications require access to the structures and relationships contained in information, which are as valuable as its content.
- Looking at information in many arrangements and organizations leads to new understanding. Supportive applications allow users to define and use multiple views without programming.

The resulting architecture uses an extended object model, summarized in Figure 1. On top of this, the group is in the process of constructing a "Performance Shell," which is basically a performance support system for building performance support systems (or, to use another set of terms, a domain-oriented cooperative tool for building domain-oriented cooperative tools). I interviewed the two architects of CELLAR, Gary Simons and John Thomson. Simons is an experienced field linguist and accomplished computer scientist, acting as project leader and "keeper of the vision" for the project. Thomson is the lead designer and chief implementor. Most of this column is a heavily-edited transcript of our two-hour conversation about their design, the ideas behind it, their experience in building it, and their vision for the future.

Marc Rettig, Gary Simons and John Thomson

Views, the First Extension

MR: I've heard you say that CELLAR is a marriage of database and document technologies. Tell me about this.

JT: People tend to think in terms of creating documents, but often they really want to create a knowledge base. For instance, it is tempting to use a word processor to create a dictionary so that one can get the formatting just right, when one should really be

creating a database of highly structured information. We therefore wanted to build a system in which a knowledge base of structured and interrealated objects could be projected onto the display as though it were a document.

For this purpose we extended the object model to add the notion that every object (by virtue of its class definition) ought to encapsulate a set of methods for displaying itself. These methods we called views.

GS: Our descriptions of these views are declarative. Unlike a traditional approach that would demand you write a procedure to build a display, our views are defined by a declarative template that specifies how chunks of information are laid out in relation to each other and what formatting properties they have. It's really an integrative statement of what I want the object to be laid out like. All of the procedural aspects of building the interface are left to the underlying CELLAR facilities.

JT: Another interesting characteristic of views is that, when you build a view, you not only have a way of laying out the data, you implicitly have a way of editing it as well. The underlying implementation of CELLAR supports editing operations on information displayed in views, so that each view defines a structured editor for the object class.

MR: Where some systems separate everything into functional layers which talk through a protocol, you have embedded the interface directly in the data.

GS: That's right. It's not a system where you have all the data here and all the interface over there. We organize such functionality by objects, and what you usually wind up with is a hierarchy of objects and views. Big objects ask little objects for views of themselves.

MR: One of the advantages always touted for OOP is reuse of objects. It sounds like the business of nesting views within views is buying you some reuse.

JT: Yes. Many OOP systems don't make as clean a separation between the information and the way it is displayed. One typically designs an application by starting with a user interface and then working back to figure out what underlying objects it will take to support it. You're fairly lucky if the underlying objects wind up supporting more than that single application that you've made them for.

But our approach always begins by building a conceptual model of the objects in the problem domain, and then adding definitions for different views as they become needed. I think in practice we actually wind up with more reuse of the same object by having different ways of looking at it. A very typical object in our system might have half a dozen views which are each like little applications.

GS: And when you want to reuse an object, you can add another view without having to change any existing code.

Parts and Relationships

MR: I understand that another early innovation was the distinction between parts and relationships: like having two kinds of instance variables, from the programming point of view. What about the conceptual modeling point of view? Why did you do this?

JT: There is something very fundamental about the part-whole relationship. The whole—in our system it's called the "owner"—has a privileged relationship with its parts. For one thing, every part needs an owner. You can't have a subentry in a dictionary without having an entry that it's part of.

MR: So you are avoiding database anomalies through constraints on different kinds of attributes.

GS: Yes, and the privileged relationship that an owner has with its part is a key concept. In views we require that you can't edit something without its owner being a gatekeeper, because when you change a part you are also changing the whole. There are special integrity checks to prevent parts from being edited when they are not displayed in the context of the whole of which they are a part.

MR: And how do you achieve normalization in the database?

GS: This is where relationships come in. There is only one copy of any piece of information—modeled as an object—which is owned by a single larger object in the part-whole hierarchy (see Figure 2). Any object may, however, be related to many other objects in the knowledge base. We represent this with "relationships," or "references" as we also call them. Every instance variable (which we actually call an "attribute" of an object in our model) is declared to be representing how an object owns other objects that are part of it or refers to other objects that are related to it.

JT: Note that this distinction between parts and relationships has given us a very natural solution to the classic problem of "shallow" versus "deep" copying. There is a very clear designation about which things are part of the object and which aren't. When you copy a dictionary entry, for instance, you know you should copy all its subentries, but not copy all the objects they refer to as synonyms. It seems to have worked reliably.

Integrity

MR: Tell me more about the integrity information in these objects.

JT: This is in some ways an area in which we are ahead of the object model, and in another way an area in which we have limited it. In a pure object-oriented language like Smalltalk, the end user has no way to access instance variables directly; this must be done by writing methods or functions which access them. If someone writes the access method well, they can do whatever integrity checking they need. But they can, of course, write access methods that don't check integrity at all.

In our system we wanted a little more freedom to access the attributes, as we call them. We don't force people to write access methods. However, we do want to ensure integrity. Thus we have gone beyond the traditional object model in requiring that every class definition includes a definition of what it means to be a valid instance of the class.

MR: How do you accomplish this?

GS: Rather than having class definitions that just list their instance variables, our class definitions largely consist of attribute definitions. It's these attribute definitions that state,

in a declarative way, the intention of the knowledge engineer concerning what it means to be a valid instance of that attribute. These declarations give a "signature" listing classes which are valid as values of the attribute, tell whether the value must be single or can be a set or list, provide a default to use when the value is missing, specify a discipline for keeping values in an ordered list automatically sorted, and declare constraints on ranges of values or relations to values of other attributes. The latter may be mandatory, but more commonly they are "critics" which give suggestions or point out inconsistencies. They post messages to an "integrity agenda," which users can browse at leisure when they are in the mood to clean up the knowledge base.

MR: So, where most languages just declare variables, list the names and maybe declare type information, you have much more.

JT: Yes, and this approach of embodying integrity information in declarative attribute definitions, rather than writing it into access methods, buys us something else. It allows us to be somewhat lazy about integrity. We don't have to check integrity the moment you set a value; we simply post an entry on the integrity agenda that says a given value needs to be checked against the constraints on its attribute definition. We can allow incomplete or invalid states to persist until the information is available to rectify things. We have a mechanism through which you can absolutely forbid a condition if you want to. But for many application areas with a lot of uncertainty, like field linguistics, it is very helpful if you can violate a constraint temporarily.

Traditional Objects

Encapsulate	*To represent*
Instance variables	Internal state-parts and relationships
Methods or functions	**Behaviors**

Extended Objects

Encapsulate	*To represent*
Parts	objects owned by an object
Relationships	**objects related to an object**
Integrity constraints	how to know if an object is complete and valid
Views	**ways to display an object**
Tools	ways to directly manipulate an object
Queries	**questions this object can answer about itself and related objects**
Methods	ways this object can update the knowledge base
Parsers	**ways to represent this object as text**
Tasks	user activities involving this object

Figure 1 Traditional objects compared with extended objects.

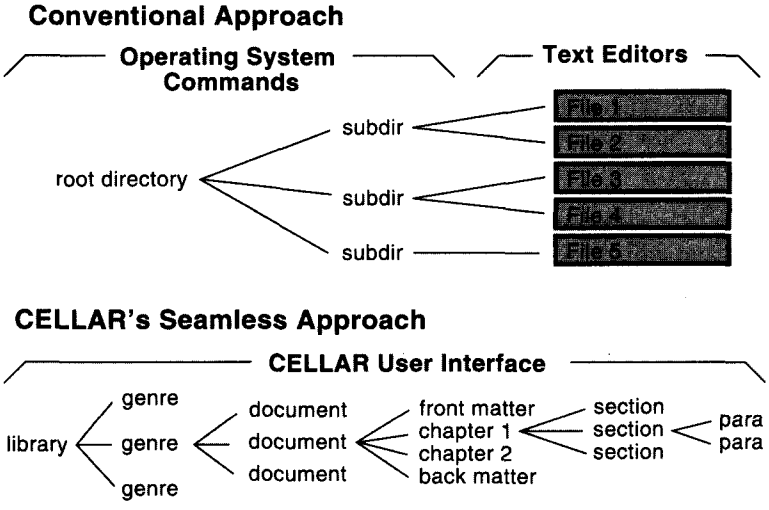


Figure 2 CELLAR presents a single "part-whole" hierarchy of objects to its users, each containing its own manipulation tools. There may be a complex web of stored relationships between any objects in the hierarchy.

Parsers

MR: Okay, I see in this list (Figure 1) that a class can also encapsulate a set of parsers? What does that mean?

GS: That's another way we have extended the object model. Another thing an object needs to know about itself is, "What would I look like if I were represented in an ASCII text file?" and "How would I convert that text representation into an instance of myself?" There are multiple parsers defined on most classes, each of which describes a different possible text file encoding.

MR: What are these parsers like?

GS: They, too, are declarative statements. They are essentially regular expressions which specify the pattern of how literal keywords and separator characters mark off the chunks of information. These elements of information may be simply strings or numbers at the lowest level, in which case they are recognized by built-in parsers on those classes, or they may be other objects from the problem domain, in which case they are recognized by calling a parser defined for that class. We get the same kind

of reuse of parsers for smaller objects within parsers for larger objects as we do for views within views.

JT: In fact, we have used this general-purpose parsing mechanism to build "source code" parsers for our modeling language. We represent "code"—the class and attribute descriptions, the queries, the parser definitions, everything executable—as a nested structure of objects. As a consequence of having parsers as a part of a class definition, we have been able to define a source code syntax and parser for all of our programming objects. Furthermore, it's quite possible for an end user to write a new programming syntax just by writing a new set of parsers for those kinds of objects.

GS: We have also implemented a structured editor for source code by defining views for all the programming classes that make the objects look like source code.

MR: So, much of the system is written in itself. There are conceptual models of the classes which have to do with programming, and the source code editors are written as view and parser definitions on these?

JT: Yes, and though we haven't done much of it yet, we hope that we can define another set of views that will create a visual representation of programs, showing the functions on streams of objects as a data flow machine.

Tools

MR: What are tools?

GS: They represent another kind of thing that an object should know about itself: all the possible ways that someone could manipulate it. Whereas a view gives a static projection of the information in an object, a tool provides a dynamic mechanism for manipulating the information in an object. A tool is a window that includes panes, buttons, menus, and other conventional controls. Defining a tool and the layout of the controls within it is very much like defining a view—the same declarative language is used. Launching a tool is just a matter of sending a message to an object: "Launch your *x* tool."

One tool that's implemented on most things is called "browser." There's a system browser, and folders have browsers. If you click on things like dictionaries, they will have specialized browsers as well.

JT: We're hoping that this will go a long way beyond the kind of thing that Smalltalk does by having inspectors that will let you see any part of any object. Only a few classes in the whole Smalltalk system have their own inspector. We're hoping it will be so easy to build customized browsers that almost every class will have one.

Tasks

MR: What about tasks? What are they?

GS: We are extending the object model to incorporate the end user's perspective on objects. In the traditional object-oriented approach, the methods encapsulated in objects define how programmers can access the objects. The same is true of the attributes, queries, views, and parsers in CELLAR. Tools give users direct access to manipulate objects, but are so broad in function that users may be at a loss to know how to use them to perform a specific task. For instance, one uses a word processing tool to perform tasks like "insert a new section," "join two paragraphs," or "change type size." When users sit down to a tool, they have these kinds of tasks in mind, and they get frustrated when they do not see how to relate these tasks to the controls of the tool.

What we have discovered is that this kind of knowledge, the knowledge about user tasks, really belongs in objects. The things a user might want to do with a dictionary need to be encapsulated with the definition of a dictionary. The things a user might want to do to the entries of a dictionary, to the subentries, and so on . . . each belongs with the class definition. So we are extending the model to describe not just how objects might talk with each other, but also how an end user might want to use these objects. We are working on adding "task definitions" to our model, which would describe tasks an end user might want to perform.

MR: How do you define a task?

GS: We have an abstract class called TaskDefinition which has five concrete subclasses.

- The simplest kind of task definition is for an "automatic task." It associates a taskname (which is a brief statement of what the user is trying to do) with a bit of program code that can automatically do it.
- Another is a "manual activity." When such a task definition is performed it simply displays an explanation to the user of how to do it.

The other three kinds of tasks build complex definitions by nesting subtasks.

- A "choice task" offers the user a choice among possible ways of doing something (which are in turn expressed as tasks).
- A "sequence task" presents a cue card which walks the user through the steps of a task, tracking progress along the way.
- A "process task" has a number of subtasks which are performed in any order and as often as needed to satisfy some given constraints.

MR: And how do users invoke these task definitions?

GS: All CELLAR tools present a help menu and that menu always includes an item called "possible tasks." When you choose that, you get a scrolling list of all the tasks defined for the class of the object which is currently selected in the screen display. When you pick one of these, the task definition executes itself and gives the user the help embodied within it.

JT: Another item that is always available in the help menu is "Explain Selection." It tells the class of the selected object and tells what attribute it is filling in the object that owns it. Two buttons on that dialog box, "Explain Object" and "Explain Attribute," go into the class or attribute definition and retrieve documentation that explains what they are.

MR: Some people might say, "I can see extending objects with interface stuff, and parsers and queries—those all feel like programming language issues. But end user tasks and documentation are not part of programming." But by putting tasks and documentation in the class definitions, you're saying that help and

We're saying that part of the process of programming is to anticipate the kinds of things users will need to know about objects and will want to do with them.

documentation is as much a fundamental part of an object as what its parts are.

GS: That's right. Presumably if a programmer is going to the trouble of implementing a class, he or she is doing so because someone needs to use it. We're saying that part of the process of programming is to anticipate the kinds of things users will need to know about the objects and will want to do with them, and that the program should then provide help to give this knowledge and assistance when they need it. It's related to Gerhard Fischer's work, integrating action and reflection, nonintrusive help systems.

Also, it's related to the performance support systems starting to show up in industry. CELLAR will nicely support a shell for building performance support systems, since it is designed to model all the information about a problem domain. In one object-oriented system, it integrates both sides of the equation for an effective application: what programmers need to know and do with a problem domain, and what users need to know and do. For the programmer this object-oriented system supports good programming methodology. For the user it supports both guided learning and independent exploration.

The Development Process

MR: So, developing an application with this feels different. Will a good programmer who is used to say, Smalltalk or C++ feel like this is foreign territory? You're giving them a ton of things to define.

GS: Well, step one in the development cycle is to build a conceptual model. That means identifying classes and attributes, and how they relate to one another. This is just garden variety object-oriented analysis, which should be the first step in the object-oriented programming project anyway. It has been our aim to make the analysis and design and implementation to be one and the same thing. We want to unify the languages people use to do these things. So, when you tell the CELLAR system about your object-oriented analysis, you have implemented the class. That's step one, and it should feel familiar to anyone who is used to preparing for object-oriented programming by doing object-oriented analysis.

Then you go beyond that to implement customized views, queries, parsers, and tools for the objects in the problem domain. As you get closer to the end you write documentation and task definitions to help people use your objects. When compared to "traditional" object-oriented programming, this may sound like adding a lot, but we were led to this approach by the realization that a complete program has all these ingredients anyway. By identifying these ingredients and building conceptual models and high-level sublanguages for them, we have been able to build a system that makes it easier to define these ingredients than it is with a general purpose language.

Effort and Results: The Cost of Leapfrogging

MR: How about performance and overhead? Some of the things you've described sound costly. Are your performance concerns directly related to extended objects, or are they just because you're trying to do something ambitious?

JT: It's hard to say. Some performance concerns are because Smalltalk is basically an interpreted language. There's a price for its power, both in time and memory space. Some concerns are because the programming language that we've developed doesn't lend itself very easily to compilation. That will make it hard to make big queries that execute quickly.

On the other hand, the basic view is interfaces don't cost us intolerably much. We can put complex views on the screen in a second or two on a fast PC.

GS: These are views which involve nested views of several hundred objects, each one of which is running some code.

Just the matter of automatically maintaining back pointers for all pointers in the system doubles the amount of storage. But our study shows us that the amount of disk you can buy for a fixed number of dollars has been doubling every year. In light of this, doubling the storage requirement does not seem like a very big price to pay for all the benefit it gives us. We've chosen to stretch the limits of the current generation. We're targeting that level of hardware that will give us acceptable performance by the time we ship the product.

JT: Even if it takes longer to get the performance up to an acceptable level—be pessimistic and say it's 1997 before people can afford a computer that will run this fast enough, and they could have afforded a suitable (slower) machine in 1995 that would be fast enough if we were able to write very carefully optimized code in C. The probability is very great that they still wouldn't be able to run it before 1997, because it would take us two extra years to build it! Or maybe not before the year 2000.

Vision

MR: What is your vision for the future of application development?

GS: I suspect that the role of knowledge engineer may eclipse the role of programmer. We're building a programming system which is also a knowledge engineering system. Step one in modeling a problem domain is knowledge engineering; step two might require some tricks of the kind that programmers do. But programmers will have to be knowledge engi-

We are building a system that will allow programmers in our organization to achieve the programming they need to do with as little effort as possible. We are building a performance support system for applications development.

neers to build a good foundation for their domain model.

JT: The next generation of tools should also make it as easy as possible to develop applications. If you can imagine it, you should be able to build it in a matter of days.

GS: We're building a very high-level tool. Each of these extensions takes it a level higher. You could do all these things with general methods or functions. But the programming system itself wouldn't have had, in advance, the conceptual model of what you were trying to do. Our extensions to the object model are really a conceptual model of what we think is programming. We are building a system that will allow programmers in our organization to achieve the programming they need to do with as little effort as possible. We are building a performance support system for application development.

JT: We've identified hundreds of tasks that people doing a field linguistic project need to accomplish which could usefully be aided with software. We can't afford to invest a person-year in building each of those programs.

For example, in the late-1980s we spent a couple of years building interlinear text analysis tools in C. Recently, using CELLAR, one of our staff took only two weeks to build a multiple-pane, multiple-language browser of several versions of the Bible, including interlinear annotation and hooks to Greek and Hebrew lexicons. The interlinear display retrieved glosses from these lexicons, and full lexica entries appeared for words selected by users.

Closing Remarks

The last part of the interview where Thomson and Simons expressed their enthusiasm for the potential of their tool has a slight odor of hype about it considering they aren't even finished building the thing yet. On the other hand, even if CELLAR isn't the project that brings a new level of productivity to software developers, someone is going to bring it to us. And chances are it will share some of the qualities displayed in CELLAR.

I'm publishing this work in "Practical Programmer" for several reasons:

- Extended objects are a nice example of "coloring outside the lines"—finding creative solutions to difficult problems. And I think they hold promise as a useful way to build software.
- The project is a good example of leapfrogging. That is, the designers have chosen not to evolve their current generation of tools, shooting instead for something that will take full advantage of the increased power of the computers just now becoming affordable to their typical users.
- This design illustrates the kind of system support necessary to build the kind of cooperative, supportive applications that I hope will appear in the near future. We can use the power of next-generation hardware by adding new features, but that will be counter-productive if people can't tell which feature to use, or how to use it. CELLAR's "task definitions" and structured "documentation objects" stored class-by-class are a start in the right direction. And the promise of "performance support for application development" is quite exciting. ◻

Further Reading

Coombs, J.H., Renear, A.H. and DeRose, S.J. Markup systems and the future of scholarly text processing. *Commun. ACM 30*, 11(Nov. 1987), 933–947.

Fischer, G. Domain-oriented design environments. In *Proceedings of the Seventh Knowledge-based Software Engineering Conference*, IEEE Computer Society Press, (1992), pp. 204–213.

Fischer, G., et al. Supporting indirect collaborative design with integrated knowledge-based design environments. In *Human-Computer Interaction*, Vol. 7, Lawrence Erlbaum, Hillsdale, NJ, (1992), pp. 281–314.

Fischer, G. and Reeves, B. Beyond intelligent interfaces: Exploring, analyzing, and creating success models of cooperative problem solving. *J. Appl. Intel* 1 (1992), 311–332.

Fischer, G., Lemke, A., Mastaglio, T. and Morch, A. I. The role of critiquing in cooperative problem solving. *ACM Trans. Info. Syst. 9*, 2 (Apr. 1991), 123–151.

Gery, G.J. *Electronic Performance Support Systems*. Weingarten Press, Boston, 1991. (Weingarten has been acquired by Ziff, and is now part of the Ziff Institute.)

Rettig M. Cooperative software. *Commun. ACM 36*, 4 (Apr. 1993), 23–28. (Discusses the work of Gerhard Fischer and Performance Support Systems.)

Rettig, M. A succotash of projections and insights. *Commun. ACM 35*, 5 (May 1992), 25–30. (Describes the study of increasing hardware performance mentioned in this column.)

***Marc Rettig** is a member of the technical staff at the Summer Institute of Linguistics, and a freelance writer.*