Beyond Models and Metaphors: Visual Formalisms in User Interface Design

Bonnie A. Nardi, Craig L. Zarmer
Software and Systems Laboratory
HPL-90-149
September, 1990

The user interface has both syntactic functions - supplying commands and arguments to programs - and semantic functions - visually presenting application semantics and supporting problem solving cognition. In this paper we argue that though both functions are important, it is time to devote more resources to the problems of the semantic interface. Complex problem solving activities, e.g. for design and analysis tasks, benefit from clear visualizations of application semantics in the user interface. Designing the semantic interface requires computational building blocks capable of representing and visually presenting application semantics in a clear, precise way. We argue that neither mental models nor metaphors provide a basis for designing and implementing such building blocks, but that visual formalisms do. We compare the benefits of mental models, metaphors and visual formalisms as the basis for designing the user interface, with particular attention to the practical solutions each provides to application developers. We describe our implementation of a visual formalism to show the potential for visual formalisms to serve as reusable computational structures that support the development of semantically rich applications.
1 Introduction

Both mental models and metaphors have been proposed as a basis for user interface design. Some investigators argue that user interfaces should "closely match the way a user thinks of a task," (Hollan, Hutchins, McCandless, Rosenstein & Weitzman, 1987), and that interfaces should reflect people's mental models (Hollan, Hutchins, & Weitzman, 1984; Norman & Hutchins, 1988). Others believe that familiar, everyday metaphors such as desktops or Rolodexes should be the starting point for interface design, since users can interpret the interface based on their prior knowledge of the source of the metaphor (Apple Human Interface Guidelines, 1987; Carroll, Mack & Kellogg, 1988; Blumenthal, 1990). In this paper we argue that neither mental models nor metaphors provide a basis for the user interfaces for many of the complex scientific, engineering and business applications users need. Basing user interfaces on mental models is, we will argue, impractical, and provides a confusing design guideline for developers. Metaphors are useful for some purposes, but inadequate to serve as the basis for interfaces of any complexity which must express their own rich semantics.

Instead, we propose visual formalisms as a basis for interface design. Visual formalisms are diagrammatic notations with well-defined semantics for expressing relations. They are based on simple visual notations such as tables, graphs, plots, panels and maps - objects that contain their own semantics and do not metaphorically recreate the semantics of some other domain. Versions of such visual notations that define a precise semantics become truly formal. As we will discuss, visual formalisms are simple but expressive, compact but rich in information. They can form the basis of user interfaces for many kinds of programs because their relational semantics are broadly applicable across many domains. A visual formalism can be specialized to meet the needs of many specific applications; for example, specializations of tables include spreadsheets, decision tables, calendars, schedules, hospital flow sheets and data displays for scientific, engineering or business data.

Visual notations are commonly used for display purposes, but it is less common for users to be able to manipulate their components - to be able to ask about the values behind a point on a plot, for example, or to expand a region on a map to show more detail. It is even less common for these displays and their components to possess any semantic information about their relationships to other displays or components - say, constraints between specific values, or the mapping from one notation to another. Computer-based versions of visual formalisms can provide these capabilities through sophisticated visual/semantic mechanisms, utilizing knowledge-based representations and interactive editing and browsing techniques such as filtering and fish-eye views. Visual formalisms, unlike mental models and metaphors, have the potential to become reusable computational structures, and thus provide practical help to software developers. A library of specializable visual formalism objects can fill a middle ground between the expressivity of general programming languages and the semantics of specific applications, giving application developers higher-level visual/semantic objects with which to develop specialized applications.

In the first part of this paper, mental models, metaphors and visual formalisms are compared in light of our ethnographic study of spreadsheet users in which modeling tasks were studied. Mental models, metaphors and visual formalisms are evaluated with respect to their potential to provide highly expressive user interfaces that capture rich application semantics, and their potential to provide practical help to software developers through reusable semantic objects that support application development at a higher level than general programming languages.
There is a vast literature on mental models in psychology, philosophy and other disciplines which cannot be summarized here (see de Kleer & Brown, 1981, 1983; Kahneman & Tversky, 1982; Johnson-Laird, 1983; Gentner & Stevens, 1983; Reisberg, 1987; Rips, 1986; Sanderson 1989), and little agreement on just exactly what “mental models” are, or if they even exist. The literature on metaphors is also immense (Ortony, 1979; Lakoff & Johnson, 1980; Tourangeau & Sternberg, 1982; Carroll & Thomas, 1982; Carroll & Mack, 1985; Holland & Quinn, 1987), and again cannot be reviewed here. In this paper we examine the concepts of mental models and metaphors with respect to human-computer interaction research. In particular, we consider the role of mental models and metaphors in the design of user interfaces for software that supports complex problem solving tasks such as design and analysis. We make a distinction between the “syntactic” user interface through which users issue instructions to the computer, and the higher-level “semantic” user interface that visually presents application semantics and explicitly supports problem solving within the application. We will be fairly critical of mental models and metaphors as a basis for designing the user interface, and it is important that readers understand that our comments apply only to the class of tasks we specifically address – complex problem solving tasks for which a rich, precise semantic interface is required.

In the last part of the paper we briefly describe the implementation of a table visual formalism. The table visual formalism is part of the efforts of our project team to build a library of visual formalisms for application developers.

2 Muddles in the Models

2.1 What’s in a mental model?

In human-computer interaction research, the notion of “mental models” has come to be a very general catch-phrase for anything having to do with end users’ knowledge of an application (van der Veer, 1990). There is a feeling that if we could “capture” mental models, then we could build good interfaces because they would “have characteristics similar to ... people’s mental models” (Hollan, et al. 1987). Hollan, et al. (1987) stated that graphical user interfaces are important because they can “depict ... models of the world which are similar to the mental models ... people seem to use to reason about the world.” The authors commented that “people are especially good at ... constructing mental models.”

But many psychologists and others who have studied mental models are much less convinced of the alleged benefits of mental models and of our ability to use them for reasoning or other complex cognition. Some of the reason for this doubt can be intuitively grasped if we first consider mental images – simpler than mental models, and a subjectively compelling mental construct. Researchers who have investigated mental images have been struck by how incomplete and inflexible they are, despite subjects’ initial reports of their clarity and vividness. For example, Reisberg (1987) discussed the “imaged tiger” experiment in which subjects are asked to imagine a tiger. No one seems to have trouble mentally conjuring up a tiger. But when the subjects are then asked to count the tiger’s stripes, they “typically register surprise at [their] inability to do this.” Another exercise is to ask subjects to image the word “pumpkin,” and then ask them to say aloud the letters of the word in reverse order (n-i-k-p ... ). Again, subjects are surprised at the difficulty of doing this, “even when the imaged word was subjectively quite clear and vivid” (Reisberg, 1987).
Reisberg observed, "Apparently then, the image is appreciably less clear than the corresponding picture (from which the letters can easily be read), and, importantly, appreciably less complete than the subjects themselves had believed."

If single, uncomplicated mental images are so incomplete and inflexible, then what of "mental models" which, as models we expect to provide explicit, comprehensive and valid representations of the elements and relations which comprise them? (See Carroll & Mack, 1985 on models.) Norman (1983) characterized mental models as incomplete, unstable, severely limited in manipulability (e.g. being able to reverse letter order in an imaged word), and lacking in firm boundaries (e.g. one device is mixed up with another).¹

Rips (1986) provided a thorough discussion of the many problems with the concept of mental models, considering the literature of psychology, philosophy and artificial intelligence. He pointed out that there are two flavors of mental modelers: those who see mental models as internal analogues of real physical phenomenon that can be mentally manipulated to reason about the world (as in Hollan et al., 1987), and those who use the term "mental models" more generally to mean any mental representation that embodies domain knowledge. Rips argued that the second group of "figurative" mental modelers is guilty primarily of loose talk, and that from a representation standpoint their notions of mental models can be captured by propositions or production rules (see, for example, Young, 1981). Rips advocated sticking to propositions and production rules until something better comes along, rather than muddying the waters with imprecise terminology.

On the other hand, the "literal" mental modelers, as Rips calls them, are more troublesome. They believe that mental models can be used to "run" mental simulations or "thought experiments," and that the physical components of some real-world system, for example, a circuit, are mentally represented as e.g., resistors, capacitors, etc. The mental model of the circuit can be mentally manipulated to reason about the circuit's behavior. A mental model is thus representationally quite different than a proposition or production rule.

Rips provides a careful evaluation of the literature from psychology and AI that casts grave doubt on the notion of doing any but the simplest "thought experiments" inside the head. For example, in an experiment that Rips conducted with Dedre Gentner, university students who had high school training in physics and chemistry (but no college-level courses) were asked questions about the relationships between air temperature, water temperature, air pressure, evaporation rate and relative humidity for an imagined pan of water sitting in a closed room. The subjects were presented with pairs of the variables and asked to decide whether a change in the first variable would cause a change in the second. The data were examined for intransitivities (i.e. if a subject said that a change in variable x causes a change in variable y, and a change in variable y causes a change in variable z, but a change in variable x does not cause a change in variable z) on the theory that if subjects had "coherent running models" of the system, their answers would contain few intransitivities.

The subjects' answers were at chance levels: they produced the same number of intransitivities as if they had flipped a coin to answer the questions. Rips concluded that mental

¹ For a poet's take on the problem, consider a passage from The Inland Island by Josephine W. Johnson (1969) in which she describes the toothwort, "a modest flower related to the turnip." Johnson writes of the toothwort, "Like most wild flowers, if picked and brought inside, it droops and has a weedy negligible look, reminiscent of one's own thoughts, which seem fresh, honest, sparkling, rare, when rooted still in the cool brain cave, but in the open air, picked and presented, tend to appear dusty and weak, irrelevant to the human condition of flesh, brass, and blood."
simulation does not work even in the simple five-variable system he and Gentner tested, and that "the evidence is . . . consistent with the view that subjects were relying on simple, error-prone rules of thumb that could easily be expressed propositionally." Forbus (1983) came to a similar conclusion from his computer simulations of "qualitative physics" (solving physics problems with qualitative information only), noting that "[in mental simulation] . . . the burden of building a complete description of possible states is too onerous outside very small domains, and is too restrictive a style to capture all the ways people use qualitative physical knowledge." (See also de Kleer & Brown, 1981, 1983; Kahneman & Tversky, 1982.)

We agree with Rips (and those he cites) that domain knowledge is extremely important in problem solving, but that it is best thought of as propositions or productions. Kintsch (1988) points out that fixed knowledge structures such as frames, scripts and schemata are too inflexible to adapt to the ever-changing contexts of everyday life. Kintsch posits a "minimally organized knowledge system . . . in which structure is not prestored, but generated in the context of the task for which it is needed." In his scheme, propositions and rules work together in "constructive processes" to handle the specific and highly unpredictable situations people routinely deal with. Kintsch analyzed the tasks of word identification in discourse and solving arithmetic word problems to illustrate how very general cognitive strategies are used to generate a large number of possible solutions that are then "weedied out" in light of the particulars of the situation, providing flexibility and context sensitivity.

As Rips noted, nothing is gained by casting propositional domain knowledge as "mental models." The concept of mental model does not suggest the kind of flexible "bottom-up" processing that Kintsch argues for, and that seems to account so well for our ability to deal effectively with change. Uncritical use of the notion of mental models is especially likely to lead to confusion in the context of interface design. What it would mean for a user interface — for graphical user interfaces in particular — to "match a mental model" would be to mirror a "literal" mental model capable of running mental simulations (exactly as Hollan and his colleagues proposed). It is precisely this kind of model that seems not to exist, or perhaps to exist only for simple problems — which we would not use computers to solve anyway.

Given all the problems that psychologists and other investigators have in pinning down just what mental models are (or deciding if they really exist, as Rips wonders), and given the instability and incompleteness that characterize them, we believe that it is counterproductive to exhort software designers to try to describe people's mental models and then build user interfaces that mimic them. This approach gives no tools to designers, but asks them to go out and do what professional cognitive psychologists have been unable to do — convincingly describe meaningful mental models of any but the simplest phenomena.

2.2 User interface: Passive or active?

A subtle effect of the focus on looking inside the head for internal representations to be somehow made manifest in the user interface is that the user interface is seen as a passive syntactic mechanism whose purpose is to transfer a user's commands to the processor. The user is the "thinker" and the user interface is an inert membrane through which information passes. The user interface does not in itself take an active role in the problem solving process (as the processor and the user do). The level of discourse about the user interface gets stuck at mouse clicks and keystrokes (those actions which transmit information to
the processor), instead of moving ahead to a consideration of the tremendous potential of
the user interface to visually present domain semantics that are too complex to be held in
mental models, and that can only be made cognitively accessible in external models that
provide a clear visualization of an application's semantics.

Let's look in more detail at the implications of the passive view of the user interface.
Hutchins, Hollan and Norman (1986) argued that users' mental representations of tasks
should be "matched" by the user interface of the programs they use to accomplish those
tasks. The interface should "shorten the distance" between a user's "thoughts" about a
task and the system actions necessary to accomplish the task. As Hutchins et al. (1986)
put it:

A short distance means that the translation is simple and straightforward, that
thoughts are readily translated into the physical actions required by the system

According to this viewpoint, users' "thoughts" about the problems they are trying to solve
must be "translated" by a program into "physical actions" such as pressing buttons or
typing characters. The "translator" notion implies that the knowledge of how to do a task
is strictly in the head, and can flow smoothly out of the head into the waiting arms, as it
were, of the gestures provided by the system.

To see the interface as a mechanism for translating thoughts is to completely miss the in-
teraction between the user and the user interface, and the way in which the user interface
itself can stimulate and initiate cognitive activity. Like other cognitive artifacts (Chan-
drasekaran, 1981; Holland & Valsiner, 1988; Norman, 1987; Norman & Hutchins, 1988), a
good user interface helps to organize and direct cognition - it is not a passive receptacle
for thoughts emanating from an internal model, but plays an active role in the problem
solving process.

To take a simple example of the active role of a cognitive artifact in problem solving, let's
look at the use of the bracket, e.g.,

256)16384

in long division. Imagine doing long division problems without the familiar bracket which
puts divisor, dividend, etc. in their proper places, and helps us keep track of each compu-
tational step. The form of the bracket, and the emerging physical representation of the
long division problem based around the bracket (our written version of the problem), are
important - they provide an effective structure for organizing a rather complex calculation.
The bracket is not a "translator" at all, but a means of organizing the data and manipu-
lations on the data. When we use it, it guides us - we expect it not to yield "translations"
of our thoughts, but to stimulate those thoughts, to help us remember how to do long
division. The process is the opposite of translation: in translation a thought must precede
any translation of it, whereas in using a cognitive artifact such as the bracket, the thought
follows the use of the artifact and is brought forth through interaction with it. The bracket
further aids problem solving in showing relationships among data: there are special places
for divisor, dividend, intermediate calculations, quotient, remainder. We are spared the
need to create a "mental model" of all of this - we have just written it down and can look
at it, save it for future use, and show it to others.
The idea that the user interface is merely a way station on the road to the processor reduces the interface to its syntactic functions only, and ignores the potential of an interface to organize and direct cognition in the manner of the bracket in long division. We now turn to a discussion of the spreadsheet interface to see how it helps users to organize and structure their applications, and how it effectively displays problem semantics in a way that directly and actively aids problem solving.

3 How People Create Models: A Study of Spreadsheet Users

We have conducted an ethnographic study of spreadsheet users to understand spreadsheets as computational devices - in particular to learn how spreadsheet users take the basic structure of a spreadsheet and mold it into an application that addresses some specific need (Nardi & Miller, 1990a, 1990b). In this paper we use examples from the spreadsheet study to exemplify real problem solving activity using computers. This material illustrates the importance of recognizing that users do not work from full-blown “mental models,” but instead incrementally develop external, physical models of their problems. The external models focus cognition and problem-solving activity.

For the field research we interviewed and tape recorded conversations with spreadsheet users in their offices and homes. We chose an ethnographic approach - studying a small number of people in some depth - because we are interested in how users themselves structure the problem solving process - a topic that by its very nature cannot be studied under the controlled conditions of the laboratory. Study participants were found through an informal process of referral. We told them that we are interested in software for users with little formal programming education and that we wanted to talk to people actively using spreadsheets. The interviews were conversational in style, intended to capture users' experiences in their own words. A fixed set of open-ended questions was asked of each user, though the questions were asked as they arose naturally in the context of the conversation, not in a fixed order. During the interview sessions we viewed users' spreadsheets on-line, and sometimes in paper form, and discussed the uses and construction of the spreadsheets. The material in this paper is based on about 350 pages of transcribed interviews with 11 users. We have also examined and worked with several different spreadsheet products including VisiCalc (the original personal computer spreadsheet), Lotus 1-2-3 and Microsoft Excel.3

Study participants were college-educated people employed in diverse companies, from small start-ups to large corporations of several thousand employees. Participants had varying degrees of computer experience ranging from someone who only recently learned to use a computer to professional programmers. Most were non-programmers with 3-5 years experience with spreadsheets. Names of informants as used in this paper are fictitious.

---

2 The interviews were conducted by the first author. We use the plural “we” here for expository ease.

3 Lotus and 1-2-3 are registered trademarks of Lotus Development Corporation. Microsoft and Excel are registered trademarks of Microsoft Corporation.
3.1 More muddles: Reconstructive surgery and false endings in spreadsheet development

In the study we found that spreadsheet users are very aware of the fact that their initial problem formulations are likely to be fuzzy, incomplete and badly structured. They like spreadsheet software because it helps them to work through these difficulties. Users often begin building a model with incomplete knowledge of the parameters needed in the model and insufficient understanding of the relations between model variables. They do have domain knowledge, but it may be very incomplete, and is not articulated into anything resembling a model -- nor do users expect that it would be. Users look to spreadsheets as a problem-solving device, a means by which they can organize their data, establish model relations and develop a deeper understanding of their problems.

The spreadsheet interface offers a structured visual format that helps the user to shape a representation of the problem. The rows and columns of the spreadsheet’s tabular framework provide a simple, effective means of organizing and presenting data. The formula language is a restricted though useful means of expressing relations between cells in the spreadsheet (Nardi & Miller, 1990b). Basically it offers a set of simple techniques based on arithmetic operations that users know how to use to allow them to relate quantities with the spreadsheet. The formula language allows users to express numerical relations at a high level, without having to string together lower level machine primitives, and in that sense it “matches” users’ knowledge of arithmetic. The spreadsheet thus offers good access to techniques people can use to create an explicit model of a particular application – but it does not in any sense match a “mental model” of an application; such models, as the following examples illustrate, do not exist. As the user enters data and formulas into the spreadsheet, the representation of the problem emerges through the medium of the spreadsheet. The representation is not “in the user’s head,” nor was it ever in the head, but becomes an artifact created by the user’s interactions with the program.

The spreadsheet itself is simply a table with a formula language – hardly a representation of a specific problem such as monitoring departmental expenses or tracking product sales. Spreadsheets have been extremely popular with personal computer users because they supply an orienting framework that permits users to work through and model their problems, not because they match pre-existing mental models.

Let’s look at some specific examples of spreadsheet users working through problems.

Ray manages a finance department for a large corporation. He was trying to figure out how to allocate investments across the different departments and organizations within departments that he serves. His goal was clear – to invest where it made the most sense. How to set up a model to achieve that goal was much less clear. Ray did not have a model of a good investment scheme; in fact he was admittedly befuddled about how to think about allocating the investments. To develop a model Ray created and compared several different spreadsheets with different schemes. In the following exchange we are looking at different versions of the investment allocation spreadsheet. Ray explains:

Ray: ... You know, we have so many different kinds of [departments], organizations in each [department], I was really trying to come up with a way to partition them, such that we locate each one in a different way in terms of needs and in terms of growth. And support areas typically get squeezed a little more, they work differently than we do, from the [departments]. The [departments]
is really where the investments should be going. So we want to look at them a little differently ... So a lot of this [the different spreadsheets we are looking at] was where I was playing with different ways of looking at it.

Ray finds that not only does he need to create and compare different versions of a spreadsheet, but that an individual spreadsheet may need, as Ray put it, "reconstructive surgery," that is, re-structuring and re-designing:

Ray: I get the most use out of spreadsheets by being able to do a lot of reconstructive surgery on [them]. You know, I can really move things around, change the whole look and feel and objective of a spreadsheet very quickly ... I may not even, in some cases, ... know the final form, look and feel of the spreadsheet that I want. I'll just start getting the data in, and then I'll start ... playing with moving rows and columns around and doing things until I see, until I get what I want (emphasis added).

Ray noted that the process of refining the spreadsheet is visual - he literally sees the correct version of the spreadsheet. Ray is not working toward a better "mental model," but is creating an artifact that embodies the ideas he is trying to capture and with which he interacts visually.

Often spreadsheet users do not even know the parameters of the problem they are trying to solve, much less have explicit representations for them in a mental model. Users find out about new aspects of a problem in the process of trying to solve it. Participants in the spreadsheet study often noted that spreadsheets are easy to change and there is no penalty for adding new things as they come to light. Rows and columns are easily added to a model, and formulas can be adjusted as new facets of a problem emerge.

Jeremy, another user in the study, is an accountant. He described how he learned to use spreadsheets. A job assignment required developing business plans for joint ventures with foreign companies. Large, complex spreadsheet models were part of the plans. Although programmers were available to help, Jeremy discovered that not only was it easier to be in control of spreadsheet development himself, but that he could use the spreadsheet to work through the problem, in particular to identify the variables of interest and to make sure that the model was complete.

Jeremy described this process:

Jeremy: We had to have rather large complex spreadsheets [for the business plans] where you had lots of variables. And I found it easier to develop that myself than to go to somebody and say here's what I want, here's what I want, here's what I want. And that's what really got me going on [spreadsheets] ... Interviewer: Why was it easier for you to do this yourself than to specify it for a programmer?

Jeremy: ... I think it was quicker and easier because I felt that I was learning as I went, as I was developing the spreadsheets, I was learning about all the variables that I needed to think about. It was [as] much a prop for myself as [a way of] ... getting the outcome.

...
And there were a lot of false endings, I should say, not false starts. I'd get to the end and think, "I'm done," and I'd look at it and I'd say, "No, I'm not, because I've forgotten about one thing or the other."

The visibility of the emerging model is very important in the problem solving process. As can be seen from Ray and Jeremy's work with spreadsheets, users critique their models by visually inspecting them. As Jeremy observed in describing his round of "false endings," he evaluated the completeness of his model by looking at it: "I'd look at it and I'd say, 'No, I'm not [done], because I've forgotten about one thing or the other.'" Problem solving occurs in the interaction between the user and a physical artifact which visually represents a problem, not in a reified space inhabited by "mental models."

Jennifer, another user in the study, is an accountant at a telecommunications firm. Part of her work involves developing spreadsheet models used by the Chief Financial Officer (CFO) of the firm. She described how her initial spreadsheet models are likely to be incomplete because the CFO does not spell out in advance exactly what he needs, but looks at an initial version of the model that she prepares for him and then decides what he really wants:

Jennifer: Oh, this [spreadsheet] is what I gave to the CFO at first, just comparing Q2 [Quarter Two] year-to-date budget to Q2 year-to-date actuals. And he said, "Well, for the board meeting I want [some other things]." Every time you do this he wants it differently. So I can't anticipate it. I just give him what I think [he wants] and then he says, "Ah, no, well, I want to have projected Q3 and projected Q4, and then total projected, and then have the whole year's plan on there."

It is difficult to imagine capturing the mental models of the CFO; in fact he himself does not organize his thoughts until he has the spreadsheet artifact to look at, react to, and critique. This example also shows the importance of an external model—in place of a mental model—for effective computer-supported cooperative work. Jennifer and the CFO communicate with one another directly via a copy of the spreadsheet itself (see Nardi & Miller, 1990a).

3.2 Mental models, external representations and user interfaces

Turning back to the question of the user interface, we must ask: If people are "bootstrapping" their way to understanding and solving problems, and in fact do not have a priori mental models of the problems they are trying to solve, then how can the user interface "match" mental models? The answer is, of course, it cannot. The job of the interface, in our opinion, is to suggest visual formats that effectively model users' problems, and to offer tools with which to build the models that utilize these visual formats, just as we have seen with spreadsheets. This means that users work not with mental representations, but with physical, external representations. Programs such as HyperCard, CAD systems, and spreadsheets provide users with the means to build, within the confines of the format supported by the program, their own interfaces to their own specific problems. These programs make no attempt to "match a mental model"; instead they provide facilities

---

HyperCard is a registered trademark of Apple Computer.
for constructing interfaces (and other application functionality) that capture the semantics of very broad classes of applications: design for CAD; creating simple databases for HyperCard; and modeling numerical relationships for spreadsheets.

Focusing on mental models deflects attention from the more important pursuit of understanding users' tasks in terms of (1) high level task goals and (2) the data structures and manipulations that must be provided to model the goals. Users need software that gives good access to data structures and operations so that they can then build their own models. In many cases we believe these data structures and operations will be useful only within limited domains – just as CAD systems, spreadsheets and HyperCard enable users to represent some problems but not others (Nardi & Miller, 1990b). This means that software designers must come to understand appropriate building blocks for different kinds of applications across many domains.

What kinds of models should we encourage users to build? As with the user programming systems mentioned above, and with the software being developed for scientific visualization (Warner, 1990; Williams, Smith & Pecelli, 1990) an important component of the models is an explicit visual representation of the underlying data and their relations. Computers give us the ability to create these visualizations to take advantage of what Reisberg (1987) calls “perceptual knowledge” – that is, knowledge that can be accessed only through interaction with external representations. As Reisberg says, “external representations... give us access to knowledge and skills which are otherwise unavailable to us.” He summarized several experiments from the literature in which subjects could not make correct judgments when asked to recall or predict something, but could recognize the correct answer. For example, in a series of experiments, college students were unable to correctly predict trajectories of moving objects (even students with training in physics), but when shown videotapes of moving objects, they could distinguish anomalous (simulated) trajectories from correct trajectories. In another experiment, subjects studied a word-list. Twenty-four hours later recall of the words was at near-chance levels. However, when subjects were shown a group of words that contained both those in the word list and control words, they were twice as likely to identify the previously viewed words compared to the control words. Reisberg observed that when we cannot remember how to spell a word, writing out a couple possible spellings is often all that is necessary – we can then pick the one that “looks right.” In short, we have access to certain kinds of knowledge only when we see it.

Of course interface designers have already taken advantage of perceptual knowledge in providing menus and pushbuttons where we once had typed commands. Menus and pushbuttons are quite useful, but what is of interest here is the more complex cognitive activity that should be supported by good visualizations. We want to focus attention on the way that computers can support cognitive activities that are part of the problem-solving associated with the application itself – activities such as perceiving trends, seeing patterns, finding individual data values in large datasets, making comparisons, testing for accuracy and completeness. For these activities the external stimulus of the physical representation is needed to organize and present the information in such a way that the viewer can access perceptual knowledge, which, as Reisberg pointed out, is quite literally unavailable to us without an appropriate external representation. In the spreadsheet study we have seen how Ray and Jeremy used the visualization of their models embodied in the spreadsheet table to test for the accuracy and completeness of their models. Both users explicitly remarked on the visual nature of the evaluation process, Ray noting that he re-structured his model “until I see... what I want” and Jeremy stating that completeness checks were done as he “looked at” the model to see what he had forgotten. It is this kind of cognitive
activity for which good visualizations – and good user interfaces – are needed.

Some investigators argue that over time, we internalize the information in external representations such that we no longer need the external representation to do the task (Hutchins, 1987). Hutchins uses the example of checklists – through repetitive use, we can learn to perform the actions in the list without the checklist itself. This is certainly true, and the internalization process is very important. However, we want to emphasize that mental representations will never replace physical ones for many of the tasks we do with computers – for example, interacting with databases; using spreadsheets; analyzing scientific and engineering data in graphs, plots, tables, maps and other forms; browsing directory trees; illustrating difficult concepts through animation. The data change too rapidly and are too vast for us to fully internalize. Furthermore, the artifacts that we use in performing tasks with spreadsheets, databases and so forth, are an important component of cooperative work practices; in fact such artifacts are essential for the communication of complex data between co-workers (Nardi & Miller, 1990a).

In summary, we are arguing that as user interface designers we should shift attention away from studying mental models and toward understanding external representations – in particular toward understanding the kinds of external representations that aid people in performing complex cognitive functions. After all, this is our opportunity with computers – to explicitly represent and manipulate complex data in ways that were literally impossible ten or twenty years ago. We believe that in our zeal to “get inside the head,” we are missing an important way in which people actually interact with the world to solve problems – i.e. with external representations. Everyday life is rich in these representations: sketches, drawings, maps, diagrams, sewing patterns, control panels, longhand arithmetic, 3-d models (of everything from molecules to solar systems), a wide range of notations (from predicate calculus to Labanotation for encoding dance movements), and calculating devices such as the abacus and nomogram (Norman & Hutchins, 1988), to name a few. We argue for turning our attention outward, not inward, to achieve an understanding of how cognition is stimulated and organized by external representations.

4 Metaphors

The idea of using metaphors in the user interface is laudable insofar as it turns our attention away from inchoate mental models and toward shared representations. However, metaphors suffer from numerous problems that make them unsuitable for expressing rich application semantics, and inappropriate for the reusable computational structures we seek.

5 An article in the San Jose Mercury News, May 1, 1990 (reprinted from the Boston Globe), headlined “Toy store yields cure for Hubble,” reported that a problem with the antenna boom of the telescope was solved in part by a model of the boom and swiveling satellite dish “assembled by an engineer from a piece of lamp cord and some components picked up at a toy store” (identified earlier in the article as Tinker Toys). The article stated, “The model was a big help in visualizing the problem, said Jean Olivier, Hubble deputy project manager.” Another visualization story is told in The Double Helix by James Watson. Throughout the book, Watson emphasized the importance of the metal models (metal, not mental) constructed in his laboratory to visualize the structure of DNA molecules. For example, he explained that in solving a key problem in the molecule’s structure, “...the essential trick...was to ask which atoms like to sit next to each other...the main working tools [for understanding the position of the atoms] were a set of molecular models superficially resembling the toys of preschool children.”
4.1 Appeasing the beast: Metaphors and the syntactic user interface

Before we look at the weaknesses of metaphors, we should ask why they play such a prominent role in the lore of user interface design. The main intuition behind the advocacy of metaphors in the user interface is that "a user can draw upon his knowledge about [a] familiar situation in order to reason about the workings of the ... new system" (Halasz and Moran, 1981). Many investigators (Rumelhart & Norman, 1981; Carroll and Thomas, 1982; Smith, Iby, Kimball, Verplank & Harslem, 1982; Carroll and Mack 1985; Blumenthal, 1990) have declared that since people often employ metaphors when first learning an unfamiliar task or domain, "designers of [computer] systems should anticipate and support likely metaphorical constructions to increase the ease of learning and using the system" (Carroll and Thomas, 1982). Right up front, the Apple Human Interface Guidelines manual says, "Use concrete metaphors and make them plain, so that users have a set of expectations to apply to computer environments" (p. 3).

Metaphors have a strong hold on our thoughts about user interfaces because they initially ease the learning of a very unfamiliar device - the computer - by providing familiar, reassuring images of comfortable everyday things like desktops and Rolodexes. With the help of these friendly objects we can placate the beast and get our work done.

This is fine in and of itself. However, we seem to have got caught in a time warp in which we still feel it necessary to search high and low for metaphors that provide the calming influence of miniature trash cans and tiny paint brushes so that we can tell the computer what to do. We are still concentrating on how to communicate with the machine, and not on how to use the machine to represent and express the rich semantics of the domains in which we work. As we develop software systems that are application-specific and that have knowledge of the semantics of particular domains, we need to go far beyond devising clever ways of portraying the computer itself.

In particular, we need to discover how to visually represent application semantics in many complex domains - to use the user interface to its fullest potential to capture and depict rich application semantics. For example, in their article on user interfaces for scientific visualization, Williams et al. (1990) explained that their goal is to "provide a user interface to data in situations where traditional database management systems, spreadsheets, statistical analysis packages, or image processing techniques are insufficient, ineffective ... or otherwise inappropriate." The authors have devised an iconic database language that is abstract and domain-independent. Underlying patterns in large datasets are revealed as "the interesting gradients and contours in the auditory or visual texture of an iconographic display." The iconographic displays have been created for data from various databases, e.g. medical scan data (such as brain scans).

The authors of this iconographic database language ask: "What kinds of structures in data can people recognize by discerning gradients, contours, and contrasting regions in the

---

6The alert reader will have already noticed that these formulations of metaphor go beyond the strict definition of metaphor meaning a figure of speech of the form "X is Y" (Carroll and Mack, 1985). What is really meant is analogy, a construct that allows us to infer that "if two or more things agree with one another in some respects they will probably agree in others," as Webster's Ninth New Collegiate Dictionary says. We don't think it worthwhile to be too picky about the differences between metaphor and analogy however, (at least with respect to user interface design), since Webster's defines a metaphor to be a figure of speech "in which a word or phrase literally denoting one kind of object or idea is used in place of another to suggest a likeness or analogy between them" (our emphasis).
texture of an iconographic display? Which icon parameters convey the most information? What are the effects of icon style, icon size, icon density, and icon overlap — and are these effects orthogonal or interdependent? What ways of associating data fields with icon parameters are the most effective in a given data domain? . . .” These questions arise from the problem-solving semantics of the data the authors wish to portray — not from the problem of interacting with the machine. It is this kind of work that deserves our interest and resources. It is the kind of work for which metaphors with their simplicity and imprecision cannot hope to carry the semantic burden.

Let’s look more closely at metaphors and what they can and cannot do.

4.2 Metaphors lack precision

Halasz and Moran (1981) and Johnson (1987) have pointed out that metaphors are bound to be incomplete representations of the systems they are meant to expose. While a syntactic user interface can get by with introducing a whole new concept such as hierarchical nested files, even though such a concept departs from the “files in folders” metaphor, an iconographic database language used, for example, to show brain scan data, (Williams, et al., 1990) cannot afford such laxity. It must be based on an underlying system of precise semantics, a system that is internally consistent and complete. Note that Williams and her colleagues have created an abstract data language, not a metaphorical one. Metaphors are good at suggesting a general orientation, but not good at accurately encoding precise semantics. Halasz and Moran (1981) observed that people usually can easily derive one basic meaning from a metaphor, but have difficulty “sort[ing] out the relevant mappings and allowable inferences.” It is precisely these mappings and inferences that we want to capture in exact form for the complex scientific, engineering and business applications that we are interested in, making metaphors an inappropriate choice for the overall design of the user interface for such applications.

4.3 Metaphors and the practical programmer

We see little opportunity to create reusable metaphors that can be plugged into specific applications. As argued above, applications have their own semantics to capture and metaphors aren’t equal to the task. But even if they were, there are other objections to their use.

In a paper that recommends metaphors as a basis for user interface design, Carroll and Thomas (1982) provided eight recommendations for producing good metaphors. Their very first recommendation is that metaphors be formulated on a case-by-case basis, taking into account the specific system to be metaphorically represented, as well as the given user population expected to use the system. (Johnson, 1987 made the same point.) This recommendation implies a very large design space — potentially infinitely large — and it is hard to see where to begin in creating reusable computational structures that would provide metaphors suitable for use “on a case-by-case basis.”

Blumenthal (1990) has attempted to create such structures. He programmed a library of reusable metaphorical structures for simple database applications. In his system, a database interface can be metaphorically pictured as a Rolodex, an address book, an index card file, a note pad, and a generic form. The designer selects the desired metaphor, and the program supplies the basic graphical forms and takes care of many of the layout
chores.

This is extremely useful in terms of saving programmer time, and the automatic layout features of the program are impressive. However, Blumenthal's system captures only superficial stylistic differences between applications—and nothing about application semantics. It works only for stock database applications, and not the specialized applications we want to move toward, such as the database of medical scan information used by Williams and her colleagues (for which a Rolodex simply would not do).

Carroll and Thomas's (1982) seven other recommendations should be carefully read by anyone contemplating designing an interface around a metaphor. The recommendations span four pages of closely spaced text, and include the dispiriting news that metaphors have to be discarded—in fact users have been weaned from them, to use Carroll and Thomas's terminology—because they eventually run out of metaphorical steam and are no longer useful.\(^7\)

That is undeniably true (see Halasz and Moran, 1981), and it makes it pretty hard to program metaphors into the user interface. Since users have to be "weaned" from their interfaces, we could of course have different user interfaces for different kinds of users, and different user interfaces for the same user at different points in his or her understanding. But that idea has serious drawbacks. From a practical point of view, there is too much coding, and potentially too many cycles devoted to deciding among interfaces (when that is done by the program). Multiple interfaces also make computer-supported cooperative work much more difficult, as group members do not share a common view of their data.

In short, metaphors do not promise to provide a rich set of reusable application objects that can be bundled together into toolkits to support the development of specialized applications. It is too difficult to get the right metaphor because of immense variation across applications and user populations. And even some of those who advocate using metaphors recommend discarding them after initial use because they lack the structure needed to transmit complex, precise semantics.

4.4 Prior knowledge and metaphors

Metaphors are slippery things, and not just because they contain irrelevancies (do we set the trash can out for garbage pick-up on Friday mornings?) and incompletenesses with respect to the domain they are meant to represent. More than that, we give them more responsibility than they can handle—they are like very clever children of whom we end up asking much too much, because they seem so bright and able. Metaphors tempt us to over-generalize and to forget distinctions that we should be remembering. For example, Carroll, Mack and Kellogg (1988) identified the ledger sheet as the "metaphor" underlying spreadsheets, and paper forms as "metaphors" for electronic forms. But spreadsheets are not based on any metaphor, nor is a paper form a metaphor that inspires the electronic

\(^7\)Johnson (1987) related a funny story that is more about weaning designers from metaphors: "A representative of Apple Computer said in a presentation on the Lisa user interface several years ago that though the Lisa's designers had tried to avoid deviations from the Desktop Metaphor as much as possible, one deviation that seemed necessary was leaving ghosted icons on the desktop when they were opened so that users could see where they had been. When I pointed out that the ghosted icons could be regarded as areas on the desktop where there was no dust, he said he couldn't wait to get back to Apple to tell the others that they hadn't deviated after all." Johnson goes on to note that, his facetious suggestion notwithstanding, the designers had of course deviated from the Desktop Metaphor: "Objects on real desktops do not shrink down to small, schematic representations of themselves when not in use."
form. Let's look at these putative metaphors in detail.

It is a mistake to think of a paper form as a metaphor for an electronic form. The paper form is a correct model of what is wanted electronically. The electronic form will differ to the extent that the computer has additional capability for handling forms, not to the extent that the paper form fails to model the electronic form. It does not in fact fail; the paper form is the true basis for the electronic form. The litmus test for this case is that the paper form will not eventually be discarded or ignored as the basis for the interface, as is the case with certain aspects of, say, the desktop metaphor. The functionality of the paper form will be extended and enhanced in the electronic version, and the paper form will continue to carry its original semantic content. In many cases the paper form itself will continue to exist as another version of the information. The paper form does not become irrelevant to experienced users, it does not contain the seeds of its own destruction by really being about something else – the paper form is exactly that which the computer models and extends.

Spreadsheets are not metaphorical ledger sheets. Spreadsheets are tables. A spreadsheet works because its tabular interface is a superb mechanism for representing and displaying the relational semantics of numerical quantities (Nardi & Miller 1990a, 1990b). Furthermore, a table is not a metaphor. It is a generic visual notation that can be used to capture the semantics of an astonishingly wide variety of applications. Visual notations have been created and used extensively by people all over the world because they visually encode relational semantics. It is important that we distinguish the precise semantic intent of a visual notation, and the much less formal, more literary intent of a metaphor. A table does not capture application semantics by being like something – it offers a diagrammatic form that provides a solution to many modeling problems. Rows and columns are an effective way to display large datasets of individual quantities that do not map into a geometric space. A table is an organizing device, not a metaphor that recalls something familiar.

A ledger sheet is also a table – it works not because it in turn is like something else (what else?) but because as a table, it serves to systematically organize and present numerical data. In the spreadsheet study, in hours of open-ended interviews, not one person ever mentioned ledger sheets in any context – including accountants who would know about them. Many people who work with spreadsheets do not even know what a ledger sheet is, nor do they need to know to be successful with spreadsheets.

The conceptual slippery slope of metaphor has led some investigators to confuse any kind of prior knowledge with metaphor. Tables are familiar to most people, and thus leverage prior knowledge, but tables are not metaphors any more than the fact that we know that the Normans conquered England in 1066 is a metaphor. Carroll and Mack (1985) confuse prior knowledge with metaphor, saying, “[interpreting a graph or chart] requires prior knowledge ... and [thus] engages metaphor.” Reading a graph certainly requires prior knowledge, but it is not metaphorical knowledge. Jarvenpaa and Dickson (1988) pointed out that people have to be explicitly trained to correctly interpret data in graphs – a graph has a precise meaning, and that meaning is not ascertainable from one’s knowledge of something else.

Metaphor is surprisingly seductive. It promises easy answers to hard questions. We may imagine metaphor where it does not exist, even when the familiarity of prior knowledge is

---

8Cameron (1989) noted that tables have been in use for 5000 years. Inventory tables, multiplication tables and tables of reciprocal values have been found by archaeologists excavating Middle Eastern cultures. Ptolemy, Copernicus, Kepler, Euler, and Gauss used tables.
not a source of confusion. We found a humorous example of this in Johnson’s (1987) story of the ghosted icons. A more serious misapplication is Carroll et al.’s (1988) suggestion that object-oriented programming is based on metaphors – the metaphor of physical objects. To say that there are metaphors – such as physical objects – that illuminate individual concepts of object-oriented programming is one thing. To propose that object-oriented programming is based on metaphors, that is it fundamentally metaphorical, is to miss the essence of a system whose complex internal logic grew out of needs peculiar to the programming problems it attempts to solve, and which must be understood completely in its own terms. Those who struggle to learn object-oriented languages might indeed wish for a civilizing metaphor to bring order to the object-oriented frontier of parametric polymorphism, contravariance, virtual functions, operator overloading and other concepts which must be grasped in their own right. Concepts of object-oriented programming will yield to no quick metaphoric fix. They are quintessentially unlike anything we are familiar with in everyday life, and will remain resistant to our best efforts to illuminate them in terms other than their own.

5 Visual Formalisms

In our discussion of mental models we emphasized the utility of external representations of application semantics and the need for software to support such representations. The examination of metaphors in the user interface focused on the need for semantic precision and expressivity in user interface languages. All along we have been concerned with the practical problems of application development and with providing designers and developers useful concepts and tools.

Visual formalisms – diagrammatic notations with well-defined semantics for expressing relations – provide a strong basis for dealing with these concerns. In his article “On Visual Formalisms” (1988), David Harel stated,

The intricate nature of a variety of ... systems and situations can, and in our opinion should, be represented by visual formalisms: visual, because they are to be generated, comprehended, and communicated by humans; and formal, because they are to be manipulated, maintained, and analyzed by computers (emphasis in original).

Harel emphasized the need to model complexity and the importance of visual problem solving using semantically rich languages:

We believe that in the next few years many more of our daily technical and scientific chores will be carried out visually ... The languages and approaches we shall be using in doing so will not be merely iconic in nature (e.g., using the picture of a trash can to denote garbage collection), but inherently diagrammatic in a conceptual way ... They will be designed to encourage visual modes of thinking when tackling systems of ever-increasing complexity, and will exploit and extend the use of our own wonderful visual system in many of our intellectual activities.

Computer-based versions of visual formalisms define formal semantics for familiar visual notations such as tables, graphs, plots, panels and maps. These notations have been refined over hundreds, if not thousands of years (Tufte, 1983; Cameron, 1989), and provide
a logical starting place for a set of objects comprising an application development toolkit. Although we expect to see new visual formalisms emerge over time, creating a successful new visual notation is no easy task. Luckily, the familiar, standard ones still have a lot of fire power in them, and can be specialized to meet the semantic requirements of many domains (Rumelhart, Smolensky, McClelland & Hinton, 1986). For example, Harel's (1988) statecharts, which formally describe a collection of sets and the relationships between them, combine graphs and Venn diagrams. Although Harel's work is quite new, Bear, Coleman and Hayes (1989) have already created a specialization of statecharts called objectcharts, for use in designing object-oriented software systems. And Heydon, Maimone, Tygard, Wing and Zaremski (1989) specialized statecharts to model a language for specifying operating system security configurations.

The purpose of implementing visual formalisms as application objects is not to supply programs to draw tables or lay out graphs. The purpose is to provide strong representational, editing and browsing capabilities in reusable objects that can be specialized for specific applications. The components of a visual formalism, for example cells in a table or nodes on a graph, must be capable of containing complex semantic information. An implementation of a visual formalism should provide editing and browsing capabilities that make sense for a particular form, e.g. the ability to select blocks of cells in a table, or to visually filter out the columns in a table which meet some test.

Visual formalisms are application frameworks. They provide a specific orienting framework in which to cast an entire application. For example, a spreadsheet, a specialized table, allows users to develop applications in which the relations between numerical quantities are laid out and organized within the rows and columns of the spreadsheet table. The spreadsheet provides a structure into which a model is cast. Users do not have to invent a structure – it is given to them. The initial phase of a modeling problem is reduced to simply recognizing a format into which a problem is framed, rather than being faced with the necessity of inventing a format from scratch (Nardi & Miller, 1990b).

An application framework is very different from a widget. A widget is a simple object whose purpose is to accept user commands or arguments, or to allow for the display and/or editing of simple data values. It exists not to organize whole applications, but to give users access to small but significant pieces of an application. A visual formalism is intended to help developers organize and present an entire complex application, or a large piece of one. Visual formalisms are different in kind from widgets because they encode and visually reveal relational semantics. A widget is much more limited in scope; for example, a widget may show the value of a single variable, or provide a means of issuing a single command. A visual formalism will in most cases be used in conjunction with widgets – for example selectable nodes on a graph might reveal pop-up menus when buttoned. Several visual formalisms may be linked together to comprise a single large application.

### 5.1 Visual formalisms in a nutshell

Visual formalisms are well-suited to the semantic and presentational goals we have laid out for them – i.e. providing manipulable external representations with well-defined semantics – for several reasons:

- **Exploitation of human visual skills.** Visual formalisms are based on human visual abilities, such as detecting linear patterns or enclosure, that people perform almost effortlessly. Visual formalisms take advantage of our ability to perceive spatial re-
ationships and to infer structure and meaning from those relationships (Cleveland, 1990). Visual formalisms are capable of showing a large quantity of data in a small space, and of providing unambiguous semantic information about the relations between the data.

- **Strong semantics.** Of the visual formalisms we have identified, graphs have defined for them the most formal semantics, and panels the least. Any implementation of a visual notation as a visual formalism will supply its own formal semantics. A computer-based version of a panel may only specify that a panel is an arbitrary collection of objects that belong together. Its strength as a presentation format makes it an extremely useful object, and an implementation might have a great deal of presentation functionality in addition to the semantics of being able to handle a collection of objects correctly for a given application.

- **Manipulability.** Visual formalisms are not static displays, but allow users to access and manipulate the displays and their contents in ways appropriate to the application in which they are used.

- **Specializability.** Visual formalisms provide basic objects that can be specialized to meet the needs of specific applications. They are at the right level of granularity — neither too specific nor too general. Visual formalisms are appropriately positioned between the expressivity of general programming languages, and the particular semantics of applications.

- **Broad applicability.** Visual formalisms are useful because they express a fairly generic set of semantic relations, relevant to a wide range of application domains. Because a large number of applications can be designed around a given formalism, visual formalisms will eliminate a great deal of tedious low-level programming, as well as give developers ideas about editing and browsing techniques with which they may not familiar, such as the use of fish-eye views for large datasets (Furnas, 1986; Ciccarelli & Nardi, 1988).

- **Familiarity.** Because the standard visual notations are so useful, they are found everywhere. Not only do they draw on innate perceptual abilities, but through constant exposure we become very familiar with them. Our schooling explicitly trains us in the use of the basic notations; e.g. using calendars and learning matrix algebra provide experience with tables. Everyday activities provide opportunities to create and use visual notations, such as writing a laundry list or reading a map.

### 6 An Implementation of a Table Visual Formalism

We have argued that visual formalisms are especially important to user interface design because they will provide practical help to developers. In order to be as specific as possible about exactly the kind of practical help we are talking about, in this section we briefly describe our implementation of a Table Visual Formalism. The Appendix contains more detailed information on the Table classes. Our implementation suggests how visual formalisms can become reusable computational objects collected into software libraries to form a basis for the development of specialized applications.

Implementing a visual formalism requires defining (1) the semantics of the visual formalism itself, such as correct table structure, and manipulations on individual components of the
visual formalism, e.g. cells in a table; (2) the presentation of the visual formalism, including multiple presentations of a given instance of a visual formalism; and (3) the application semantics expressed within the framework defined by the visual formalism.

We have implemented a Table Visual Formalism as part of our object-oriented development environment, ACE (Application Construction Environment) (Zarmer, 1989; Zarmer & Canning, 1990), implemented in C++ and the Interviews user interface toolkit (Linton, Vlissides & Calder, 1989). The contribution of the Table Visual Formalism is to give developers high-level functionality for developing applications for creating, structuring, modifying, editing, and browsing tables. Unlike the work of Beach (1985) and Cameron (1989) whose programs draw complex tables, we are concerned with providing object-based table structures that can be active application components, not static pictures.

As with any toolkit, we believe that carefully designed reusable objects provide significant leverage to developers. Our Table Visual Formalism benefits from being a part of ACE which provides abstractions such as dialog management, change notification, and presentation in an advanced user interface architecture modeled after Nephew (Szekely, 1989), with significant extensions (Zarmer, 1989).

We define a table as a rectangle filled with contiguous, non-overlapping rectangular cells, each of which contains a discrete piece of information (a content entry). A content entry is any object, e.g. an integer, string, bitmap, or instance of an application-specific class. Each content entry has a semantic and presentation component.

Our Table Visual Formalism differs from a spreadsheet in that: a cell can have any type of object for its content; a general programming language can implement cell behavior; it is more structurally flexible, allowing spanning rows and columns; it can be specialized, at a lower level than spreadsheets, into different applications such as a hospital flow sheet, an interactive data display, a decision table.

Both the table as a whole, and each cell within the table, are components with their own presentation and semantics. The Table Visual Formalism provides functionality for adding, removing, splitting, joining, moving and copying cells. Users can define named regions (rectangular groups of cells), associate keywords with a region, and iterate operations over regions. Tables can be scrolled horizontally and vertically, and zoomed in and out for different views. Portions of the table can be visually filtered to provide more focused views of the information in the table. Individual cells can pop up and pin down “big cell views” so that large tables with small cells can show the information in selected cells in more detail. (Please see the Appendix for more detail on the Table Visual Formalism.)

7 Summary

We believe that the next generation of software programs will emphasize helping users to solve complex problems in scientific, engineering, and business applications. We compared mental models, metaphors and visual formalisms as a basis for designing the user interface for such applications.

Mental models provide little guidance to software developers because they are apt to be unstable, incomplete, inflexible and difficult to discover. Instead, developers should concentrate on understanding users' high-level task goals and the data structures and manipulations that must be provided to create explicit models that allow users to achieve
their goals. Users’ domain knowledge is also of critical importance, but developers should not expect to find it in the form of stable, well-articulated models, as there seems to be little evidence from cognitive psychology and other related disciplines that such models exist. Kintsch’s (1988) work is of particular interest as he suggests that propositional knowledge is activated and integrated in the immediate context of the problem to be solved, to provide flexibility and sensitivity to the particulars of the situation.

Metaphors are useful in the syntactic user interface in providing a general orientation such as a desktop. They are inadequate for the semantic interface as they cannot encode complex application semantics with any precision. The goal of a semantic interface is not to “seem like” something else, but to faithfully model true application semantics. We argued that metaphors receive a lot of attention in the user interface world because they promise easy elucidation of difficult concepts. But their utility is limited to the learner’s first encounter with a subject, and the problem of understanding the deeper complexity of a system remains, and must be dealt with by coming to grips with the semantics of the system itself.

Visual formalisms offer great potential for designing the semantic user interface. Visual formalisms encode precise semantics, take advantage of human visual abilities, provide macro-level frameworks for application design, and can be offered as reusable computational structures to support the development of specialized applications. As we saw with spreadsheets, visual formalisms can do much more than display data – they actively help users to work through complex problems. This is perhaps the highest calling of the user interface – to be an active participant in an interactive exchange with users engaged in solving difficult problems. Visual formalisms support users in problem solving activities by providing an over-arching application framework, and a clear visualization of application semantics.

It is important that the ways we devise for designing the user interface offer direct support to developers. Because visual formalisms already exist as recognizable visual notations, they map easily into computational objects that can be collected into reusable, extensible software libraries. We described our implementation of a Table Visual Formalism to suggest how the semantic and presentational aspects of visual formalisms can be managed in an object-oriented table structure that is a high-level building block for designing and implementing specialized applications.

8 Acknowledgments

Thanks to Lucy Berlin, Betsy Brenner, Robin Jeffries, Jeff Johnson, Nancy Kendzierski, Jim Miller and Jasmina Pavlin for helpful comments on earlier drafts of this paper.

9 Appendix – Table Classes

The presentation and semantic aspects of Tables are implemented through 6 classes:

- **Table** represents the semantics of a table as a whole, in particular adjacency information about cells. The operations on Tables mainly deal with changing the *structure* of a Table – the arrangement of rows and columns.
• Cell represents an individual cell of a table. Users will normally be unaware of Cell objects – they will only think of the content object (the integer, string, etc.) that is in the Cell. The Cell can also have names, relational expressions (to express relations within the table), etc. associated with it.

• TableView is the visual presentation of a Table. It translates the semantic adjacency of Cells to a visual arrangement of CellViews. There can be more than one TableView for a given Table to provide multiple presentations of the same Table.

• CellView presents a Cell. The generic CellView provides visual feedback (e.g. highlighting) for selected Cells (i.e. cells the user selects with mouse or keyboard), and determines how to present the Cell's content object.

• TableEditor supports the editing process, providing interface state, such as a currently-selected-cell, which must be maintained on a per-user basis.

• TableEditorView is a presentation of a TableEditor. It provides a graphical interface for the TableEditor's editing operations, and a presentation of interface state, such as a presentation of the currently selected object.

A read-only table application can be implemented with the first four classes. A table that will be edited will make use of the last two classes as well. Each of these classes can be specialized through subclassing.

Class Table represents the semantics of tables, that is, the semantics of the table structure itself (not the content entries in cells, which can be anything pertinent to the application). It implements a “table engine” that manages a set of cells and enforces the structural rules of Tables, but provides no user interface. A Table is able to grow or shrink to hold any number of Cells, but is finite at any given time. Table operations can add or remove cells from the Table, or change semantic relationships, expressed as adjacency, between cells.
A Table has operations to *split* and *join* cells. In the Table shown in Figure 1, the cell labeled “Widgets Sold” is a cell that has been joined to span several other cells.

A **section** is a portion of a table that is defined by any two parallel lines that extend completely across the table, either horizontally or vertically. In other words, a section is one or more adjacent full-length rows or columns. Sections are the unit for the operations that *add*, *delete*, *move*, or *copy* sets of cells into or out of the table, because adding or removing a section is the only unit guaranteed to preserve a rectangular table. In Figure 1, a section is shown in light gray. Note that a section can cut through part of a cell. Deleting this section would eliminate the fully-enclosed cells (like the one labeled “3D”), and would shrink the partially-enclosed cells (like the one labeled “Widgets Sold”).

A **region** is a rectangular group of contiguous cells within a table. A region could contain an entire table, or only a single cell. A region identifies a collection of cells and associates a name or keyword(s) with it. Names and keywords be accessed by relational expressions, table viewer operations, etc. A region is shown in dark gray in Figure 1.

To **construct** a table, a user can either (1) ask for a default 1 x 1 table, and then edit it using the re-structuring operations (join, split, add-cell, etc); (2) ask for an “infinite” table, that is, a very large table, such as those found in spreadsheets; (3) edit the table properties that specify the number of rows and columns in the table. Tables can be saved with all structure and content intact.

The generic Table class can be specialized to add, restrict, or remove operations. For example, a spreadsheet application might remove the split and join operations, to ensure that the table remained a grid. Other applications might add more specific editing operations, such as inserting a section with a particular format of split and spanning cells.

**Class** **Cell** stores table information needed on a per-cell basis. The main component of a Cell is a content object. A Cell need not have a content object, and the generic class Cell does not automatically create one.

Many applications will be able to use the generic class Cell. Reasons for creating a subclass of Cell might include: supporting multiple values in a cell; restricting content object types; providing specialized content parsing/unparsing; or providing specialized cell relations.

**Class** **Table View** arranges presentations of a Table’s cell views into rows and columns that reflect the relationships between the cells. **Table View** is responsible for deciding how big each row and column should be, and provides operations for end users to override the default sizes.

**Table View** provides means of managing large tables. First, one or more sections can be **filtered** (removed) from the display. This does not affect the semantic Table object at all; it simply leaves more room for other cells to be seen. Using named regions, keywords and constraints, users will be able to request, for example, that a Table “Show only columns with budget estimates over $1000.” The **Table View** can be **scrolled** horizontally and vertically. **Zooming** allows a user to zoom in on a detailed view or zoom out for a larger view.

The major reason to create an application-specific subclass of **Table View** is to change the layout policy. One application might require all rows and columns to be exactly the same size, while another might require that row and column size be based on the size of the cell.
content presenters (e.g. giving a large string extra room). Applications may differ in how they want a Table View to respond to a change in the overall size of a Table: one may wish more/fewer cells to be shown, while another may want the same cells to be shown, only larger/smaller.

It is important to note what a Table View does not do: it is not involved with the manipulation of a Cell’s content (the contents’ own presenter does that), or feedback about selected cells (the Cell Views do that). Table View does not choose the gestures or dialog styles used to invoke Table operations – these are chosen by the application designer using ACE components.

_Class Cell View_ provides presentations of a cell. Cell View objects know the color, shading, outlining, background pattern and special formatting of cell contents (e.g. numbers shown as dollar amounts) of a cell presentation. This is useful so that tables collected into libraries can be used as templates for other applications. It is also useful when copying regions. Users may want to set up tables with this kind of information even before they have information about cell contents.

Cell View has two main jobs: to provide feedback for selected cells, and to choose a presenter for the cell content object. Subclasses might change either the type of feedback given or the policy for choosing presenters.

A Cell can have more than one presentation, including larger pop-up views that show more detail than can be shown in the cell within the Table (for cells with e.g. a large bitmap, or a lot of text). These “Big Cell Views” can be popped up on request, moved to any place on the screen, and left on-screen if desired, so that several may be viewed at once. Having Big Cell Views combines the advantages of the table’s presentation of relational information for many cells, with the ability to see more detail on selected cells.

### 10 References


