Putting Metaphors to Work

John M. Carroll and Mary Beth Rosson
Department of Computer Science
Virginia Tech (VPI&SU)
Blacksburg, Virginia 24061-0106
U.S.A.

e-mail: carroll@cs.vt.edu
rosson@cs.vt.edu
Telephone: 1-703-231-6931

Abstract

Metaphor has proven to be one of the richest and most robust ideas in the design of computer applications and user interfaces. The basic idea is very simple: present functionality in such a way that the user can access and apply specific prior knowledge while learning and using a novel tool. But the practical and theoretical ramifications of this idea both in the brief history of human-computer interaction and in its current prospects are quite considerable. In this paper, we first summarize a view of the relevant history. We then develop the notion that metaphors should be conceived of as bound to contexts of use: The recognition and interpretation of metaphors typically depends upon the establishment of a meaningful task context. We think there is a need to focus consideration of metaphors on the scenarios of use from which they arise. We suggest that this reconception of metaphors as bound to scenarios of use converges with recent developments in scenario-based specification and object-oriented design, and that it provides new opportunities for putting metaphors to work in the specification, design and implementation of systems.

Keywords: Interface Metaphors, User Interface Architecture, Object-oriented Analysis and Design, Scenario-based Design

1. Metaphors, Yesterday and Today

A major theme in the emergence of human-computer interaction is a steady movement toward more direct incorporation of contexts of use into the objects of analysis and design. Metaphor has both contributed to this evolution and been redefined by it. In the late 1970s, most considerations of ease-of-use and ease-of-learning focused on low-level actions (e.g., keystroke counting and formal properties of command languages). The design and analysis of metaphor in this period helped to refocus the field on semantic relationships among interface objects and actions (Carroll & Thomas, 1982, Smith, Irby, Kimball, Verplank & Harslem, 1982).

Throughout the 1980s, the analysis of interface metaphors was one of the key arenas in which classically structuralist models of human cognition were increasingly rejected as a foundation for human-computer interaction in favor of analyses that took seriously the goals, prior knowledge, and activity contexts of the people who use computer systems and applications (e.g., Carroll & Mack, 1985; Carroll, Mack & Kellogg, 1988). For example, the cognitive science analysis of metaphors as quasi-formal "structure mappings," so appealing in laboratory demonstrations, was not capable of analyzing the fact that people actively construct and creatively use metaphors through a process of interaction and discovery. This process — the part the purely structural analysis idealized away — proved to be the most interesting part of what was going on.

Today, metaphors are pervasive in human-computer interaction; they are, for example, a major theme of the FRIEND21 Project (Nonogaki & Ueda, 1991). Metaphor is so thoroughly integrated into analysis and design work on applications and user interfaces that the topic as such can be hard to discern. To a great extent, metaphor *is* human-computer interaction. The challenge today, we believe, is to drive the analysis of metaphor even more aggressively toward the incorporation of contexts of use, and to develop software architectures and development environments that support metaphor-based design.

In its first two decades, human-computer interaction has gotten a lot out of metaphor. It should continue to find new ways to put metaphors to work.

2. Metaphors inhabit tasks

Metaphors can be engaged (that is, recognized or constructed by users) purely by a user interface's look-and-feel: to wit, if something looks like a trashbin, if one discards files by dragging them to it, then one should think of it as a trashbin. However, more typically metaphors are evoked in the context of a task a person is trying to accomplish, that is, by relatively substantial goals and activities. A gedanken experiment can convey what we have in mind. There are many properties of a desktop system that suggest its central metaphor: The various applications and data present themselves with suggestive icons (e.g., electronic mail as a little mailbox). The icons may be moved about by direct manipulation techniques; they can be "dragged" to the trashbin and "dropped" into it. The various applications can be interleaved and integrated to work through a task scenario; for example, one can start up the electronic mail application to create a message, check some figures in a spreadsheet, import a memo attachment from a notebook, and then send the mail.

Which of these properties is most critical to the desktop metaphor? The picture icons can be replaced with labeled boxes, and dragging can be replaced with source-and-destination clicking — both with little consequence. What seems to matter more is that a variety of functions and data are spatially presented and simultaneously accessible, that they can be seen, accessed, and used together as necessary to complete a task. Thus, systems without multitasking capabilities or systems that can only display applications fullscreen (thus destroying the spatial task context for the user) cannot support a serious desktop metaphor.

Examples like this lead us to conclude that metaphors depend on the activity contexts in which they occur. The various desktop objects must behave consistently at the task level; for example, all of them share data in the same way, "opening" always means the same thing, and so forth. At the lower level of physical actions, they may, and sometimes must, have idiosyncratic behaviors (the trashbin has different behaviors than other folders with respect to the files it contains; it appears fatter when it contains something).

Activity contexts underpin some of the most interesting and important properties of metaphors that have been discussed over the years. Without a use context, there can be no composite metaphors: sets of metaphors must exist together in a meaningful context in order to interact. For example, a piece of mail one is examining in an in-box evokes a document metaphor. The folder organizing prior correspondence with the sender of that mail engages a file-tree metaphor. But the two comprise a composite metaphor for handling mail only in the context of a mail-handling task (e.g., the recipient wishes to reply to the piece of mail and in doing so to remind himself or herself of prior mail correspondence with the sender).

Task contexts delimit the extent to which composite metaphors are analyzed. Thus, as one opens the folder of prior correspondence, peruses past mail, and then returns to the mailer to finish editing a response, one also incidentally learns something about the composite metaphor. But the user probably only analyzes the metaphor to the extent he or she needs to for this task. The person probably does not ponder the data structure implications of a list of documents each with an embedded file hierarchy. To the extent that the available functionality matches what one wants to do, one may not even be aware of the metaphor as a metaphor.

A way of seeing how dependent metaphor compositions are on task contexts is to observe that when compositions span task boundaries, their mutual contradictions may not be noticed. Carroll and Lasher (described in Carroll et al, 1988) studied a person using a programmable calculator who referred to one input metaphor to understand error correction (namely, the notion that each new entry was inscribed into a log that could subsequently be edited) and another to understand the availability of calculations in routine, non-error situations (namely, the notion that each new entry was instantly assimilated into a cumulating result). The two metaphors had obvious mutual contradictions as comprehensive models for the device, but this apparently was not a problem in actual use.

Task contexts support and guide the recognition and resolution of cases for which a metaphor target mismatches aspects of the metaphor source. In the mail system example, the possibility of having your own folder for past correspondence appear "in" a new piece of mail mismatches the source domains of physical mail and physical folders. However, in the task context of answering mail, the mismatch may provide relevant and useful functionality that may help carry out a task and thereby make sense of the mismatch in a way that expands the concept of mail and file support. We suggest that the problems with "dynamic" metaphor mismatches, reported recently by Hirose (1992), might be understood as cases in which a metaphor is not adequately bound to a task context. However, we have not studied these reports enough to draw conclusions.

3. Metaphors specify tasks

The conceptual analysis of metaphors as bound to task contexts can support design work by helping to broaden the consideration of metaphor from the level of look-and-feel to the level of user goals and activities. This kind of "conceptual" application is currently the most common and most successful use of metaphors in design (e.g., Carroll et al, 1988, Madsen, 1994). Recent work has begun to develop "computational" applications of metaphor, that is, to bridge directly from the conceptual analysis of metaphor to software architectures and tools that organize and support the design and development of applications and user interfaces.

A simple example is a structure-mapping system developed by Blumenthal (1990) that reasons about a description of an application, a description of a real-world metaphor source, and a set of relations mapping the latter to the former. The system tries to maximize the number of mapped relations by evaluating relations not included in the design specification it is given, and by creating new application entities to correspond to given real-world entities.

The system incorporates a limited notion of task context in constructing mappings. For example, in evaluating a Rolodex metaphor for a data manager, the face-up Rolodex card (that is, the one above the spindle) was mapped to both the currently viewed data record (in Browse Mode) and the record template (in Add Record Mode). However, only in the Browse Mode mapping did the system map size and location attributes: the currently viewed data record has a size and location, but the record template does not.

Other work gives task context a more central role. Bass, Kazman and Little (1992) described a "conceptual architecture" in which metaphors are the central computational objects of the user interface. Bass et al. differentiate between three levels for describing applications: abstract tasks, metaphors, and physical tasks. Abstract tasks are a succinct characterization of the user's problem domain: the potential user interactions, the data structures and relations underlying those interactions. For example, the user of a mail system will want to view pieces of mail. Metaphors instantiate concepts from the abstract task level in concrete and familiar terms. For example, the user's mail is held in an in-box, which can be scanned, and its items opened for viewing. The in-box is an example of the "list" metaphor; viewing is a function of the "buffer" metaphor. Physical tasks implement metaphors within specific device contexts, that is, as specific sequences of user actions and system responses. For example, showing the in-box list might be achieved by double clicking on the mailbox icon. Thus, metaphors bridge between the abstract functional specification of the application and the low-level user actions that implement that specification.

A key aspect of Bass et al.'s architecture is the correspondance they create between metaphors and abstract data types that have real-world analogs. Thus, the in-box is a list, a concrete and familiar type of entity, that can also be formally specified as an abstract data type. Bass et al. use metaphors to modularize the specification of user interface functionality. They stipulate that metaphor functions can operate on only one data type. Thus, viewing a piece of mail can be defined as a function of the buffer metaphor, and scanning the headers of mail received can be defined as a function of the (in-box) list metaphor, but finding a piece of mail can only be defined at the abstract task level because it incorporates both scanning a list and loading a buffer.

There are many open questions about Bass et al.'s approach. For example, it is not clear what is lost from the conceptual notion of metaphor when it is equated computationally with abstract data type. Bass et al. emphasize lists, trees, buffers and queues as examples of metaphors. But it is notable that in their view the desktop metaphor itself cannot be analyzed as integral at the metaphor level. And other familiar metaphors, like rehearsal (Gould & Finzer, 1984), cannot be analyzed at all. Furthermore, even if this architecture produces a good system decomposition with respect to specification, it is not clear what its impact would be on system implementation. However, modulo these issues Bass et al. have raised a simple and bold framework for unifying the substantial tradition of conceptually-oriented work on metaphor with the up-to-now independent tradition of work on software architectures.

4. Metaphors design tasks

The concept of metaphor plays a central role in many object-oriented design methodologies, though this has only occasionally been clearly acknowledged (Rosson & Alpert, 1990). The object-oriented paradigm conceives of computation as message-sending interactions among highly encapsulated software objects. Thus, object-oriented design (OOD) methods focus on identifying key design objects and their "responsibilities" in these message-sending collaborations. In this "intelligent object" design paradigm software objects "know" about the tasks they will participate in (Rosson & Alpert, 1990). For example, a piece of mail knows how to format and send itself, and to what folder(s) it is related and how.

Objects in the OO paradigm are designed to be intelligent about the tasks they particpate in; the need to analyze such task-based responsibilities has prompted development of a variety of scenario-based OOD methods. In these methods, scenarios are analyzed to create a problem domain model of objects and responsibilities — this model is the starting point for the software design (e.g., Wirfs-Brock, Wilkerson & Wiener, 1990). For example, a weather display and simulation system might have as part of its starting representation a scenario in which someone wishes to explore what it would be like if a hurricane formed in the Gulf of Mexico and moved up the east coast of North America while there was a wintertime high pressure system over southern New England.

The problem domain model will contain objects like cold fronts, warm fronts, high and low pressure systems, and the storm system. This model is refined as the designer identifies appropriate abstractions among the objects and recognizes cross-scenario constraints among their attributes and relations. The designer may realize that a velocity should be modeled as a distinct object, or may recognize that warm and cold fronts share various sorts of behavior, for example, the tendency to move eastward.

The entities of the task domain are being used as metaphors, but not just conceptual metaphors. They are design objects. Physical characteristics and behaviors of weather objects (e.g., that cold fronts move in certain patterns, that their collisions with warm fronts cause particular changes to their structure, that the velocity of a storm system is influenced by factors such as topography and air pressure) are analyzed and encapsulated within appropriate design objects as responsibilities. In the resulting software, these computational entities will maintain those characteristics and enact those behaviors.

The emphasis on problem domain simulation as a first approximation OOD strategy is founded on the belief that physical structure in the world provides a good first approximation to a computational model that is modular and robust to change (Meyer, 1988). However, the approach is also implicitly oriented toward the design goal of supporting metaphoric understanding and discovery on the part of users: The design objects emerge from models of the extant problem situation, and are reinterpreted and refined as part of the software design process. The final design will reflect the initial metaphors derived from the problem domain model, but will also reflect the sharpened abstractions that have emerged. One benefit of developing software metaphors like this is to build a foundation for users who later will traverse these same paths of reinterpretation as they construct and use their own metaphors for using and understanding the application.

5. Metaphors implement tasks

The OOD methods we have described incorporate metaphoric views of problem domain entities as a vehicle for articulating a domain model. If development of the design occurs within an object-oriented language and environment, however, the contribution of these problem metaphors can extend beyond design into implementation.

Rosson and Carroll's (1993) Scenario Browser tool supports this more extended contribution of problem metaphors. It coordinates the development of textual sketches of events and interactions comprising a scenario with the development of Smalltalk/V® code (Digitalk, 1989) implementing these scenarios in a persistent-object workspace. The Scenario Browser also coordinates the text and workspace views with various rationale views in which the designer records arguments for various design decisions. Thus, as a developer designs and implements the objects underlying a set of scenarios, his or her reasoning may be captured for later use by other designers (for example, as the metaphors are evolved or reused in other scenarios) or by users.

5.1 Developing scenarios. Development of an application in the Scenario Browser is an iterative process of scenario elaboration, analysis and evolution. For the weather simulation example, a designer might begin by sketching out a variety of high-level learning tasks. The Scenario Browser provides a typology of six general usage situations that encourage broad coverage of potential user concerns; the situations reflect our efforts to generalize over scenario sets we have developed for a variety of applications (Carroll & Rosson, 1992). The situations include orienting (e.g., wondering about the kinds of things the simulation could be used for), searching (looking for some particular expected feature such as a "run animation" button), opportunism (choosing to explore further some aspect of the system, such as browsing a set of example simulations), procedures (carrying out a basic task like creating the hurricane simulation described above), making sense (trying to understand why the system behaved in some way, perhaps wondering why a newly created animation does not terminate but keeps cycling through over and over), and tuning (figuring out how to improve on one's current usage patterns, e.g., reusing a piece of one simulation in another).

The initial scenarios sketched in the Scenario Browser typically do not include details of user interactions. Our design approach encourages designers to first model the problem domain as a coherent set of basic tasks, and then to develop an appropriate user interface to this problem model (Goldberg, 1990). From the perspective of usability, this approach focuses early design attention more on usefulness than on ease of learning or ease of use. It also reflects our belief that a key ingredient of designing usable applications is the development of a meaningful problem domain model. Note that from the perspective of metaphoric thought, this means that the initial metaphor capture will focus on *what* a problem entity contributes to a task, not on the details of *how* it does this.

5.2 Psychological design rationale. Some designers may work exclusively with textual scenarios in reasoning about and elaborating a system's functionality. However the Scenario Browser encourages designers to make their reasoning more explicit by analyzing the "claims" associated with particular use scenarios. A claim encodes a bit of psychological design rationale — it consists of an artifact feature and its anticipated positive and negative consequences for a situation of use. For example, allowing users to run a developing weather simulation at any point encourages early and continued testing of their ideas, but may slow down their work, and may in fact discourage more systematic planning. The design method embodied in the Scenario Browser assumes that designers will work with such claims as the design progresses, capitalizing on positive consequences as much as possible while mitigating negative consequences.

An important source of psychological design rationale is a designer's reasoning about problem metaphors. In the hurricane scenario, putting a cold front object "in control" of its trajectory (e.g., it is responsible for finding out about the topography it is moving over and taking appropriate action, rather than making topographic features responsible for notifying fronts that move over them) makes the front more active as a metaphor than the land it travels over. While this may map well to our initial intuitions about weather objects, the designer might want to record this bit of rationale for continued evaluation as the design evolves.

5.3 Interleaving analysis, design and implementation. A key aspect of the Scenario Browser environment is that the specificaton of tasks is intertwined with the design and implementation of the software supporting the tasks. As soon as one or more basic tasks has been sketched out, the designer begins to build an object-oriented software "solution" for the scenarios. The software development activity is also scenario-based, in that it consists of the identification, instantiation, elaboration, and evolution of Smalltalk objects needed to implement individual use scenarios. This direct contribution of problem scenarios to implementation as well as analysis and design differentiates our work on the Scenario Browser from other scenario-based OOD methods (e.g., Jacobson, Christersson, Jonsson & Overgaard, 1992; Wirfs-Brock et al., 1990). The early and continuing interplay between the elaboration of use scenarios (as specifications of a system's functionality) and the implementation of these scenarios (as a representation of the software supporting the specified functionality) reflects our general belief that a design process that interleaves analysis, design and implementation is more likely to result in useful and useful systems.

Implementation of a scenario takes place within a persistent-object workspace associated with the scenario. The designer creates instances of existing Smalltalk classes (e.g., a particular weather object might be implemented initially as a Dictionary of attribute-value relations), creates and tests out messages needed to carry out the scenario (e.g., accessing the value of a characteristic like velocity, developing a response to a `contact:withForce:` message received from a hurricane object), and sets up the relationships among objects needed to enable their collaboration (e.g., deciding whether the hurricane object and the cold front should "point" to one another, or communicate indirectly through some other object).

The objects developed for a particular scenario can be manipulated both symbolically via Smalltalk expressions in the workspace and more directly via an "object map", a graph of the important objects contributing to the scenario (the designer decides which objects should be incorporated in this graph, but the graph itself is created by analyzing the chosen objects and their connections). Figure 1 depicts a graph of objects that might have been developed at some intermediate point in developing the hurricane scenario.

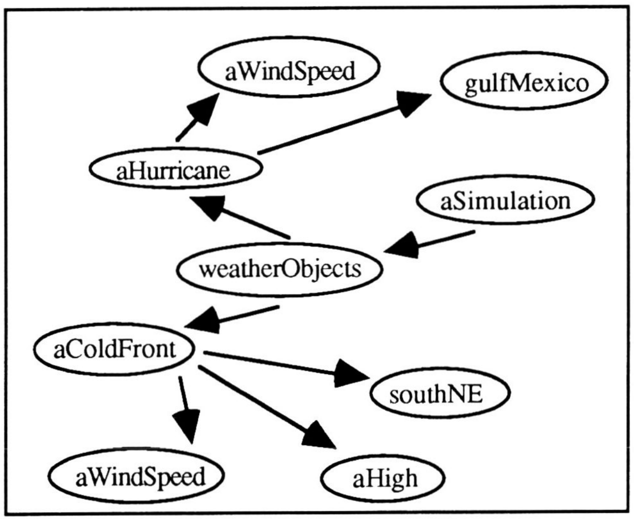

Figure 1. The nodes in the graph represent instances of Smalltalk classes that might have been developed to implement the hurricane scenario. Arcs represent instance variable connections (e.g., the arc from `aHurricane` to the object `gulfMexico` indicates that the latter is the value of an instance variable — perhaps `location` — of the former). The Scenario Browser allows designers to manipulate such object-object connections directly.

The object map is useful for working out what objects are needed to implement a scenario and how they should be connected. The actual code implementing these objects (i.e., the class definitions, the methods that are evaluated in response to messages passed among them) is created and maintained in a code browser. This browser is a Bittitalk Browser (Rosson, Carroll & Bellamy, 1990), a filtered view of the large Smalltalk/V class hierarchy that provides access to only the classes currently being used to implement a scenario's objects.

Designers using the Scenario Browser are offered a concrete problem situation within which to work out initial software abstractions. The concreteness of the situation aids in the identification of objects; perhaps more importantly, it provides a context for analyzing the interaction of these objects, which is a prerequisite for distributing object responsibilities and setting up the lines of communication necessary for objects to carry out their individual responsibilities (Wirfs-Brock et al., 1990). Furthermore, the specific scenarios set up very concrete situations in which to engage the intelligent object metaphor, taking different problem objects' "points of view" (Robertson, Carroll, Mack, Rosson, Alpert & Koenemann-Belliveau, 1994) to analyze alternative distributions of responsibilities and lines of control. The filtered Bittialk Browser contributes to this focus on a concrete situation, by presenting to the designer only the abstractions operational in the situation under development.

5.4 Software rationale. Just as a scenario specification can be analyzed for the claims it makes about an artifact's consequences for users, a scenario implementation can be analyzed for the claims it makes about the software's consequences for its users — the developers and maintainers of the software. Here again, the contributions of metaphor might be noted, in that the adoption of a particular problem metaphor might have ramifications for developers as well as users (e.g., making the Smalltalk abstraction for cold fronts more "active" might make it easier to extend in interesting ways as the application is used and enhanced).

Although we are only now beginning to experiment with application development in the Scenario Browser, we expect that the parallel development of tasks and their implementation will have a number of advantages. Designer-user communication is likely to be facilitated, as software design effort will always be directed at a particular scenario (or set of scenarios), providing a concrete task-oriented context for discussion about design alternatives. To the extent that the object-oriented solutions embed good models of the problem domain, this communication should be facilitated even more (Bruegge, Blythe, Jackson & Shufelt, 1992). Indeed, the reification of problem metaphors as software abstractions might serve an important role in extending users' initial understanding of what the system can provide.

Because reasoning about software alternatives begins early, before task specification has been completed, continual feed-forward from tasks to implementation, as well as feedback from implementation to tasks is enabled. In many cases, these interactions will be of a constraining nature — a task sets requirements for the software (e.g., that certain obvious weather objects like fronts and hurricanes should be respected in the software model), or the software constrains the details of a task (e.g., that some abstraction must serve in a "controlling" role). The advantage of our approach in these cases is that such constraints can be recognized early and their effects on the developing design can be analyzed and explored immediately within the context of the tasks they affect. Of more interest are cases in which a task raises new and unforeseen opportunities for its implementation, or vice versa (e.g., creating an active metaphor for a cold front may lead to new ideas for weather interactions that model situations not possible in the real world). System development often ignores the potential for mutual inspiration of task and software design; our hope is that environments like the Scenario Browser will make such interchange commonplace.

There are several similarities between the style of deign supported by the Scenario Browser and the style of design envisioned in Bass et al.'s conceptual architecture. Scenarios in the Scenario Browser are abstract tasks, but are perhaps less abstract (or at least more annotated) than what Bass et al. have in mind. The object model view of a scenario in the Scenario Browser is similar to Bass et al.'s view of the metaphor level as an analysis of tasks into abstract data types. Consistent with object-oriented design, the Scenario Browser takes a less constrained view of what can serve as an abstraction (i.e., a "class" in object-oriented design, which encapsulates both data and behavior). And the Scenario Browser encourages a more detailed implementation at this level; it would be typical to specify not only what the metaphors are (i.e., as classes with sample instances), but also their design interactions (lines of communication established through component and shared objects). Our experiments with the Scenario Browser have focussed on the design of tasks rather than a task's user interface, but Bass et al.'s view of the physical task level as implementation of metaphors is compatible with use of the Scenario Browser for user interface development.

6. Discussion

The observation that user interface and application metaphors are bound to task contexts entails that metaphors should be designed and developed as elements of designed tasks. Thus, it comprises another argument for the centrality of scenario-based design methods in human-computer interaction (Carroll & Rosson, 1990, 1992). More broadly, as we announced at the beginning of this paper, the observation that metaphors are bound to scenarios of use corroborates a larger movement in design theory toward more direct incorporation of contexts of use into the objects of analysis and design.

The conception and role of metaphor in human-computer interaction have come a long way from the structure-mapping notions of simple, static isomorphisms. It is standard now to seriously address the processes through which humans construct metaphors, the concerns they manage and the insights they experience, and the task contexts to which metaphors are bound. But perhaps best of all, this richer and more dynamic conceptual view of metaphor can be put to work in new ways, embodied in software architectures and development environments.

7. Acknowledgements

This paper is a further development of "Binding metaphors to scenarios of use" which appeared in the Proceedings of the FRIEND21 1994 International Symposium on Next Generation Human Interface (Tokyo, February 2-4).

8. References

Bass, L., Kazman, R. & Little, R. (1992). Toward a software engineering model of human-computer interaction. In *Engineering for Human-Computer Interaction, Proceedings of the IFIP TC2/WG2.7 Working Conference,* Amsterdam: North Holland, pp. 131-153.

Blumenthal, B. (1990). Strategies for automatically inforporating metaphoric attributes in interface designs. *Proceedings of UIST: The Third Annual Symposium on User Interface Software and Technology* (Snowbird, Utah, 3-5 October), pp. 66-75

Bruegge, B., Blythe, J., Jackson, J., & Shufelt, J. (1992). Object-oriented system modeling with OMT. In *Proceedings of OOPSLA'92: Object-oriented Programming Systems, Languages and Applications.* (Vancouver, British Columbia, October). New York: ACM Press, pp. 359-376.

Carroll, J.M. & Thomas, J.C. (1982). Metaphor and the cognitive representation of computing systems. *IEEE Transactions on Systems, Man, and Cybernetics, SMC, 12,* 107-116.

Carroll, J.M. & Mack, R.L. (1985). Metaphor, computing systems, and active learning. *International Journal of Man-Machine Studies, 22,* 39-57.

Carroll, J.M., Mack, R.L. & Kellogg, W.A. (1988). Interface metaphors and user interface design. In M. Helander (ed.), *The handbook of human-computer interaction.* Amsterdam: North-Holland, pp. 67-85.

Carroll, J.M. & Rosson, M.B. (1990). Human computer interaction scenarios as a design representation. In *Proceedings of HICSS-23: Hawaii International Conference on System Sciences,* IEEE Computer Society Press, Los Alamitos, Ca., pp. 555-561.

Carroll, J.M. & Rosson, M.B. (1992). Getting around the task-artifact cycle: How to make claims and design by scenario. *ACM Transactions on Information Systems, 10,* 181-212.

Digitalk. (1989). *Smalltalk V PM.* Los Angeles: Digitalk, Inc.

Hirose. M. (1992). Strategy for managing metaphor mismatches. *CHI'92: Posters and Short Talks*, page 6.

Jacobson, I., Christersson, M., Jonsson, P., & Overgaard, G. (1992). *Object-oriented software engineering: A use case driven approach.* Reading, MA: Addison-Wesley.

Madsen, K.H. (1994). A guide to metaphorical design. *Communication of the ACM,* in press.

Goldberg, A. (1990). Information models, views, and controllers. *Dr. Dobb's Journal, 166,* 54-61.

Gould, L. & Finzer, W. (1984). Programming by rehearsal. *Xerox Palo Alto Research Center SCL-84-1,* Palo Alto, CA.

Meyer, B. (1988). *Object-oriented software construction.* Englewood Cliffs, NJ: Prentice-Hall.

Nonogaki, H. & Ueda, H. (1991). FRIEND21 Project: A construction of 21st century human interface. *Proceedings of CHI'91.* New York: ACM, pp. 407-414.

Robertson, S.P., Carroll, J.M., Mack, R.M., Rosson, M.B., Alpert, S.R. & Koenemann-Belliveau, J. (1994). ODE: The Object Design Exploratorium. *IBM Research Report, RC 19279,* Yorktown Heights, NY.

Rosson, M.B. & Alpert, S.R. (1990). The cognitive consequences of object-oriented design. *Human-Computer Interaction, 5,* 345-379.

Rosson, M.B. & Carroll, J.M. (1993). Extending the task-artifact framework. In H.R. Hartson & D. Hix (Eds.), *Advances in Human-Computer Interaction, 4,* Norwood, NJ: Ablex, pp. 31-57.

Rosson, M.B., Carroll, J.M. & Bellamy, R.K.E. (1991). Smalltalk scaffolding: A case study of minimalist instruction *Proceedings of CHI'90.* New York: ACM, pp. 423-430.

Smith, D.C., Irby, C., Kimball, R., Verplank, B., and Harslem, E. (1982, April). Designing the Star user interface. *Byte, 7(4),* 242-282.

Wirfs-Brock, R., Wilkerson, B. & Wiener, L. (1990). *Designing object-oriented software.* Prentice Hall, Englewood-Cliffs, New Jersey.