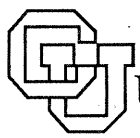
COMPUTER SUPPORT FOR SITUATED, PERSPECTIVAL, LINGUISTIC INTERPRETATION IN NON-ROUTINE DESIGN

Gerry Stahl

abridged Ph.D. Dissertation

CU-CS-689-93



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

COMPUTER SUPPORT FOR SITUATED, PERSPECTIVAL, LINGUISTIC INTERPRETATION IN NON-ROUTINE DESIGN

CU

Depa
Univer ...er

Boulder, (...509-0430 USA

Computer Support for Situated, Perspectival, Linguistic Interpretation in Non-Routine Design

by

GERRY STAHL

This is a technical report version (drastically abridged) of:

Stahl, G (93) *Interpretation in Design: The Problem of Tacit and Explicit Understanding in Computer Support of Cooperative Design*. PhD Dissertation. Department of Computer Science. University of Colorado at Boulder. December, 1993.

For an unabridged copy, ask for tech report CU-CS-688-93:

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, CO 80309-0430 USA
(303) 492-7514
email: dotty@cs.colorado.edu

For further information or comments, contact the author:

Gerry Stahl
2234 "C" Spruce Street
Boulder, CO 80302
(303) 444-2792
email: gerry@cs.colorado.edu

COMPUTER SUPPORT FOR SITUATED, PERSPECTIVAL, LINGUISTIC INTERPRETATION IN NON-ROUTINE DESIGN

Abstract

This is a drastically abridged version of a dissertation that analyzes the central role of interpretation in non-routine design. Based on this analysis, the dissertation constructs a theory of computer support for interpretation in cooperative design. The theory is grounded in studies of design and interpretation. It suggests three mechanisms of a software substrate for computer-based design environments, illustrated by application to a sample task of lunar habitat design.

This abridged version includes only an overview chapter, the three chapters that describe the implementation of the computer support mechanisms, and a brief concluding chapter. The motivation for the mechanisms has been left out of this version so that the practical suggestions for computer scientists will not be obscured by lengthy empirical, theoretical, and philosophical investigations. It is hoped that readers will be inspired to turn to the dissertation for the complete discussion. For this reason, consistency has been maintained with the dissertation by retaining the original chapter numbers.

The process by which designers transform their tacit preunderstanding into explicit knowledge is termed "interpretation" in the dissertation. Interpretation is necessary for solving design problems and for collaborating with other designers. Considerable explicit knowledge is thereby generated in the natural course of designing. Often this knowledge includes the most valuable information that can be presented to designers who revisit these design projects or who undertake similar projects in the future. If representations of this knowledge have been defined using computer-based design support systems, then the representations can be captured by these systems for the support of subsequent design work.

The disseratation presents a theory of computer support for interpretation in design in three stages: First, the role of interpretation in design is explored by reviewing descriptions of design by Alexander, Rittel, and Schön; by conducting a protocol analysis of lunar habitat design; and by applying Heidegger's philosophy of situated interpretation. Second, this analysis of interpretation is extended to define a theory of computer support. The features of this theory—support for the situated, perspectival, and linguistic characteristics of interpretation—are used to evaluate previous work on software design rationale systems. Third, design principles are discussed for HERMES, a prototype hypermedia substrate for building computer-based design environments to support interpretation in tasks like lunar habitat design. The hypermedia integrates a perspectives mechanism and an end-user language to capture and modify representations of the design situation, alternative perspectives on design tasks, and terminology for conceptualizing design issues. It is the third section—the discussion of the hypermedia substrate, perspectives mechanism, and end-user language—which is featured in the present abridged version.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. OVERVIEW

The following chapters present a *theory of computer support for innovative (non-routine), cooperative design* based on an *analysis of interpretation in design.* They will argue that the central impediment to computer support of innovative design is that designers make extensive use of *situated tacit understanding* while computers can only store and display *explicit representations of information.*

The process by which designers transform their tacit preunderstanding into explicit knowledge is termed *interpretation.* (See Part I for an analysis of interpretation in design.) Interpretation is central to the process of solving design problems and is part of the process of collaborating with other designers; the explicit knowledge that is generated by this interpretation is therefore a natural by-product of innovative, cooperative design. (See Part II for a theory of computer support based on this generated knowledge.) Representations of this knowledge defined using computer-based design support systems can be captured by these systems for the support of subsequent design work, including the maintenance and modification of the designed artifacts. (See Part III for details of a computer system for supporting interpretation in design.)

Chapter 1 provides a chapter-by-chapter overview of the dissertation. It discusses the claims, arguments, and themes that arise in each of the subsequent chapters, without going into the detail necessary to defend the claims, support the arguments, or work out the themes. Its purpose is to provide a readers' guide to the flow of the dissertation, motivating how one discussion leads into or provides the background for another. Section 1.1 offers a preliminary presentation of the central concept of interpretation, anticipating the analysis of this concept from various approaches in the dissertation. Each of the other sections provides an overview of a specific chapter.

1.1. UNDERSTANDING INTERPRETATION

To say that interpretation is central to innovative design is to stress that in order to design the designer must to some degree understand and be able to articulate the significance of the artifact being designed. This may include, for instance, understanding what is desired in a task specification, how possible composite parts of the artifact will function and interact, or how people can use the designed artifact. According to the analysis presented below, such understanding is possible for people but not for computers. People understand things because they are actively involved with them in the world. The significance of artifacts for a person is determined by the artifacts' relationships to other artifacts, activities, and people whose significance is already understood as part of the person's situation. Understanding combines

personal and socially shared perspectives on the world. All of this takes place primarily in *tacit* ways, i.e., unverbalized. However, one's tacit understanding of something can be partially articulated or expressed *explicitly* in spoken, written, or graphical language—either to deepen one's own understanding or to communicate with others.

Two aspects of the process of interpretation can be distinguished.

(1) There is a tacit *preunderstanding* based on previous background knowledge; items from this preunderstanding can be articulated explicitly.

(2) There is the possibility of revising that preunderstanding based on *discoveries* that are opened up by it.

That is, one can interpret something *as* something that one already knows about, or *as* a variation that differs from that in ways that are discovered as a result of the breaking of one's tacit expectations. Accordingly, interpretation in innovative design involves both human understanding of extensive background and a creative ability to revise one's understandings iteratively.

The analysis of interpretation developed below distinguishes three characteristics of interpretation: being situated, having a perspective, and using language.[1]

(a) *Being situated* means that what is to be interpreted is tacitly understood by virtue of its associations within an open-ended network of related artifacts, people, human purposes, and other concerns. All of these associations are themselves understood as part of one's background understanding of one's involvements.

(b) *Having a perspective* means that there is a focus on a certain aspect or that a specific approach is taken in interpreting something.

(c) *Using language* means that a particular vocabulary is available as part of a tradition that provides a conceptual framework for the interpretive task.

Each of these characteristics of interpretation is grounded in a form of preunderstanding that can be transformed through a corresponding phase of discovery. This two-dimensional structure is presented in Table 1-1.

---

[1] Note that the numbering scheme of 1, 2 and a, b, c is used consistently in this chapter as an organizing structure for the dissertation. It indicates correspondences among items listed; in particular, it indexes the way in which computer support features correspond to the characteristics of interpretation. Subsequent chapters are also organized around discussions of these characteristics and features, as emphasized in this Overview. Frequently, the numbering system is dropped and key terms are *italicized* as reminders that the discussion is focusing on (1) preunderstanding and (2) discovery, or on the (a) situated, (b) perspectival, and (c) linguistic character of understanding.

Table 1-1. The structure of human interpretation.

	(a) situated	(b) perspectival	(c) linguistic
(1)preunderstanding	expectations	focus	conceptualization
(2) discovery	surprises	deliberations	refinements

In articulating tacit understanding, interpretation both discloses inherent implications and discovers unanticipated consequences in the situation. Through interpretation, designers might (a) try to externalize their expectations about a design situation by drawing a sketch and then discover surprises when they explore the sketch. Similarly, they might need to revise their understanding as a result of (b) shifting their focus on a problem and deliberating from alternative perspectives or (c) changing the way they conceptualize an issue and refining the definitions of terms in their language.

The structure of human interpretation carries over to design. The design process is a cycle or spiral of interpretation: (1) some item of the initial preunderstanding—the grasp of the design situation, the perspective for viewing, the language for conceptualizing—is made explicit, reflected upon, and further articulated in new design decisions. (2) This leads to the discovery of unanticipated consequences or contingencies and a new understanding that requires revisions to the understanding of the design problem, its viewpoint, or terminology. (1′) The new understanding then becomes re-submerged into a modified tacit understanding that forms the starting-point for the next iteration of interpretation and design.

The analysis of interpretation in design motivates a theory of computer support. According to this theory, computer support for interpretation in innovative design differs from autonomous software systems for routine design by focusing on supporting or augmenting human activities rather than automating them, because only people have the understanding and creativity required for interpretation. This computer support takes two general forms in order to support the two phases of interpretation:

1. It provides access to a wealth of information that might be useful as a basis for interpreting new design tasks. This information for *reuse* is culled primarily from previous design experience, and includes (a) partial representations[2] of design situations, (b) alternative ways of considering tasks, and (c) terminology helpful for conceptualizing problems.

---

[2] Note that the computer manipulates symbolic *representations* of things in the situation, whereas the designer has a situated *understanding* of the things. According to Heidegger's philosophy, representations are explicit forms of information that only arise under certain conditions and on the basis of people's normally tacit understanding of things within the context of meaningful involvements. In Chapter 4, the *situation* is defined as this context of meaningful involvements, which provides a precondition for meaningful representations.

2. It facilitates the revision of stored information so designers can tailor existing representations to novel problems and can capture innovative designs to extend the computerized knowledge-base and to communicate ideas to collaborators. This *plasticity* of representation—the ability to mold, form, adapt, alter, or modify the representations—applies to all design knowledge, including (a), (b), and (c) of point (1).

The proposed theory of computer support suggests an approach to building software systems that has been prototyped in a system named HERMES. HERMES is a substrate for building design environments to support interpretation in innovative design. Motivated by the analysis of interpretation, HERMES provides the following features to support reuse and plasticity of representations of each of the three characteristics of interpretation, being situated, having perspectives, and using language (see Table 1-2):

a-1. A persistent hypermedia network for storing partial representations of design situations and for browsing among them.

a-2. Efficient mechanisms for revising the representations (multimedia nodes) and modifying their associations (links).

b-1. A perspectives mechanism that organizes specialized or personal ways of filtering out information of interest

b-2. Procedures for switching perspectives or for creating new ones by merging existing perspectives and modifying their inherited contents in the new one.

c-1. An end-user language that provides useful domain terms, rules for critiquing designs, and queries for displaying stored information.

c-2. The ability to modify or generate new terms, critic rules, and queries or to use the language for defining computations.

Table 1-2. Computer-based mechanisms to support interpretation in design.

	(a) situated	(b) perspectival	(c) linguistic
(1) reuse	hypermedia network	perspectives mechanism	end-user language
(2) plasticity	revising representations	merging multiple perspectives	defining new expressions

Although computers cannot understand things the way people do, they can serve as a computational medium to support people's interpretive processes. The computer support mechanisms listed in Table 1-2 can augment cooperative design in a number of ways, including:

a-1 As a long-term memory or repository for information that was created in past designing and is now available to be shared by designers using the repository.

a-2 As an external memory for representing and revising designs to see how alternative variations appear.

b-1 As a retrieval mechanism for organizing and managing design knowledge and filtering through just what is relevant.

b-2 As a display mechanism to define new personal and shared views of designs.

c-1 As a linguistic medium for expressing knowledge in a canonical form that can be used for computations by the software.

c-2 As a communication medium to generate new knowledge to be shared with others.

A comparison of Table 1-1 and Table 1-2 shows that *the mechanisms of computer support are based on the structure of unaided human interpretation.* The computer support is intended to extend the power of designers to operate under conditions of "information overload," in which it is becoming increasingly difficult to work effectively without the use of computers.

Computer support will inevitably change the practices of collaborative design. This need not be considered harmful—particularly in cases where traditional procedures have become inadequate—if important factors like the characteristics of interpretation are preserved and adequately supported. Computational media have the potential for changing the activities of professionals even more than the media of written language did in the past, because of significant opportunities for the computer to play a computationally active role in organizing, analyzing, displaying, and communicating the information. The ways in which design tasks are accomplished will change dramatically as the computer augments and supports designers to do many of the same tasks they have done unaided in the past, like designing and modifying artifacts.

The proposed theory of computer support for interpretation in design goes to the root of the problem of tacit and explicit understanding. Designers approach their task with a background of skills, know-how, and experience that they are generally not aware of as they design but that is a necessary precondition of their work as trained professionals. For instance, architects have the ability to understand the situations people might face in the buildings they design, they know how to sketch and visualize relationships from the perspectives of different concerns, and they move freely between various frameworks or traditions that provide meaningful languages or metaphors for expressing their insights. Computers have no such tacit preunderstanding; they can only retrieve and manipulate what people have already formulated in explicit propositions or drawings. People and computers are not analogous processors of information. If computers are to support human cognition effectively, then these differences must be understood and taken into account.

By describing the transformation of tacit to explicit human understanding, the *analysis of interpretation* not only clarifies how human cognition differs from computer information processing, but also suggests how computers can support the

way people think. Philosophically, the analysis of interpretation provides the key to a theory of people-centered computer support. Technically, the analysis enumerates the functionality needed for computer support of interpretation in design. Practically, it points out that the process of innovative design and the requirements of collaboration generate both the need for computer support and the sources of explicit knowledge that make it possible. For instance, large, multi-person design projects often confront the problem of information overload, where computers are required to manage volumes of technical knowledge. At the same time, these cooperative design processes naturally articulate much explicit knowledge that could prove useful in subsequent computer-supported design work.

The theory of computer support for interpretation in design is presented in three Parts: in Part I, Chapters 2, 3, and 4 develop the analysis of interpretation in design. In Part II, Chapters 5, 6, and 7 draw the consequences of the problem of tacit and explicit understanding for computer support. In Part III, Chapters 8, 9, and 10 describe how the technical features of the HERMES substrate support interpretation in collaborative design.

The analysis of interpretation is developed by reviewing insightful descriptions of design by design methodologists Alexander, Rittel, and Schön (Chapter 2). Characteristics of design enumerated in that review are then used to guide a study of transcripts of a design session involving a task of lunar habitat design (Chapter 3). The design process—as characterized by design methodology and as illustrated with lunar habitat design—is then conceptualized as a process of interpretation by using Heidegger's philosophy of interpretation (Chapter 4).

The consequences for computer-based design systems are drawn by further developing the analysis of tacit and explicit understanding in design (Chapter 5), and extending it to include a theory of the computer support of interpretation (Chapter 6). This theory is applied to evaluate traditions of software design environments and design rationale systems; useful techniques in these previous systems are explored and their limitations noted (Chapter 7).

The technical description of computer support for cooperative design describes the central functionality of HERMES. It has a hypermedia knowledge-base to support (a) the representation of design situations (Chapter 8). A virtual copying mechanism provides (b) perspectives on design knowledge (Chapter 9). An end-user language is used for (c) articulating formerly tacit understandings in explicit language (Chapter 10)

The order of presentation in the dissertation corresponds to the process of interpretation. First, in the Introduction and Part I a *preunderstanding* is sketched to provide a starting point for interpreting the problem of computer support for innovative design. A review of design methodology provides a *perspective* from which to understand design, formed by merging the perspectives of the three design methodologists. A lunar habitat design project provides a concrete design *situation* for grounding the developing understanding of design. Heidegger's philosophy provides a *language* and conceptual framework for talking about interpretation in

design. Second, in Part II this preunderstanding is used to explore possibilities for computer support that are opened up by the preunderstanding. This is accomplished partially by drawing out the theoretical consequences in order to extend the analysis of interpretation in design to include a theory of its computer support. It is further accomplished by discovering the achievements and the limitations of previous software systems in providing the kind of support for design that is called for. Third, in Part III the arrived at understanding allows for a discussion of the HERMES system as an explicit illustration of possible responses to the problem of supporting interpretation in design .

Predecessor systems to HERMES (principally JANUS and PHIDIAS) were already headed in the direction that HERMES adopts. Discussions of these earlier design environments made frequent reference to Alexander, Rittel, and Schön, for instance, and insisted on supporting rather than automating design. The theory of computer support for interpretation in design presented in this dissertation extends this approach theoretically and practically. Its focus on interpretation *situates* its people-centered approach unambiguously in an analysis of human understanding. By providing a coherent *perspective* for viewing systems to support design, the theory suggests principled extensions to the functionality of design environments, such as those incorporated in the HERMES substrate. It provides an explicit *language* as a basis for a coherent conceptual framework.

Each section in the remainder of Chapter 1 provides an overview of a chapter of the dissertation. The first three sections each provide an argument for interpreting design as a process of interpretation. The other sections draw the implications of this argument for the computer support of design. The three characteristics of interpretation run through all the chapters. Table 1-3 shows the correspondences between the central themes in the different chapters. *These correspondences link the theoretical analysis of interpretation to the operational mechanisms that provide computer support for these characteristics.* For the sake of simplicity, the table does not indicate that each of the entries involves both reuse of past information and creative modification, however this is true both for the three characteristics of interpretation and for their corresponding software mechanisms, as already shown in Tables 1-1 and 1-2.

Table 1-3. Correspondences among the chapters.
Note that the three mechanisms of HERMES in Chapters 8, 9, and 10 correspond to the three characteristics of interpretation that permeate and structure the dissertation.

Chapter	Theme	(a)	(b)	(c)
1	interpretation	situated	perspectival	linguistic
2	methodology	Alexander	Rittel	Schön
3	lunar habitat	privacy conflict	privacy concern	privacy gradient
4	preunderstanding	prepossession	preview	preconception
5, 6	computer support	represent situation	have perspectives	make use of language
7	previous systems	JANUS	PIE	PHIDIAS
8, 9, 10	HERMES software substrate	hypermedia network	perspectives mechanism	end-user language

1.2. THE METHODOLOGY OF DESIGN

A central claim of this dissertation is that design can be viewed as fundamentally a process of interpretation. In this interpretive process, elements of the designer's tacit background preunderstanding are made explicit. The first evidence in support of this claim is a review of the writings of three influential design methodologists. It is argued that their diverse but complementary descriptions of the design process highlight characteristics of what is here called interpretation. They recognize the importance of both tacit understanding and explicit representations, as well as the iterative movement between them. Among the three writers, the dimensions of (a) the situation, (b) perspectives, and (c) language are all stressed. Furthermore, each of these dimensions is recognized to entail both (1) traditions of past knowledge to start from and (2) an ability to revise that knowledge to promote and grasp innovation.

Alexander (1964) pioneered the use of computers for designing. He used them to compute diagrams or patterns that decomposed the structural dependencies of a given problem into relatively independent substructures. In this way, he developed *understandings of the design situation* for solving a task based on an analysis of the unique design situation.

Later, Alexander (1977) assembled 253 patterns that he considered useful for architectural design, based on an extensive study of successful past designs. These patterns were to be *reused* and *modified* to form personal pattern languages for expressing the individual perspectives of different designers. They were schematic enough to be adapted to a broad range of specific design situations.

Alexander felt that the design profession necessarily made *explicit* the understanding that was "unselfconscious" in traditional cultures in which everyone

designed their own artifacts. His structures and patterns were meant to be tools for explicitly representing design situations for "self-conscious" design. However, he always also recognized the need for *tacit* or intuitive understanding as a basis for good design.

For Rittel (1973), the heart of design was the deliberation of issues from *multiple perspectives*. In a collaborative effort, each participant may bring different personal interests, value systems, and political commitments to the task. Also, people with different technical specialties or professional skills may contribute to a design. These are actually different kinds of "perspectives." The theory of computer support in Chapter 6 distinguishes three classes of perspectives that need to be supported:

* personal or group perspectives
* technical specialties (e.g., plumbing)
* domain traditions (e.g., residential kitchens)

However, they all provide the same function of determining what issues will be addressed, what alternatives will be considered, and what criteria will be applied. Because they all determine the organization or relevance of information in a similar way, they can be discussed as one kind of determinant of interpretation and can be operationalized and supported with one software mechanism (a perspectives mechanism).

The important thing for Rittel was not the subjective character of interpretation deriving from its basis in personal perspectives, but the way in which deliberation among perspectives can lead to innovative solutions that would not have arisen without such interaction. Deliberation is an interpretive process in which understanding of the problem situation and of the design solution emerges gradually as a product of iterative revisions subject to critical argument from the various perspectives. This can take place for an individual designer as well if the designer consecutively adopts different perspectives on the issues. Rittel foresaw computer support for this. His idea of using computers to keep track of the various issues at stake and alternative positions on those issues led to the creation of issue-based information systems.

Schön (1983) argued that designers constantly shift perspectives on a problem by bringing various professionally trained tacit skills to bear, such as visual perception, graphical sketching, and vicarious simulation. The designer's intuitive appreciations shape the problem by forming a subsidiary background awareness of the design task's patterns, materials, and relationships. By then experimenting with tentative design moves within this tacitly understood situation, the designer discovers consequences and makes aspects of the problem explicit. As this is done, certain features of the situation come into focus and can be named or characterized in a *language*. When focus subsequently shifts, what has been made explicit may slip back into an understanding that is again tacit, but is now more developed.

Schön (1992) provided empirical evidence for the roles of the situation, perspectives for viewing, and conceptual frameworks in the iterative process of

interpretation in design. His experiments showed how the designer uses tacit skills and preunderstandings to uncover unanticipated discoveries, to reflect upon them, and to develop new understandings, new perspectives, and new articulations of the evolving design situation.

1.3. THE EXAMPLE OF LUNAR HABITAT DESIGN

A second argument for understanding design as a process of interpretation is presented in Chapter 3. Here, a protocol analysis of designers collaborating shows that most of what went on was interpretation.

As part of the research for this dissertation, a study was undertaken of lunar habitat design. Lunar habitat design is a task that is not well understood compared to many other, more mundane design tasks. It is not a routine matter that can be done according to well-formulated rules or by applying available template solutions. Furthermore, it is representative of a broad range of high-tech design tasks. Such tasks typically involve extensive technical knowledge. They seem to call for computer support.

The volume of information available to people is increasing rapidly. For many professionals this "information overload" means that the execution of their jobs requires taking into account far more information than they can possibly keep in mind. The lunar habitat designers here provide a prime illustration of such professionals. In working on their high-tech design tasks, they must take into account architectural knowledge, ergonomics, space science, NASA regulations, and lessons learned in past lunar missions. These designers turn to computers for help with their complex, technical problems. That is why a group of lunar habitat designers initiated the software development effort that led to this dissertation.

Providing the computer support needed by lunar habitat designers is not straight-forward. Designers need to be able to consider wide varieties of experience, professional know-how, technical concerns, and previous solutions that are relevant to their current tasks. However, the problem is not so much one of storing large amounts of information as one of deciding what information to retain that might be relevant to novel future tasks and how to present it to designers in formats that support their mode of work. It is a problem of how to manage the information and present it so that it can usefully serve the design process. The necessary decisions must be made by the designers who are involved with these tasks. Computer techniques for capture and display of information must be under the control of people engaged in the interpretation of the information.

As part of the effort at developing computer support for lunar habitat designers, thirty hours of design sessions were videotaped and analyzed. The designers were asked to design a 23 foot long by 14 foot wide cylindrical habitat to accommodate four astronauts for 45 days on the moon. A protocol analysis of segments of the video recording was conducted.

The analysis of the videotape of the designers' activities shows that design time is dominated by processes of interpretation, i.e., the explication of previously tacit knowledge in response to *discoveries* of surprises. As part of the interpreting by the designers, graphical representations were developed for describing pivotal features of the design *situation* that had not been included in the original specification; *perspectives* were created for looking at the task in different ways; and *language* terminology was defined for explicitly naming, describing, and communicating formerly tacit understandings. The definitions of the situated understanding, perspectives, and language continually evolved as part of the design process in an effort to achieve an adequate understanding of the design task and the evolving artifact.

The nature of interpretation and the three dimensions of preunderstanding are illustrated in Chapter 3 with an example from the lunar habitat design sessions. This designing primarily consisted of sketching and discussion that explicated visual and conceptual expressions used for understanding, explaining, and guiding the emerging design. The example analyzed has to do with the tacit notion of privacy and a default perspective on bathroom design related to this notion. The following paragraph briefly summarizes the example.

The designers felt that a careful balance of public and private space would be essential given the long-term isolation in the habitat. This is an important concern that receives limited treatment in official NASA design guidelines. An early design decision proposed that there be private crew compartments for each astronaut. An initial sketch revealed problems with adjacencies of public and private areas, leading to an interpretation of privacy as determining a "gradient" along the habitat from quiet sleep quarters to a public activity area. In the process, the conventional American idea of a bathroom was subjected to critical reflection when it was realized that the placement of the toilet and that of the shower were subject to different sets of constraints based on life in the habitat. The tacit American assumption of the location of the toilet and shower together was made explicit by comparing it to alternative European perspectives. The revised conception permitting a separation of the toilet from the shower facilitated a major design reorganization.

In this way, a traditional conception of "private space" as a place for one person to get away was made explicit and explored within graphical representations of the design situation. As part of the designing process, this concept was revised into a notion of "degrees of privacy", which facilitated the design process. The failure of the NASA guidelines to provide significant guidance despite a clear recognition by NASA of the importance of habitability and privacy considerations raises the problem of how to represent effectively notions like privacy that are ordinarily tacit. This problem provides the central test case for this dissertation. In Chapter 9, a scenario shows how designers using HERMES can define interpretive critics to evaluate the distribution of public and private spaces in a lunar habitat. A detailed analysis of how these critics are defined in the HERMES language is then presented in Chapter 10.

In this and other examples, the designers needed to revise their representations to enhance their understanding of the problem situation. They went from looking at privacy as a matter of individual space to reconceptualizing the whole interior space as a continuum of private to public areas. The conventional American notion of a bathroom was compared with other cultural models and broken down into separable functions that related differently to habitat usage patterns. The new views resulted from argumentative discussions motivated by design constraints—primarily spatial limitations and psychological factors of confinement. In these discussions, various perspectives were applied to the problem, suggesting new possibilities and considerations. Through discussion, the individual perspectives merged and novel solutions emerged. In the process, previously tacit features of the design became explicit by being named and described in the language that developed. For instance, the fact that quiet activities were being grouped toward one end of the habitat design and interactive ones at the other became a topic of conversation at one point and the terminology of a "privacy gradient" was proposed to clarify this emergent pattern.

1.4. THE ANALYSIS OF SITUATED INTERPRETATION

Chapter 4 presents a third argument for focusing on interpretation in design: computer support of innovative design should be based primarily on an analysis of human understanding. As Norman (1993) puts it, "Without someone to interpret them, cognitive artifacts [like computer support systems] have no function. That means that if they are to work properly, they must be designed with consideration of the workings of human cognition." The philosophy of interpretation provides just such a consideration.

This contrasts with many previous approaches to computerization of design and to artificial intelligence, which lean toward theories on the natural science model (e.g., mathematical physics), like information theory and predicate logic formalisms. Human sciences (e.g., cultural anthropology or non-behaviorist psychology), however, necessarily center on human interpretation because their subject matter is defined by what people consider to be important and by how people construe things. As one moves from routine design to highly innovative tasks, the distribution of roles in the human-computer relationship shifts more onto the people involved, and it becomes increasingly important to take into account their cognitive functioning.

An initial framework for clarifying the respective roles for computers and people in tasks like lunar habitat design is suggested by theories of *situated cognition.* Several influential recent books[3] argue that human cognition is very different from computer manipulations of formal symbol systems. The differences imply that people

---

[3] A series of publications in the last decade has, in effect, defined an approach to cognitive science and to the theory of computer support for design that goes by the name "situated cognition." These include Schön (1983), Winograd & Flores (1986), Suchman (1987), Ehn (1988), and Dreyfus (1991).

need to retain control of the processes of non-routine design because these processes rely heavily upon what might be called situated interpretation. Computers can provide valuable computational, visualization, and external memory aids for the designers by supporting such interpretation in design.

Situated interpretation, as used here, refers to a view of human understanding as taking place within tacit contexts of background skills, human concerns, and linguistic traditions that provide its grounding. Interpretation is not just a function of a disinterested rational mind, but relies on the interpreting person or people being actively involved with the situation, which includes the artifact being interpreted and supplies the basis for that artifact's significance. (See Heidegger's fuller definition of situation below and in Chapter 4.)

Situated cognition theory disputes the prevalent view based on the natural sciences model that all human cognition is based on explicit mental representations such as goals and plans. Winograd and Flores (1986) hold that "experts do not need to have formalized representations in order to act" (p.99). Although manipulation of such representations is often useful, there is a background of preunderstanding that cannot be fully formalized as explicit symbolic representations subject to rule-governed manipulation. This tacit preunderstanding even underlies people's ability to understand representations when they do make use of them. Suchman (1987) concurs that goals and plans are secondary phenomena in human behavior, usually arising only after action has been initiated: "When situated action becomes in some way problematic, rules and procedures are explicated for purposes of deliberation and the action, which is otherwise neither rule-based nor procedural, is then made accountable to them" (p.54).

This is not to denigrate conceptual reasoning and rational planning. Rather, it is to point out that the manipulation of formal representations alone cannot provide a complete model of human understanding. Rational thought is an advanced form of cognition that distinguishes humans from other life forms. Accordingly, an evolutionary theorist of consciousness like Donald (1991) traces the development of symbolic thought from earlier developmental stages of tacit knowing (e.g., episodic and mimetic memory-based cognition). He shows how these earlier levels persist in rational human thought as the necessary foundation for advanced developments, including language, writing, and computer usage.

Philosophers like Wittgenstein (1953), Polanyi (1962), Searle (1980), and Dreyfus (1991) suggest a variety of reasons why tacit preunderstanding cannot be fully formalized as data for computation. It is too vast: background knowledge includes bodily skills and social practices that result from immense histories of life experience. We are unaware of much of it: these skills and practices are generally transparent to us. It must be tacit to function: the examples of biking, swimming or playing a musical instrument suggest that procedural knowledge at least gets in the way of skilled action if it is explicit. More generally, tacit knowledge is a precondition for explicit knowing: we cannot formulate, understand, or use explicit knowledge except on the basis of necessarily tacit preunderstandings.

The philosophical foundations of situated cognition theory were laid out by Heidegger (1927), the first to point out the role of tacit preunderstanding and to elaborate its implications. For Heidegger, we are always knowledgeably embedded in our world; things of concern in our situations are already meaningful before we engage in explicit cognitive activity. We know how to behave without having to think about it. For instance, an architect designing a lunar habitat knows how to lift a pencil and sketch a line or how to look at a drawing and see the rough relationships of various spaces pictured there. The architect understands what it is to be a designer, to critique a drawing, to imagine being a person walking through the spaces of a floor plan. Such tacit, background skills or preunderstandings of the design situation are necessary prerequisites for being able to design an artifact.

Heidegger defines the *situation* as a person's interpretive context—including the physical surroundings, the available tools, the circumstances surrounding the task at hand, the person's own personal or professional aims, and social or cultural relations. The situation constitutes a network of significance in terms of which each part of the situation is already meaningful. That is, the person has tacit knowledge of the situation as a whole; if something becomes a focus, it is perceived as already understood and its meaning is defined by its relations within the situation. Everything is tacitly understood in its relations to other things and to the whole.

According to situated cognition in contrast to rationalist views, to an architect a rectangular arrangement of lines on a piece of paper is not first perceived as meaningless lines that need defining attributes (to be determined by subsequent rational thought). Rather, given the design situation, it is already understood as (say) a sleep compartment for astronauts. The sleep compartment is implicitly defined as such by the design task, the shared intentions of the design team, the other elements of the design, the availability of tools for revising the drawing, the sense of space conveyed by the design, the prevailing NASA terminology. This network of significance is background knowledge that allows the architect to think about features of the design, to make plans for changes, and to discover problems or opportunities in the evolving design. At any given moment, the background is already tacitly understood and does not need to be an object of rational thought manipulating symbolic representations.

At some point the architect might realize that the sleep compartment is too close to some source of potential noise, like the flushing of the toilet. This physical adjacency would come into focus as an explicit concern against the background of relationships of the preunderstood situation. Whereas a common sense view might claim that the sleep compartment and toilet were already immediately and objectively present, and that therefore their adjacency was always there by logical implication, Heidegger proposes a more complex reality in which things are ordinarily hidden from explicit concern. In various ways, they can become uncovered and discovered, only to re-submerge soon into the background as our focus moves on.

In this way, our knowledge of the world primarily consists neither in mental models that represent reality nor in an unmediated and objective access to objects.

Rather, our understanding of things presupposes a tacit preunderstanding of our situation. This is analogous to the view of Kuhn (1962), who argues that scientists' experimental observations presuppose their tacit ability to use their experimental equipment and to apply their frameworks of hypotheses and theory. Only by being already situated in our world can we discover things and construct meaningful representations of them by building upon, explicating, and exploring our tacit preunderstanding. Situated cognition is not a simplistic theory that claims our knowledge lies in our physical environment like words on a sign post: it is a sophisticated philosophy of interpretation.

According to the philosophy of situated interpretation, human understanding develops through interpretive explication involving both (1) preunderstanding and (2) explorative discovery of the situation. In Heidegger's analysis, interpretation provides the path from tacit, uncritical preunderstandings to reflection, refinement, and creativity. The structure of this process of interpretation reflects the inextricable coupling of the interpreter with the situation, i.e., of people with their worlds. One's situation is not reducible to one's preunderstanding of it; it offers untold surprises, which may call for reflection, but which can only be discovered and comprehended thanks to one's preunderstanding. Often, these surprise occasions signal *breakdowns* in a person's skillful, transparent behavior, although one can also make unexpected discoveries in the situation through conversation, exploration, or external events.

A discovery breaks out of the preunderstood situation because it violates or goes beyond the network of tacit meanings that make up the preunderstanding of the situation. To understand what one has discovered, one must explicitly *interpret* it *as* something, as having a certain significance, as somehow fitting into an understood background. Then it can merge into one's comprehension of the meaningful situation and become part of the new background. Interpretation of "something *as* something" requires a reinterpretation of the situated context if the discovery does not fit into the previously understood situation.

For instance, the lunar habitat designers discovered problems in their early sketches (their representations of the design situation) that they interpreted as issues of privacy. Although they had created the sketches themselves, they were completely surprised to *discover* certain conflicts among the functions of adjacent components, like the sleep compartments and the toilet. The discoveries could only occur because of their *situated* understanding represented in the drawings. The designers paused in their sketching to discuss the new issues. First they debated the matter from various *perspectives*: experiences of previous space missions, cultural variations in bathroom designs, technical acoustical considerations. Then they considered alternative conceptions of privacy, gradually developing a shared *vocabulary* that guided their revisions and became part of their interpretation of their task. They reinterpreted their understanding of privacy and articulated their new view using the terminology of a privacy gradient.

These themes of being situated, having perspectives, and using explicit language correspond to the three-fold structure of preunderstanding in Heidegger's philosophy. He articulates the pre-conditions of interpretation as: (a) *pre-possession*

of the situation as a network of preunderstood significance; (b) *pre-view* or expectations that things in the world are structured in certain ways; and (c) *pre-conception*, a preliminary language for expressing and communicating. In other words, interpretation never starts from scratch or from an arbitrary assignment of representations, but is an evolving of tentative prejudices and anticipations. (1) One necessarily starts with a preunderstanding that has been handed down from one's past experiences and inherited traditions. (2) The interpretive process allows one to reflect upon this preunderstanding methodically and to refine new meanings, viewpoints, and terminologies for understanding things more appropriately.

The analysis of interpretation based on Heidegger's philosophy stresses the role of tacit preunderstanding as the basis for all understanding. Preunderstanding consists primarily of the characteristics of prepossession, preview, and preconception. It also implicitly incorporates the structure of "something *as* something." Through interpretation, this preunderstanding is articulated. The resultant explicit understanding can be externalized in discourse. This can be taken further through the methodologies of science to codify knowledge. Each stage in this process preserves the original structure of the preunderstanding. It is because of this structure that metaphors, speech acts, and scientific propositions have the structure they do of something *as* something, something *is* some predicate, or something *has* some attribute.

The process of explication through interpretation forms the basis for computer support by transforming tacit understanding into increasingly explicit forms that can eventually be captured in computer-based systems.

1.5. TACIT AND EXPLICIT KNOWLEDGE IN DESIGN

Heidegger's analysis of interpretation must be *applied* to the realm of design before it can be used as the basis for a theory of computer support of design. Three general problems must be considered:

* First, although his philosophy is presented in a very general way, Heidegger's examples come primarily from people's relations to physical things in the world, rather than to imagined artifacts that they are designing.
* Second, he stresses that things are always understood on the basis of preunderstandings we already have, which makes it hard to say how innovative design ideas are understood.
* Third, of course, Heidegger (writing in the mid-1920's) did not address the issue of computer representations as a form of explicit knowledge.

Chapter 5 accomplishes the application of Heidegger's analysis to design in three steps.

* First, it shows that Heidegger's philosophy can be extended naturally to design.

* Second, it discusses the problem of application, which addresses the issue of how previously captured knowledge can be adapted to innovative new designs.
* Third, it spells out a taxonomy of transformations of tacit understanding to explicit knowledge adequate for providing a basis for computer representations of normally tacit design knowledge.

Heidegger's concept of the situation transfers well to design. As the network of relationships in the understood world, the situation corresponds closely to the set of constraints and adjacencies that are of concern in design and that are sometimes even represented explicitly in design documents. Heidegger's definition of interpretation as the explication of tacit understanding, involving discoveries, is also applicable to the process of design, in which relationships are explored and discoveries made. Consideration of interpretation in the design context clarifies how breakdowns in action require repair to the tacit underlying understanding of the situation. Although Heidegger's examples focus on the individual, his recognition of the social dimension and the importance of shared understanding allows his analysis to be extended to design, which is largely collaborative.

Heidegger's philosophy occupies an important position in the twentieth century recognition that reality is socially constructed. People have access to their world (intentionality) because the world is in many ways a human, social creation. Of course, reality also has an immanence which can contradict our expectations and present surprises, just as we can make discoveries in designs of our own creating. The point is that an understanding of the world or of innovative designs requires the situated interpretation of a person: it cannot be reduced to a set of rules or a computer algorithm. The same goes for knowledge, which encapsulates understanding. To apply knowledge from past cases to a new design, one must apply it within a situated, perspectival, linguistic understanding. That means that computer software for designing should be people-centered and should support the situated, perspectival, linguistic character of human understanding.

Chapter 5 defines *tacit* as being expressed without words or speech, and *explicit* as being fully revealed or verbally expressed. It defines a taxonomy of forms of information along the continuum between these extremes and describes the transformations from one form to the next based on Heidegger's analysis. These transformations are summarized in Figure 1-1. Each transformation involves a reinterpretation of the informational content in a new medium. With that comes a gain in precision balanced by a loss of grounding. As a result of the increased clarity and the change of form, new discoveries are made about the content of the information.

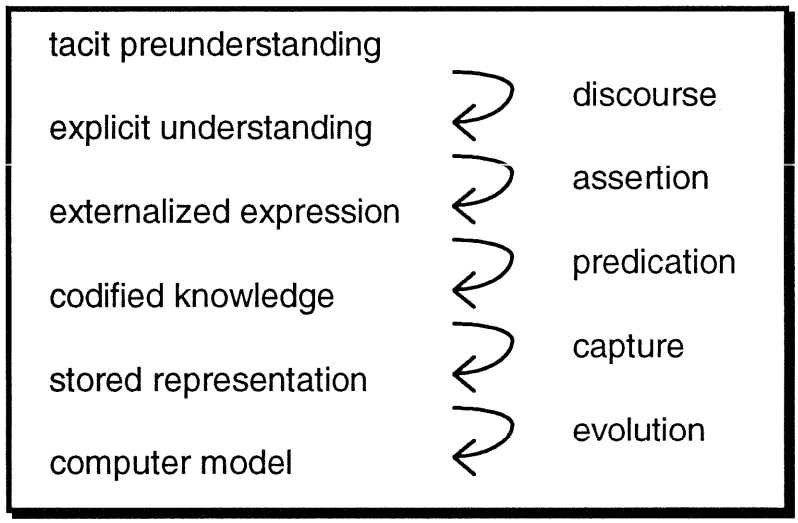


Figure 1-1. Transformations of tacit to explicit information.
The left-hand column lists consecutive forms of information. The right-hand column names the transformation processes from one form to another.

Heidegger uses the term *discourse* for the fundamental shift to putting one's understanding into words, even if the words are not yet *asserted* in speech to be shared with someone. After tacit preunderstanding is articulated in discourse as explicit understanding, this understanding can then be asserted and externalized in spoken or written language (such as documented design rationale). Such knowledge can be further codified in accordance with formal procedures (e.g., scientific methods). These are important transformations for creating widely shared knowledge. The movement from externalized to codified information can go from informal to formal (i.e., capable of being processed by computer). Shipman (1993) discusses this stage of formalization and methods for supporting it within computer-based design environments. This is relevant to the further stages of articulation, which involve computers: capture of the information in computer representations and modification of these representations to adapt them to new requirements. The theory of computer support for design proposed in Chapter 6 suggests that all stages of information articulation can take advantage of computer support. If designing takes place within a computer-based design environment, then designers can use and modify computer representations to support the design process from the start. As Reeves (1993) recommends, the design environment can serve as a medium of communication to support collaboration. In the process, design information can be captured automatically without becoming a burdensome task to be done in retrospect.

1.6. CONSEQUENCES FOR A THEORY OF COMPUTER SUPPORT

The ideas of situated cognition and Heidegger's philosophy of interpretation stress how different human understanding is from computer manipulations of symbols. These analyses suggest a *people-centered approach* of augmenting, rather than

automating, human intelligence. According to this view, software can at best provide computer representations for people to interpret based on their tacit understanding of what is being represented. Representations used in computer programs must be carefully structured by people who understand the task being handled thoroughly, because the computer itself simply follows the rules it has been given for manipulating symbols, with no notion of what they represent. People (e.g., software designers or software users) who understand the domain must codify their knowledge into software rules sufficiently to make the computer algorithms generate results that, when interpreted by people, will be the desired results. Only if a domain can be strictly delimited and its associated knowledge exhaustively reduced to rules, can it be completely represented in advance (by the software designers) so that tasks in the domain can be automated.

Many tasks like lunar habitat design that call for computer support do not belong to well-defined domains with fully catalogued and formalized knowledge bases. These tasks may require (a) exploration of possibilities never before considered, (b) assumption of creative viewpoints, or (c) formulation of innovative concepts. Software to support designers in such tasks should provide facilities for the designers themselves (as the software users) to create new representations and to flexibly modify old representations. As the discussion of Alexander emphasizes, the ability *to develop appropriate understandings of the situation dynamically* is critical to innovative design. Because they capture understandings that evolve through processes of interpretation, representations need to be modifiable during the design process itself and cannot adequately be anticipated in advance or provided once and for all. Lunar habitat design is an example of an exploratory domain in two senses: (1) it is a new domain with relatively little in the way of accepted conventional knowledge, and (2) it involves continual innovation to meet novel, over-constrained, politically sensitive mission specifications.

The assumption of the existence (even in principle) of an objective, coherent body of domain knowledge that can be used without being reinterpreted in new situations and from different perspectives is misleading. As Rittel says, non-routine design is an argumentative process involving the interplay of unlimited perspectives, reflecting differing and potentially conflicting technical concerns, personal idiosyncrasies, and political interests. Rather than trying to supply all knowledge in advance, software to support this type of design should capture alternative deliberations on important issues as they arise and document specific solutions. Then, these can be available to support interpretive deliberations. Furthermore, because all design knowledge is relative to perspectives, the computer should be used *to define a network of over-lapping perspectives* with which to organize issues, rationale, sketches, component parts, and terminology to reflect the different viewpoints designers adopt. That will facilitate the retrieval of information relevant to a particular interpretive stance.

As Schön emphasizes, design relies on moving from tacit skills to explicit conceptualizations, and on the ability to reformulate the implications in linguistic expressions. Additionally, design work is inherently communicative and increasingly

collaborative, with high-tech designs requiring successive teams of designers, implementors, and maintainers. Software to support collaborative design should provide *a language facility for designers to develop a sharable vocabulary* for expressing their ideas, for communicating them to future collaborators, and for formally representing them within computer-executable software. An end-user language that provides an extensible domain vocabulary, is usable by non-programmers, and encourages reuse and modification could help provide support for designers trying to express their interpretations..

Heidegger's analysis of interpretation says that new interpretations are based on preunderstandings developed in the past or handed down by tradition. In this sense, it is likely that the information designers need most when they reflect on problems may have previously been made explicit at some moment of interpretation during past designing. Accordingly, one promising strategy for accumulating a useful knowledge base is to have the software capture knowledge that becomes explicit while the software is being used. As successive lunar habitats are designed on a system, issues and alternative deliberations can accumulate in its repository of design rationale; new perspectives can be defined with their own modified representations, terminology, and critic rules; and the language can be expanded to include more domain vocabulary, conditional expressions, and query formulations.

This is an evolutionary, bootstrap approach, where the software can not only support individual design projects, but simultaneously facilitate the accumulation of expertise and viewpoints in open-ended, exploratory domains. This means that the software should support designers in formalizing their knowledge when it becomes explicit. The software should reward its users for increasing the computer knowledge base by performing useful tasks with the new information, like providing documentation, communicating rationale, and facilitating reuse and modification of relevant knowledge.

The theory suggested by the analysis of interpretation in design is diagrammed in Figure 1-2. As the cycle of interpretation proceeds, driven by the needs of designing and collaboration, explicit knowledge that is generated can be captured by the computer support system. The computer system relies on a combination of stored representations (for representing situations, defining perspectives, and articulating language expressions) and plasticity (for tailoring the existing representations to the requirements of the specific design process). This combination makes support of interpretation in design possible and simultaneously drives an evolution of the stored knowledge base.

The theory proposed in Chapter 6 views the computer as a design medium. It is a multimedia device capable of representing the diverse forms of information used in design: text, graphics, pictures, pen sketches, numbers, voice, animation, and even video. It can use all these media to externalize design concepts and to store them for future use, serving as a medium of externalization and long-term memory. This means it can be used as a medium of communication among team members and a medium for embedding an artifact's design history within the design of the artifact itself—Reeves (1993) argues for the role of such a medium in supporting collaborative work.

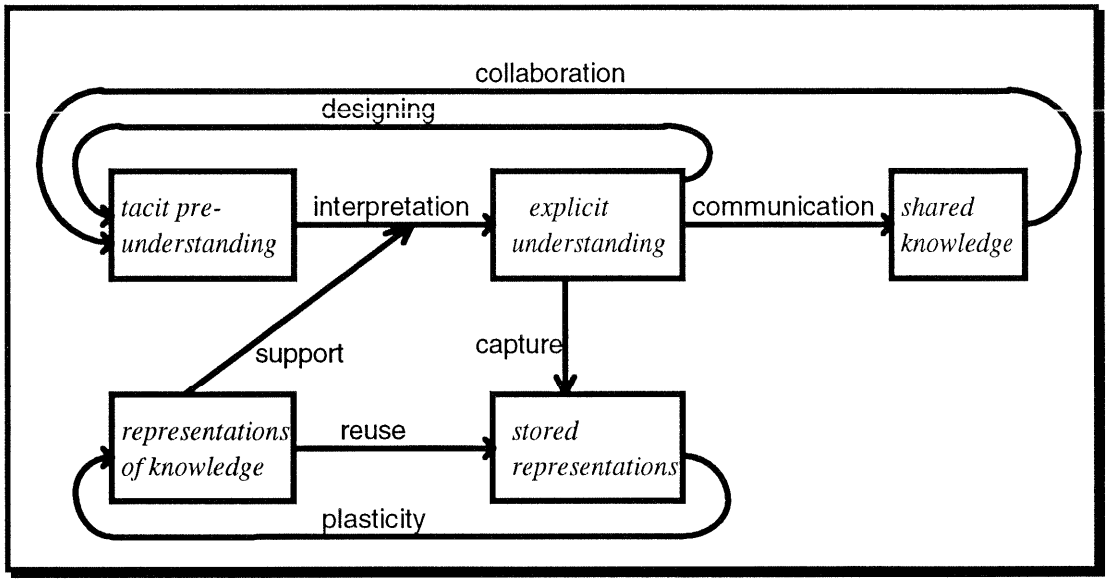


Figure 1-2. The theory of computer support for interpretation in design.
The cycle of human interpretation (illustrated on the top) is mirrored by a cycle of evolution of the computer knowledge base (below), that uses captured explicit knowledge to support future interpretation.

The uses of the computer as designing medium mentioned in the preceding paragraph are primarily passive uses. The impact of written language on civilization shows that even passive media can be powerful. However, the computational power of the computer suggests using it as an active medium as well. Certainly, numerical computations can be left to the computer: calculate square footage of designs or total their costs. But information can also be made dynamic, with representations modified on the basis of the state of other parts of a design. Furthermore, the information stored in the computer can be managed by it, perhaps organizing and displaying information based on a structure of defined perspectives. A language can make the system programmable by designers, so they can adjust displays to their changing needs. Part III will show how HERMES accomplishes this by means of a computationally active form of hypermedia, that integrates a perspectives mechanism and an end-user programming language.

One of the most powerful consequences of designing in a computational medium is the possibility of integrating all the relevant information. An example of this is the mechanism of *interpretive critics* (see Fischer, et al., 1993a). It is an extension of *specific critics* (Nakakoji, 1993). Specific critics are critiquing rules that analyze the representation of a design situation and optionally display a message depending on the results of the analysis. For instance, if two appliances are closer than they should be in the design of a habitat, a critic might display a warning, suggest looking at related design rationale issues, and show similar stored designs that avoid the problem. The specific critics are dynamically computed based on the design

specification that has been entered into the design rationale. The critic thus integrates information about the graphical design, the textual rationale, the computational critic rules, and other designs. It does this in a way that supports the needs of the designer without providing overwhelming amounts of information. Interpretive critics are even further embedded in the contexts of design because they can be defined differently in different interpretive perspectives. Their active behavior depends on the current perspective and the way in which terms in the language are defined in that perspective. They use the language that is being used for the particular design, they are tied to the currently active perspective, and they analyze the represented situation.

The view of computer support systems as computationally active communication media is consonant with a liberatory view of the role of computers in society. Feenberg (1991) argues that the expert system approach based on technical rationality philosophy is profoundly anti-democratic and that an alternative approach to computers as communicative media is needed to give people control over their lives:

> Systems designed for hierarchical control are congruent with rationalistic assumptions that treat the computer as an automaton intended to command or replace workers in decision-making roles. Democratically designed systems must instead respond to the communicative dimension of the computer through which it facilitates the self-organization of human communities, including those technical communities the control of which founds modern hegemonies. (p.108)

The theory of computer support presented in this dissertation pursues the democratic alternative, founding it on a respect for the irreducible nature of human interpretation.

The key is control. Computer systems are sophisticated tools for exerting control of information. As powerful as they are, computers have no understanding of the information they manipulate. Even in autonomous AI systems, all the interpretation is done by people—typically by the programmers who set up the system and the users who view the output. Innovative design is an arena in which the interpretation cannot be done in advance because this design requires understanding and interpretation at every step. Therefore, the role of computers in non-routine design must be to support designers. Human designers must retain control over (a) how things are represented, (b) which things are stored together, and (c) what terms are used to articulate ideas. Unless this control is vested in people who can use their interpretive skills, questions concerning what information might be relevant in a given context or in the future remain intractable for all but carefully delimited, well-understood, completely codified domains. The only heuristics proposed for the management of design knowledge are those tacitly followed by traditional design practice: (1) that knowledge represented, organized, and articulated in the past may be useful in the future, and (2) that designers will need to use their powers of interpretation to modify and apply reused knowledge in unique situations. (The problem of application addresses the fact that every situated, perspectival, linguistic understanding is unique and yet must be interpreted as similar to other cases; it is discussed in Chapter 5.)

The theory of computer support provides a principled basis for designing a computer system to support innovative design in such tasks as lunar habitat design. Before exploring the ideas suggested for such a system, the existing tradition of design environments is considered. This is a tradition of computer systems supporting the augmentation of human design efforts. It provides a basis upon which new ideas can be developed through extensions that are guided by the theory.

1.7. PREVIOUS SOFTWARE SYSTEMS FOR DESIGN

For thirty years now, at least since Alexander (1964), efforts have been underway to use computers to support design. Much work in the area of computer support for design has concentrated on two approaches that will *not* be explored here:

* Providing stand-alone tools for drafting and modeling, where the computer system has little or no representation of the semantics of what is being designed—e.g., so-called "computer aided design" (CAD).
* Automating the design process, where the computer is given a specification of a problem and is expected to produce a design with minimal interaction with a human user—e.g., an expert system for design.

Although these approaches have proven useful for certain tasks or within restricted domains, in general they have been shown to be quite limited. Winograd & Flores (1986) and Dreyfus & Dreyfus (1986), for instance, have argued that expert systems are in principle essentially limited when it comes to tasks like creative design. They have based their arguments largely on Heidegger's philosophy and other ideas that are discussed in this dissertation. Rather than duplicating their line of criticism, Chapter 7 will draw their positive implications for building software systems that can support innovative design.

There have always been some researchers who sought ways to use technology to *augment* human problem solving (e.g., Bush, 1945; Engelbart, 1963), rather than to model, simulate, or replace it. More specifically, there is a tradition in design methodology and design rationale capture efforts, going back to Rittel and his associates (Rittel & Webber, 1973; Kunz & Rittel, 1984) that advocates the use of computer-based design systems as cognitive aids for human designers.

Recent work in this tradition is reviewed in Chapter 7 and used as a starting point for designing a system to support interpretation in design. In particular, the design environments that will be reviewed (JANUS, MODIFIER, PHIDIAS) are *domain-oriented* in the sense that they try to embody generally accepted knowledge of their specialized design domains. In contrast, the domain-independent design rationale capture systems (KRL, PIE, DRL) focus on capturing and displaying potentially opposing perspectives on design issues. By synthesizing ideas from these different systems, the new approach will extend the notion of domain-orientation to support multiple interpretations of the domain as well.

The consequences of the theory of computer support for interpretation in design developed in Chapter 6 motivate and guide the survey of previous software systems. Established techniques implemented in the computer-based design assistants are reviewed and their limitations are critiqued on the basis of the theory. While mechanisms for representing situations, defining perspectives, and using language are found in some of these systems, the plasticity and integration of these mechanisms are quite limited. In many ways, these systems retain principles from expert system theory and are not oriented toward supporting interpretation in design even when they happen to provide some mechanisms that could be used for that.

JANUS (Fischer, et al., 1989) is a design environment combining graphical and textual representations of the design *situation*. It introduces a multi-faceted architecture that includes a palette of design components for building graphical representations of kitchen layouts, a catalog of stored design cases, an issue-base of design rationale, and a daemon mechanism for active critics. This system provides an important model of a design environment. Its lack of support for users to create new representations is recognized and addressed by a successor system named MODIFIER.

MODIFIER (Girgensohn, 1992) defines all the knowledge representations with parameterized property sheets. Then it provides a user interface to these system internals. While it offers extensive support for the user to modify representations, this still involves the user in modifying LISP expressions, altering hierarchical inheritance trees, and generally having to be concerned with system internals. Thus, it supports the user (with extensive help text, examples, checklists, and even critic rules concerning modifications) to engage in tasks of maintaining a sophisticated software system rather than supporting the user in interpretive tasks of design. Another problem with MODIFIER is that it provides no mechanism for organizing modifications into alternative versions to support personal and shared versions.

Several systems for knowledge representation and design rationale capture propose the use of multiple *perspectives*, a mechanism that this dissertation recommends. KRL (Boborow & Winograd, 1977) presents a sophisticated formal language for knowledge representation that incorporates a mechanism for perspectives. PIE (Boborow & Goldstein, 1980; Goldstein & Boborow, 1980a, 1980b) develops the ideas of KRL further as the basis for a design environment for software development. DRL (Lee, 1990; Lee & Lai, 1991) explores issues in design rationale capture using languages based on Rittel's IBIS as well as KRL and PIE. These systems provide invaluable experience in designing languages for knowledge representation and design rationale, and in using perspectives mechanisms. However, their implementations lack the generality called for in certain ways. Furthermore, they are not particularly appropriate to the kind of hypermedia structure that seems useful for representing a broad diversity of design information. They provide important examples and recommend useful principles for the kinds of languages and perspectives mechanisms useful in supporting design. The lessons from these systems are combined in Chapter 8 with two design criteria: (1) the implementations should be appropriate to a hypermedia structure of knowledge representation and (2) end-users

should be able to revise and extend the vocabulary of the language and the structure of the system of perspectives.

PHIDIAS (McCall, et al., 1990) is another design environment like JANUS. It does not include as many components or a critiquing mechanism, but it does demonstrate the utility of a query *language* for users to define displays of design rationale. The PHIDIAS language has a number of important features: it is designed for navigation of hypertext and it is based on several syntactic characteristics of English. Vocabulary in the language can all be defined by users, so it supports adaptability. PHIDIAS uses a form of hypertext that has a fine granularity; thus textual displays of design rationale, for instance, may be computed dynamically through the use of queries defined in the language. The PHIDIAS language provides a good starting point for the design of a computationally powerful language that is appropriate to hypermedia and that can support interpretation.

In response to the shortcomings of previous systems, an integrated software prototype named HERMES is proposed. HERMES is a persistent hypermedia substrate for building design environments to support interpretation in design. Its mechanisms operationalize the positive design principles of the analysis of interpretation and the theory of computer support for interpretation in design.

1.8. HYPERMEDIA IN THE HERMES SYSTEM

In Greek mythology, Hermes supported human interpretation by providing the gift of spoken and written language and by delivering the messages of the gods. As part of the research for this dissertation, a prototype software system named HERMES has been designed to support the preconditions of interpretation (a) by representing the design construction situation to support prepossession, (b) by providing alternative perspectives to support preview, and (c) by including a language to support preconception.

HERMES supports tacit knowing by encapsulating mechanisms corresponding to each of the preconditions:

* Interpretive critics (Fischer, et al., 1993b) are used for analyzing the design situation, which is represented in arbitrarily complex hypermedia data structures. These critics are expressions in the HERMES language that perform an analysis of the current state of some representations and then optionally display a message. The evaluation of the critic expressions or rules is dependent upon the currently active interpretive perspective, which determines the versions of the expression, of its constituent terms, and of the representations being analyzed.
* Named perspectives (Stahl, 1993b) organize and manage alternative sets of information relevant to different purposes. By switching to a new perspective by selecting its name from a list, a designer can change how the representation

of the situation appears, what interpretive critics are active, and in general what contents of the hypermedia network are "visible" from the viewpoint.

* Language terms (Stahl, et al., 1992) define computations across the knowledge base. While these expressions can be arbitrarily complex if viewed in complete detail, they are typically constructed in a series of stages. At every stage, the components of the term's definition can themselves be given names.

With each of these mechanisms, complexities are hidden from the user by being encapsulated in named objects. These complexities can gradually be made explicit upon demand so the designer can reflect upon the information and modify it. Together, these and other mechanisms make HERMES a *computationally active medium* in which designers can do their work.

HERMES is a knowledge-representation substrate for building computer-based design assistants like the Lunar Habitat Design Environment (LHDE) shown in Figure 1-3. It provides a hypermedia structure for designers to build representations of design knowledge.

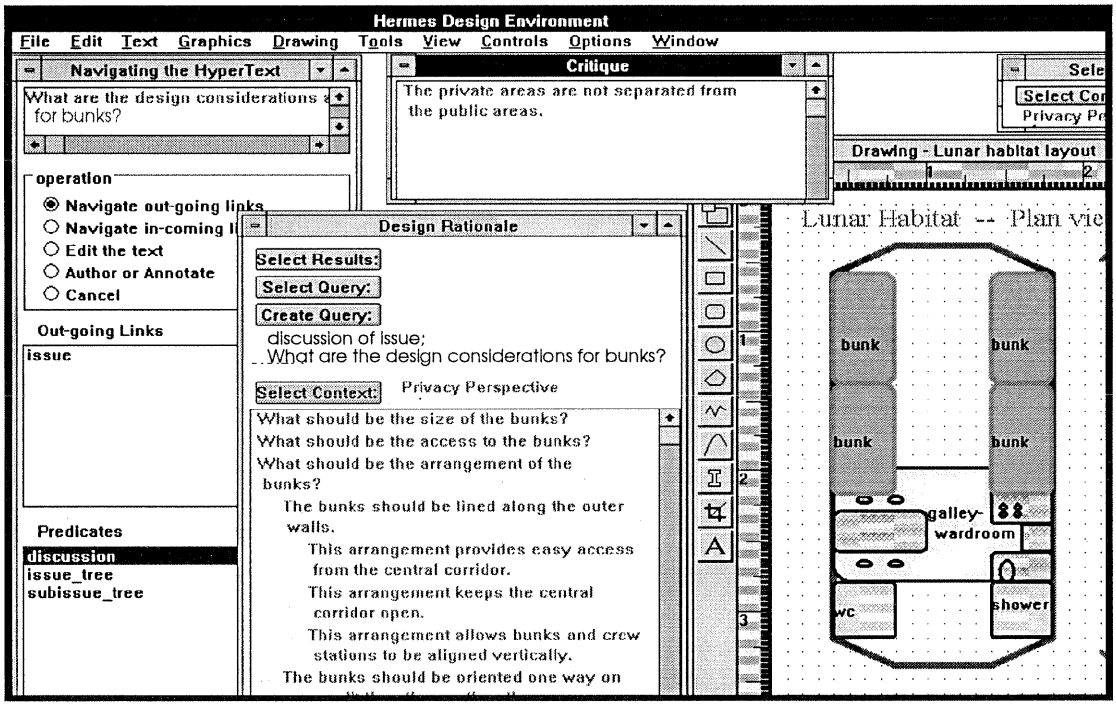


Figure 1-3. Arranging sleep compartment bunks using HERMES.
Windows shown (left to right) include a dialogue box for browsing the hypermedia content, a selection from the design rationale issue-base, a critic rule's message, a graphical sketching area, and a button for changing interpretive perspectives.

The network of knowledge corresponds to the design situation. Multi-media nodes of the knowledge representation can, for instance, be textual statements for the issue-base, CAD graphics for sketches, bitmap images to illustrate ideas, or language

expressions for critics and queries. The inter-linked hypermedia structure facilitates browsing by designers. It can also be used to support associative memory (Hinton & Anderson, 1989) or case-based dynamic memory (Schank, 1982; Kolodner, 1984). All displays are defined by queries that dynamically assemble arbitrary collections of multimedia items. For instance, the Design Rationale window in Figure 1-3 shows the textual issues, answers, and arguments that resulted from a query that was executed by a user's request to see the "discussion" of a previously viewed issue.

The hypermedia *knowledge representation* structure of HERMES is designed to facilitate the representation of design situations and to encourage their tailorability. Its generalized node and link structure models the network character of the situation as a network of inter-related, pre-understood significances and their associations. Its object-oriented implementation allows for the integration of information in different media—reflecting the need to bring together many forms of information in design. It provides graphics for sketching, text for issue-bases or design rationale, and other media for annotations to support the exploration of represented situations. All the media and mechanisms are designed to maximize plasticity of representation. The HERMES hypermedia structure incorporates a perspectives mechanism for managing and viewing all information and an end-user language for defining queries for displays, as discussed below.

Special emphasis is placed on the synergistic integration of the hypermedia, perspectives, and language mechanisms in the HERMES substrate. Definitions of perspective hierarchies and language expressions are stored in the hypermedia network so they can be browsed and modified like all other information. By using nodes of the hypermedia network to define the names of perspectives and links to determine the inheritance relationships among perspectives, the HERMES system can support annotation of these nodes to store information related to the purpose or origination of the perspectives. Similarly, the nodes that define terminology and expressions in the HERMES language can be linked like a semantic network (Quillian, 1967).

In turn, the definition of the hypermedia structure itself incorporates both perspectives and language expressions. Instead of having a fixed content in some medium, nodes can have their content defined by the evaluation of an expression in the language. Nodes and links can be conditional upon some computation defined in the language and involving other nodes and links. Furthermore, hypermedia information to be displayed is always dynamically computed in the currently active perspective—even language expressions can have different effects in different perspectives. In these ways, node contents can be dependent upon the state of other data in the hypermedia network. The interactions of the integrated hypermedia, perspectives, and language provide significant control and malleability for the designer. Design environments built on this substrate can have many features that support interpretation in design with consistent abilities to represent knowledge and to tailor the representations.

1.9. PERSPECTIVES IN HERMES

HERMES includes a perspectives mechanism for organizing all knowledge represented in the system. This mechanism is general and can be used to define a variety of different kinds of "perspectives" for categorizing information and for organizing inheritance of information among perspectives. For instance, hierarchies of perspectives can be defined for technical specialties (e.g., plumbing, ergonomics), knowledge domains (kitchen design, partial gravity design), worldviews (Bauhaus, austere missions), specific designs (i.e., cases), individual preferences, shared team decisions, and experimental "what-if" versions. New perspectives can merge information from multiple existing perspectives and then modify the information as seen through the new perspective without affecting it in the original perspectives. This can facilitate periodic, non-disruptive reorganizations of the knowledge base as it evolves.

The perspectives mechanism of HERMES helps to support the collaborative nature of design by multiple teams. Drawings, definitions of domain terms in the language, computations for critic rules, and annotations in the issue-base can be grouped together in a perspective for a project, a technical specialty, an individual, or a team. A new perspective can be defined to archive a version of a design for historical purposes so it will not change as team members continue to work on new versions. Every action in HERMES takes place within some defined perspective, which determines what versions of information are currently being accessed. Perspectives can collect knowledge according to various categories. For example, there can be perspectives for individual designers or design teams; for technical or professional specialties; for traditional or cultural domains; for specific projects; or for historical versions of projects.

Since information in HERMES is always viewed through a perspective, switching perspectives can support the deliberation of alternative approaches. By redefining in different perspectives the same graphic objects or linguistic terms used in conditionals, queries, and critics, one determines how things will be displayed (interpreted) differently in different perspectives. Thus, as shown by a scenario in Chapter 9, critics in a "privacy perspective" might analyze habitat layouts using a concept of privacy gradient defined in that perspective, whereas the same critics would in effect have different definitions in other perspectives and would therefore produce different results. The interpretive critics for privacy that are used in the scenario are analyzed and explained in detail in Chapter 10 as a case study in use of the language.

The approach of HERMES supports communication among designers. The representations of the design situation may include documentation of design rationale by specifying resolutions of issues in an issue-base. For lunar habitat design, such documentation is contractually required by NASA. Requirements traceability and clear communication of rationale are necessary for a design to move from the original

design team to subsequent groups for approval, technical elaboration, mock-up, and eventual construction. Documentation is notoriously difficult to produce. Design rationale is most effectively captured when it is an explicit concern. Formulations developed in the HERMES language by designers in the midst of designing can supplement the situation representations, stating for the benefit of future designers looking at their work what aspects were originally considered important and what rules of thumb were developed then. Viewing the design from the original team's perspective preserves their interpretation, while subsequent groups can define their own modified perspectives. Individuals in work teams can share ideas, viewpoints, and definitions by using group perspectives that inherit from and modify the contents of their different personal perspectives.

1.10. THE HERMES LANGUAGE

HERMES features a language for designers to use. The language is defined as a series of subset languages to facilitate learning by new users. First it should be noted that previously defined terms and expressions are used most of the time. These are simply selected from lists of relevant terms. Then there is a beginner's version of the language that is very similar to the PHIDIAS language, which proved easy to use for non-programmer novice users. This level of the language suffices for defining or modifying most common terms and queries. An intermediate level provides access to virtually all features of the language except those related to graphics. Finally, an advanced level can be used for graphics-related tasks, like defining interpretive critics. Most system displays and component interfaces are defined in the language, so they can be modified through use of the language.

The HERMES language defines domain vocabulary for referring to represented objects and their associations (the nodes and links of the hypermedia). It also provides expressions for stating queries to define displays and for stating rules to critique designs. The expressions fall into three major syntax categories: (a) definitions of lists of nodes, (b) expressions for filtering out nodes not meeting stated criteria, and (c) operations to traverse various kinds of associations. These support the situated, perspectival, and linguistic character of interpretation by naming representations of things in the design situation, filtering out objects for display based on viewing criteria, and providing expressions for exploring associations. Objects in each of these categories can be either (1) reused or (2) refined by combining expressions in useful ways. This defines the six primary syntactic classes in the language; four other classes provide auxiliary terms and features. The syntactic classes are listed with brief descriptions in Table 1-4.

The language provides a tacit form of language usage for non-programmers. Most of the sequential processing is kept implicit, due partially to the declarative form of the language structure. Also, expressions that were originally figured out explicitly are given names in domain terminology. In Figure 1-3, for example, the user clicked on an issue about sleep compartment bunks and then chose the "Predicate"

(Computed Association), `discussion`. This predicate was already defined to produce a hierarchy of issues with their answers and arguments. The user did not have to be concerned with the recursive structure of this hierarchy or its iterations through multiple links. All of those computational matters were implicit in the definition of the predicate. The user could simply select the predicate by name. This example of choosing "discussion" from a list of predicate names in Figure 1-3 is typical of how the language is used in HERMES. Even when one is creating a new expression, one selects syntax options in dialogue boxes and selects predefined terms from lists. This minimizes the need to remember syntax and terms, prevents many kinds of errors, and avoids the impression that one can simply use free-form English to define expressions.

Table 1-4. Syntactic classes of the HERMES language.

	syntactic class	*description*
a-1	Datalists	options for identifying hypermedia nodes.
a-2	Computed Datalists	permitted combinations of language elements that determine sets of nodes
b-1	Filters	predicates characterizing nodes for selection
b-2	Computed Filters	permitted combinations of language elements that define filter conditions
c-1	Associations	links and other associations of nodes
c-2	Computed Associations	permitted combinations of language elements that determine sets of Associations
d-1	Media Elements	nodes of various media: text, numbers, booleans, graphics, sound, video, etc.
d-2	Computed Media Elements	permitted combinations of media elements, e.g., arithmetic and boolean computations
e-1	Pre-defined Terminology	connective terms, measurement primitives, fixed values for attributes and types
e-2	Computed Terminology	namable quantifiers and numerical comparisons

The HERMES language pervades the system, defining mechanisms for browsing, displaying, and critiquing all information. This means that designers can use the language to modify and refine the representations, views, and evaluations of all forms of domain knowledge in the system. All vocabulary in the language is modifiable by the designers. Every language expression (and every component of a larger expression) can be encapsulated by a name, so that many statements in the language can be defined with common terms from particular design domains. Considerable effort was put into the design of the language to make the appearance of expressions as easily interpretable as possible. Chapter 10 presents many examples

and discusses the techniques used to achieve a readily interpretable appearance. This is just one way in which the language is designed to support tacit usage. Much of the knowledge that people must explicitly use in writing programs in conventional programming languages (assignment, variables, functions, quantification, etc.) has been hidden from the user in the HERMES language (see Chapter 10 for a detailed description of this). The power of these mechanisms is available through the language, but designers need not think in terms of the computational mechanisms. However, when it is necessary for a designer to explore the definition of a user-defined expression in the language in order to understand it more explicitly, this can be done.

Combined with the perspectives mechanism, the language permits designers to define and refine their own interpretations. This allows the HERMES substrate to extend systems beyond the domain-oriented approach of the knowledge-based design environments that HERMES grew out of, by supporting multiple situated interpretations of the domain. That is, the previous systems pre-defined most domain knowledge in a single, generic knowledge base. However, all representations are relative to human interpretation and interpretation is perspectival. HERMES lets designers reinterpret linguistic expressions of knowledge already in the system and store them in appropriate perspectives. This retains the relationship of design knowledge to interpretive perspectives. It also replaces the notion of a single body of domain knowledge (whether fixed or evolving) with a system of multiple perspectives on the domain. Furthermore, this extension encourages inter-related or relevant knowledge from diverse domains to be brought together in specific perspectives.

1.11. CONCLUSION

The analysis of situated interpretation argues that only people's tacit preunderstanding can make information meaningful in context. Neither people nor computers alone can take advantage of the huge stores of data required for many design tasks; such information is valueless unless designers can use it in their interpretations of design situations. The data handling capabilities of computers should be used to support the uniquely human ability to understand. The theory of computer support for interpretation in design suggests that several characteristics of human understanding and collaboration can be supported with mechanisms like those in HERMES for refining representations of the design situation, alternative perspectives, and linguistic expressions. The theory provides a coherent framework for a principled approach to computer support for designers' situated interpretation in the age of information overload.

In elaborating the argument of the previous paragraph, this dissertation seeks to make three kinds of contributions: to a philosophy of interpretation, to a theory of computer support, and to a system for innovative design.

* It makes a philosophic contribution by clarifying the foundations of situated cognition theory in Heidegger's philosophy of interpretation and extending

that philosophy through an analysis of interpretation in design and through a theory of computer support for interpretation in design.

* It makes a contribution to computer science by arguing that systems to augment human skills in innovative design should be oriented toward providing support for the processes of interpretation.

* It makes a practical contribution by prototyping three crucial mechanisms for design environments: a hypermedia substrate that integrates a perspectives mechanism and an end-user language.

These contributions reflect a belief that our age calls for alternatives to a technical rationality philosophy, an expert system approach to computerization, and a view of the designer as an isolated and unaided subject.

PART III. COMPUTER SUPPORT OF NON-ROUTINE DESIGN

The following chapters discuss the three major features of HERMES: the hypermedia knowledge representation, the perspectives mechanism, and the end-user language. HERMES is an instantiation of the theory of computer support proposed in Part II. The discussion of these features of HERMES is intended to illustrate how a system based on the theory might look—a set of mechanisms for supporting the situated, perspectival, linguistic character of interpretation. While the theory suggests the usefulness of a language and a perspectives facility, many very different kinds of languages and perspectives mechanisms are possible. The particular mechanisms in HERMES that have been prototyped as part of this dissertation, suggest one possible approach. The discussion of these mechanisms should illustrate the application of the theoretical framework previously developed to the concrete design of software; these mechanisms represent an attempt to *transform the philosophical interpretations into practice.*

In this Part, Chapter 8 discusses the integrated hypermedia structure. This provides the medium for representing the design *situation* using the many media of design. The *perspectives* mechanism of Chapter 9 provides for flexible organization of all knowledge in the system in order to support collaboration. The *language* presented in Chapter 10 offers designers increased power for interpreting, communicating, and capturing their tacit understandings more explicitly.

Each of these chapters is divided into three sections. The first reviews the needs which must be addressed by the mechanisms discussed in the chapter. The second describes in some detail the implementation of the mechanisms in the HERMES prototype. The third illustrates how the explicit mechanisms are actually used by designers working in HERMES. Generally, the interfaces to these mechanisms encapsulate their computations so that they normally function behind the scenes of relatively tacit usage by designers, only becoming more explicit when the designers must articulate their understanding.

Together, the three mechanisms that are detailed here are intended to support interpretation in design. Specifically, they support the situated, perspectival, linguistic character of design. The kind of design they are meant to support is that of exploratory domains like lunar habitat design, which can be characterized as innovative in nature and collaborative in structure. The computer support proposed has been developed particularly to help designers move back and forth along the spectrum of tacit and explicit understanding. The description of each mechanism will show how it promotes tacit usage as well as facilitating more explicit understanding when that becomes temporarily necessary.

CHAPTER 8. REPRESENTING THE DESIGN SITUATION

Many forms of knowledge are required to support design. The lunar habitat designers in Chapter 3 used sketches of previous designs, graphical representations of design components, discussions of design rationale, terminology for thinking about the design, information from experiences of former space missions, drawings from references, and guidelines from NASA documents. They viewed problems from alternative perspectives and they deliberated issues using concepts that were redefined in the process. Rather than simply constructing a solution from these many pieces of retrieved knowledge, the designers continually modified the knowledge, trying numerous variations. They continually reinterpreted their task, candidate solutions, and the knowledge that went into the solutions.

To support what Part I of the dissertation described as the process of interpretation in design with a computer-based design environment requires a system that provides many media of representation. Furthermore, the representations of knowledge in the media must be designed to support incessant modification, tailoring, customizing, or plasticity by end-users.

According to Part II, a design environment should be people-centered, supporting the human designer's ability to interpret and make judgments. It should support tacit usage as well as allowing designers to make knowledge successively more explicit to meet their specific interpretive needs. This suggests incorporating an end-user language for explicating terms and a perspectives structure for organizing different people's customized versions of knowledge. To take advantage of the computational power of the computer, a design environment should provide a computationally active medium in which the designers can work individually, communicate with the computer, and collaborate with other designers on team work.

The HERMES system described in Part III attempts to meet these requirements by providing a substrate of functionality that can be used by all components of a design environment. It defines a multi-media structure in which all elements of knowledge can be defined and interconnected. All knowledge is represented as data that can be retrieved and modified by the end-user. The knowledge representation structure integrates a perspectives mechanism so that all representations of knowledge are organized into hierarchies of user-defined contexts. It also integrates a language that designers can use for defining and modifying representations of knowledge, including definitions of computer agents such as critics, queries, and displays.

Section 8.1 describes the characteristics of the HERMES substrate. It discusses how it meets the requirements from the analysis of design as interpretation presented in Part I and from the theory of computer support for interpretation in design proposed in Part II. Section 8.2 shows how the substrate is defined at a more

technical level. It discusses the knowledge storage, retrieval, modification, and interconnection mechanisms. Section 8.3 then illustrates how a lunar habitat design environment with multiple components can be built on top of the HERMES substrate. In addition to outlining how components for construction, rationale, specification, and catalogs can be built, it highlights the usefulness of the hypermedia, perspectives, and language in defining these components.

8.1. A COMPUTATIONALLY ACTIVE MEDIUM FOR DESIGNERS

The HERMES substrate. HERMES is a *substrate* for building design environments to support interpretation in innovative design. Many of the previous design environments discussed in Chapter 7 got along without primary attention being given to a substrate level. This is because those systems prototyped functionality specific to individual components. However, recently there has been a proliferation of efforts related to JANUS to introduce functionality that spans all the components of a design environment. KID (Nakakoji, 1993) pushes the non-substrate, multi-faceted approach to its limit, integrating design decisions made in one component with displays in others by "linking mechanisms" to bridge different knowledge representations. But even here, the beginnings of an integrating language are established with the formulations of specification-linking rules, which tie together several major components (critic rules, specification, catalog, domain distinctions). MODIFIER's (Girgensohn, 1992) approach to end-user modifiability of data in all components was an effort that naturally led to integration. The components whose knowledge became modifiable (e.g., the palette and its critics) were, in effect, redesigned to be based on a minimal common substrate of LISP tools for using property sheets. INDY (Reeves, 1993) proposes history capture mechanisms and embedded annotation techniques that apply to events in all parts of the system. In order to implement this, it was necessary to rewrite JANUS to represent all events in the system uniformly. Similarly, the idea of a "programmable application" (Eisenberg, 1992; Eisenberg & Fischer, 1992) suggests the applicability of an end-user programming language throughout a system, as noted in Chapter 7.

An explicitly designed substrate is a way to have various special components implementing multi-faceted functionality while at the same time providing a base of common functionality that is shared by all these components. Certainly, a construction component needs to provide some special supports that are not appropriate to a design rationale component. However, it may be useful to have hypermedia linking, partitioning of knowledge by perspectives, and definition of expressions in an end-user language available in many or all the components. An architecture based on an integrated substrate can support the multi-faceted functionality required for a design environment.

X-NETWORK (Shipman, 1993), for instance, has hypermedia linking, multi-user access, and incremental formality mechanisms that must apply to multiple components; it implements these within a hypermedia object system substrate.

HERMES is also a substrate that can provide functionality that applies to all parts of a design environment built on it. Its language supports end-user-programmability of all components and its perspectives affect all knowledge used in the system. Its critics, palette, catalog, construction, and argumentation displays are all programmable in the language and their definitions or contents are dependent upon the selected perspective.

There are several benefits to creating a design environment substrate. As shown in Section 8.3, it facilitates creating new components within an integrated, high-functionality system by exploiting powerful existing data structures. It permits adding additional trans-component functionality (e.g., for supporting learning, collaboration, interpretation, evolving formality, or agent mechanisms) by enhancements at the substrate level. It provides an integration that helps both developers and end-users because the various components now use standardized structures, mechanisms, and interfaces, so techniques learned in one component transfer well to others.

The layered architecture of HERMES has the following structure (see Figure 8.1):

(1) *Programming environment.* This layer includes commercial object libraries for list processing, graphics, B+ indexing, windowing user interface, etc., as well as the PASCAL source code compiler.

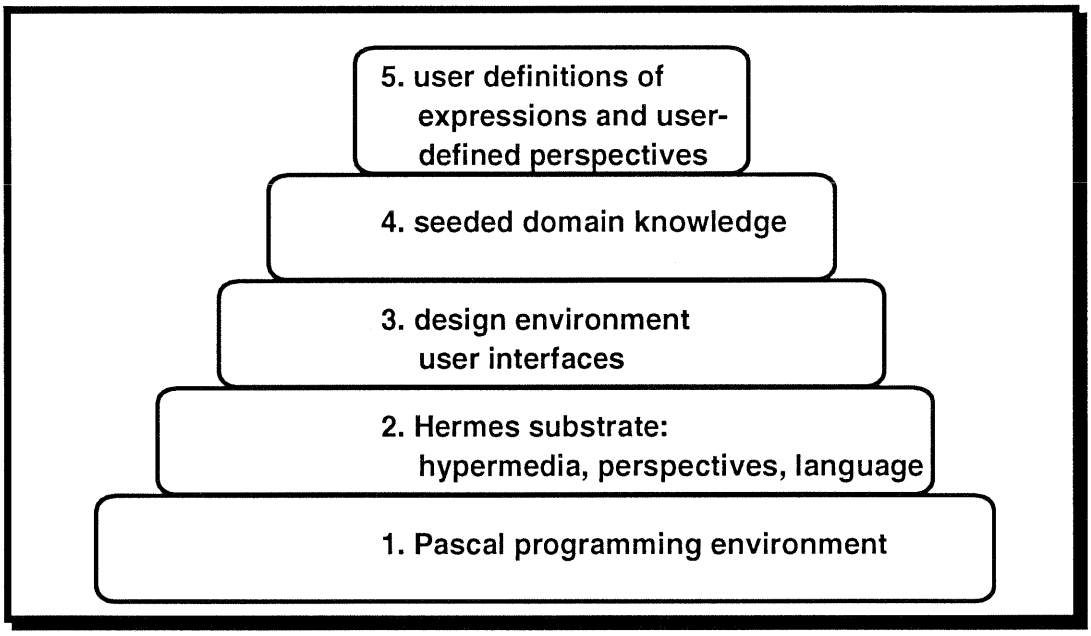


Figure 8-1. Layered architecture of HERMES.

(2) *HERMES substrate.* In addition to the hypermedia structure, the language definition, and the perspectives mechanism, this substrate level includes an efficient, scalable object-oriented database management system for persistence of the

hypermedia data structure. With the language interpreter, this substrate alone consists of about 200 object classes (roughly 20,000 lines of code). The power and flexibility of HERMES for empowering users to represent, manipulate, and interpret domain knowledge comes from the complex interactions of the functionality of the substrate—much more than from the higher-level components of the multi-faceted user interface built on top of it.

(3) *Design environment user interface.* Components like adaptive palettes, design catalogs, and adaptable argumentation are defined as specialized window objects (graphical user interface features). They use the functionality of the substrate to retrieve hypermedia nodes in the active perspective using queries in the language. Some interface components are necessary for user access to the language and perspectives; others are specific to an application, like lunar habitat design. User interface components can take advantage of terms defined in the language, so that end-users can modify the behavior by redefining the terms.

(4) *Seeded domain knowledge.* The system is initially seeded with knowledge specific to the domain for which the system will be used, such as lunar habitat design. This includes definitions of useful terms and queries in the language and an initial hierarchy of perspectives for organizing knowledge. This seeded information is represented using mechanisms defined in the substrate and is stored in the database.

(5) *User definitions and perspectives.* Users can read, modify, and add to any of the domain knowledge. They can organize alternative versions of text, graphics, and language definitions (e.g., domain distinctions, critics, and queries in the language) by perspectives. The substrate is designed to empower users of various skill levels to reuse, modify, and extend all forms of information stored in the knowledge base and to reorganize it into meaningful perspectives.

A hypermedia system. The HERMES system is built on an extended form of *hypermedia.* Hypermedia can be understood as a system of *nodes* having content of various kinds connected together by *links* to form a network or graph structure. Alternatively, if one focuses on the language elements and their interconnections, the HERMES hypermedia can be viewed as an extended form of semantic network (Woods, 1975). In HERMES, the content of nodes can take the form of various media, such as text (e.g., for the issue-base), graphics (for the construction area), or expressions in the HERMES language (like critic rules). In this way, everything that needs to be represented in the computer to support design can be represented in an appropriate data structure that is still part of an integrated system. Each element of information can be interconnected with other elements as needed.

The media requirements for a system to support design are extensive. As mentioned in the introduction to this chapter, the lunar habitat designers in the transcripts used the following: sketches of previous designs, graphical representations of design components, discussions of design rationale, terminology for thinking about the design, information from experiences of former space missions, drawings from references, and guidelines from NASA documents. In order to represent these in the HERMES system, the hypermedia substrate defines the following media for the content

of nodes: character (text), number (reals), conditions (boolean-valued expressions), graphics (vector graphics), images (bitmaps), pen-based sketches, sound (recorded voice), and video recordings.

Because HERMES needs to display information in accordance with interpretations that are not pre-defined but are defined by the user, all displays must be computed dynamically. This is done with dynamic displays, in contrast with the page-based approach of most hypertext systems. In a program like HYPERCARD, a presentation of design rationale might contain a pre-formatted page of issues. Embedded with an issue might be a button for its justification. Clicking on that button brings up another page of text presenting the justification. Similarly, in JANUS a page of design rationale contains highlighted terms; clicking on one of them displays information about that term, allowing one to browse through pages of related textual information. In HERMES, however, the justification must be recomputed based on the current interpretation. This is done by executing a query specifying the information desired (e.g., `justifications of answers of a certain issue`) and based on the currently active interpretive perspective. The results of the query are then displayed, in place of a pre-formatted page. This approach was adopted from the PHIDIAS design environment, which featured a limited query language for allowing the user to structure textual displays (McCall, 1989). Because in this approach design rationale is generally stored at the relatively fine granularity of sentences rather than pages, it can be modified either by changing or adding short sentences, or by modifying the definition of the query.

The HERMES language provides the means of navigating the links of the HERMES hypermedia. Links between nodes have *types*, like an `answer` link to connect an issue with its answers. In addition, as described in Chapter 10, the language defines processes of information retrieval, analysis, filtering, display, and critique, which make link traversal more dynamic than just following static link types. Expressions in the language can be incorporated in computational agents or in interface features of a design environment. All terms and expressions defined in the language are stored as nodes of the hypermedia. The language can also be embedded in the hypermedia structure in various ways. For instance, nodes and links can be made conditional on an arbitrary expression in the language that evaluates (at run-time) to true or false. The content of a node can also be defined by the result of an expression in the language that evaluates to a list of other nodes. These two uses of the language to make the content of nodes dependent upon the run-time evaluation of expressions are known as *conditional nodes* and *virtual structures*, respectively. (See Halasz, 1988, and McCall, et al., 1990a.)

The hypermedia system also defines and incorporates HERMES' perspective mechanism. The links between nodes maintain lists of which perspectives can or cannot view the connected nodes. When the link is traversed during the evaluation of an expression in the language (which is, at an implementation level, the only way that the node the link leads to can be retrieved or displayed), the currently active perspective is compared with this list.

Active media. The HERMES hypermedia provides a computationally active medium for designers to work in. All information retrieval, display, analysis, and critique is performed by navigating the hypermedia network of nodes and links. The content of the nodes may be dynamically dependent upon other content in the network, as in conditional nodes and virtual structures. Whether or not such nodes are involved, the retrieval of information depends upon an expression in the language, which may in turn be composed of many other terms, whose definitions can be changed. Furthermore, information retrieval and display is always dependent upon the current perspective and the list of perspectives from which it inherits. All of these dependencies are under the control of the person using the system. However, the synergy of the various dependencies (definition of the retrieval expression, content of nodes, definition of language terms, choice and structure of perspectives) quickly exceeds the ability of people to foresee the results in detail. Rather, users of the system proceed with a largely tacit understanding and the computer works out the details. In this way, people can concentrate on the interpretive tasks while the computer takes care of the detailed but routine bookkeeping. This exploits the advantage of a computational medium over passive external media like paper.

People-centered system. The language provides a central control mechanism over computational processes in the HERMES system. As such, it makes the control over all computations ultimately available to designers using the system. The language is a means of communication between the computer and its users, through which end-users can specify in as much detail as they wish how information is to be stored, retrieved, analyzed, displayed, and critiqued. At the same time, the system is seeded with default definitions so that designers do not have to be concerned with these matters in any more detail then they need to be as a result of their design tasks.

Because HERMES is designed for exploratory domains like lunar habitat design, however, a seeded knowledge base is only a starting point and source of reusable definitions. Design requires incessant modification and tailoring of definitions of all relevant knowledge based on the particular design situation, the active perspective, and the linguistic frameworks and terminology in use. This means that all information in the system must be flexibly modifiable.

It is not only that there are no longer any experts in the traditional sense because systems of knowledge have become too extensive and too rapidly changing for individuals to master (Fischer & Nakakoji, 1992). Beyond this, in exploratory design tasks like lunar habitat design, there is no such thing as an objective body of domain knowledge that could even in principle be defined once and for all. So-called domain knowledge arises through processes of interpretation that are situated, perspectival, and linguistic. This certainly does not mean that such knowledge is arbitrary or that it cannot be justified. On the contrary, it is grounded precisely in the situations, viewpoints, and traditions that provide its background knowledge and in the deliberations that importantly accompany it. But, the point is that alternative versions of the knowledge are applicable under different conditions and only designers can determine relevance.

Evolving knowledge base. The plasticity of HERMES' language and other media takes off from the ideas of PHIDIAS. In PHIDIAS, node and link types were user-defined. This was a simple matter of allowing users to define new names for types of nodes and links. Then, new nodes and the links between them could be labeled with any one of these types. The importance of this came in its effect upon the PHIDIAS query language (discussed in Chapter 7). This language consisted largely of options for combining node and link types. So by careful choice of type names, query expressions could be made to read descriptively and the language could be extended to include new terms. The HERMES language is far more complex, but it retains the principle that all semantic elements should be user-definable and namable. In fact, this principle is extended to the various media as well, so that everything in the knowledge base can be named and modified.

All representations of knowledge in the HERMES system are maintained as data in the hypermedia information base on disk. This makes it easy for the system builders who define components for design environments built upon the HERMES substrate, for knowledge engineers who seed or reseed the knowledge base, as well as for end-users who tailor the information to their own needs. Standard interfaces are available for browsing, editing, and extending knowledge in all media.

The HERMES substrate is designed to support constant tailoring of all information in the knowledge base. All nodes in the hypermedia can be browsed, modified, annotated, or deleted within the current perspective. Much knowledge is defined by language expressions, which can likewise be edited. The terms used in expressions can also be edited, and so on recursively. Knowledge is organized by perspectives. Together, the hypermedia, perspectives, and language provide considerable control over all knowledge in the system by designers using it. The following chapters will detail how this works.

8.2. KNOWLEDGE REPRESENTATION IN THE HERMES SUBSTRATE

Figure 8-2 below shows how the functionality of the most important objects in the HERMES substrate is built up. Starting at the top is the generic *HERMES named object.* Any object descended from this can optionally have a name and can be stored on the object stream (file) that functions as the database for HERMES. The objects below it in the hierarchy successively accumulate data slots (indicated in parentheses) and methods.

The *Active object* adds two features that provide considerable power for the advanced user: conditionals and procedures. Any object that inherits these (for instance, all varieties of nodes and links and language elements) can be made conditional upon a language expression or can incorporate an arbitrary procedure. A conditional can be any boolean expression defined in the HERMES language. When an object with this conditional is encountered in traversing the hypermedia, the conditional is evaluated. If it evaluates to true, then the link can be traversed or the node evaluated and displayed, otherwise, the object is ignored. A procedure is a user-

defined procedure written in any commercial programming language that supports WINDOWS dynamic link libraries (DLLs), e.g., PASCAL or C++. HERMES includes a DLL with ten procedure identifiers, so that users can define and compile up to ten procedures. The procedure identifiers can then be attached to HERMES objects. When the object is encountered during hypermedia traversal, the procedure is run. This mechanism of procedural attachment has also been used internally to implement one of the procedures for the HERMES perspectives mechanism (see the implementation of "lazy virtual copying" in Chapter 9). With these mechanisms, procedures written in either the HERMES language or in a general purpose programming language can be embedded anywhere in the hypermedia database.

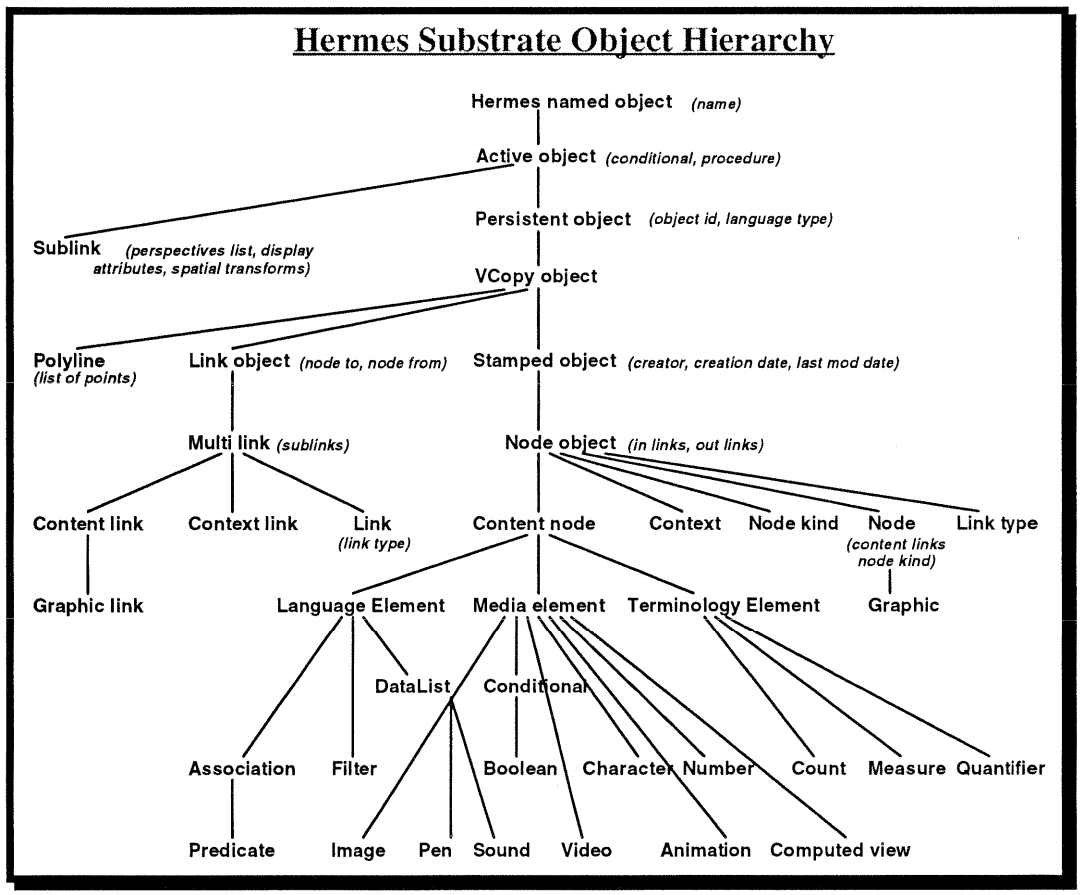


Figure 8-2. The HERMES substrate object hierarchy.

Persistent objects can be retrieved from the HERMES database. They have a unique object id, which is used internally for direct random access to the stream on disk. A set of methods for persistent objects defines an efficient database management system that performs buffered reads from disk. Once accessed, objects are cached in memory by these methods since they are likely to be traversed again. For objects that have user-defined names, a B+ index is used to retrieve the internal object id for object retrieval. This means that even when the database is scaled up to millions of

objects, any object can be retrieved from disk either by user-defined name or by internal id with no appreciable increase in the number of disk accesses. The index to the stream maintains the node kind of each stored node, so lists of nodes of a given kind can be generated. Similarly, the index of named objects maintains the object type, so lists of named objects of a given language type can be displayed quickly for pick lists in the interface.

VCopy objects can participate in the virtual copying mechanisms that implement perspectives in HERMES. A set of ten object methods (defined in Section 9.2) are used for the virtual copying of nodes, links, hypermedia networks, and subnetworks.

Stamped objects are time-stamped with the name of the person who was logged in when the object was created, the date and time of creation, and the date and time of the last modification. This information is useful for browsing the knowledge base with queries in the language. It can also be used for security systems built on top of HERMES.

Node objects are the "first class objects" of the HERMES hypermedia system. They can all be interconnected in the hypermedia, referred to by the language, and organized into perspectives. This is the basis for the interlinked hypermedia structure. Any node objects can, for instance, have annotations or arbitrary features attached to them. A node object maintains a tree of links coming in to it and a tree of links going out. The trees of links consist of lists of link lists, where each link list contains links of a given link type. This list of lists is sorted by link type. The link lists contain the object ids of the individual links. This structure makes for efficient access of a node's links for traversal and language expression evaluation.

Links are stored independently of the nodes that they connect, because they may contain considerable data and may be accessed, traversed, or modified without needing to read in their attached nodes (which may be very large for bitmaps, video, etc.). A link consists of a list of *sublinks*, which maintain information about perspectives, display attributes (e.g., color, font), and spatial transforms (e.g., scaling or rotation for 3-D graphics). By combining a list of sublinks between a given two nodes into one link rather than having multiple links between the same two nodes, the number of links that need to be read in from memory is minimized. Combining all links between two nodes is important because there may be very bushy trees of sublinks due to the perspectives mechanism's implementation. For many functions, one needs to look at all or many of the sublinks. Also, often one only wants to cross one sublink of a link (the first one), otherwise one would get multiple copies; this is efficiently done with a for-each or for-first method on a list of sublinks.

Contexts, *node kinds,* and *link types* are very simple node objects. They just have user-defined names. Contexts are linked in a hierarchy that defines the perspectives and their inheritance relations (see Chapter 9). Node kinds and link types can have synonyms defined. When they are created, the HERMES interface suggests a plural form to be defined as a synonym. This is useful for making language expressions smoothly readable.

Nodes have no content themselves. Rather, they have content links that connect them to content nodes that contain the content (e.g., characters, numbers, language elements). This separation of the named nodes from their content is useful and efficient in a number of ways. It allows a given node to have multiple contents. It may have a different content in different perspectives; it may have several contents of the same or different media; or it may be part of a hierarchy of graphical objects, from a complex lunar habitat, through its components and subcomponents, down to its ultimate polylines of points in 3-D space. The separation of nodes and contents allows perspectives information (as well as display attributes and spatial transforms) to be stored in the intervening links. There are also accessing efficiencies that result from the separation.

Content nodes provide the knowledge representation media. The *language elements* and *terminology elements* are explained in Chapter 10 and in Appendix C. The *media elements* provide the various media required for supporting design. These media elements are traditional objects of hypermedia systems. However, as part of the HERMES substrate their retrieval, modification, display, and analysis take place through mechanisms that are standardized across components, allow integration, are fine-grained, are organized in perspectives, provide for plasticity, and are computed dynamically.

8.3. LUNAR HABITAT DESIGN ENVIRONMENTS

This section will indicate how design environments built on top of HERMES can achieve goals that have long been set for JANUS and PHIDIAS but not previously achieved. In particular, it will argue that the combination of a powerful, integrated hypermedia substrate, a perspectives mechanism, and an end-user language facilitate the desired functionality.

Figure 8-3 shows a screen view of five open windows that are typical of the HERMES interface. This screen view is taken from a prototype Lunar Habitat Design Environment (LHDE) built on top of HERMES. From left to right, the windows are:

1. A control dialog for navigating hypermedia. It shows the selection of the `discussion` predicate for navigating the out-going links from an issue, "What are the design considerations for bunks?" `Discussion` is an expression in the HERMES end-user programming language, defined as an indented hierarchy of issues, subissues, answers, and arguments. The results of the query, `discussion of` the selected issue, is displayed in the next window.
2. The Design Rationale window shows the results of the query evaluated in the *privacy* perspective. The query was defined in the previous control dialog window by choosing a predicate relevant to the `issue` link type going out of the selected issue.

3. The Critique window displays the result of the critics analyzing the construction of a lunar habitat. The critics were evaluated as defined within the privacy perspective. The user initiated critiquing with a button (not shown) in the next window.

4. The Drawing window or construction area displays the current design. This window has buttons (not shown in the Figure) to change perspective, save the drawing in the current context, navigate links connected to the drawing (its rationale), and critique the construction.

5. The Context selection window (partially shown) allows the user to change to a new interpretive perspective in the context hierarchy. This affects contents of textual nodes, definitions of elements of the language used for expressing queries and critics, and contents of drawings.

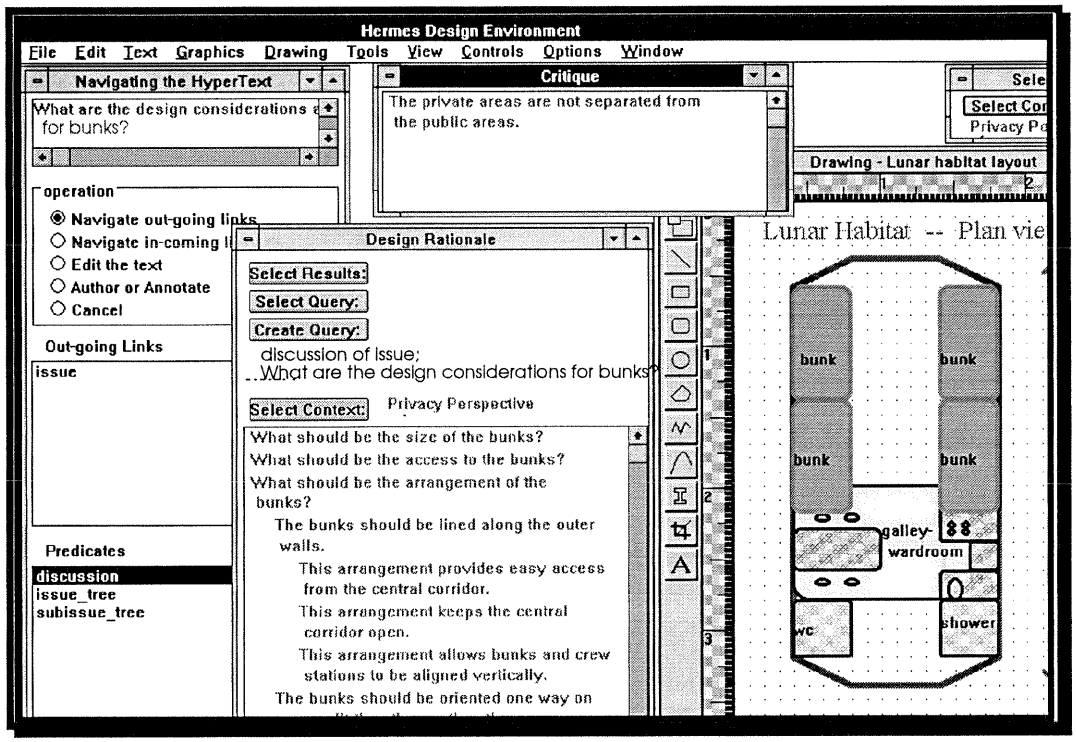


Figure 8-3. A screen view of the LHDE interface.

In this interface to LHDE, one can see a 2-D graphical construction area similar to that of JANUS and PHIDIAS. Subsequently, a version of PHIDIAS II has been built on top of the HERMES substrate by researchers in the College of Environmental Design's CAD lab. It features a very general 3-D construction area, which can be viewed from any angle and distance. It allow a designer to move through the design space and view things at greater or less distance. The LHDE interface shown in Figure 8-3 has a palette of simple geometric shapes along the left edge of the drawing window. The PHIDIAS II interface has palettes of chairs, tables, etc. specifically for

lunar habitat designs. In both cases, the palettes are "hard-wired" and cannot be modified by end-users. However, this is not necessary when using the HERMES substrate. Instead, one could define an expression in the language to display a palette. This has not been done because PHIDIAS II's 3-D graphics system is not yet fully integrated with the HERMES substrate. The advantages of an integrated approach will be discussed below.

The LHDE interface shows a view of design rationale. This is dynamically displayed based on the results of the language expression, `discussion of` the issue selected ("What are the design considerations for bunks?"). Note that the system user did not have to worry about "programming" in the language. Everything was done by direct manipulation, and the language implemented things behind the scenes. The user selected an issue with the mouse in a previous Design Rationale window. The Navigation dialog appeared, with the "Navigate out-going links" option already chosen as the default and with the names of types of links coming out of the selected issue (namely, `issue`, i.e., subissues) listed in the Out-going Links box and the names of predicates "relevant" to those types (i.e., language expressions that begin by traversing links of those types) listed in the Predicates box. When the user selected `discussion` from the list of Predicates, the system automatically applied the `discussion` predicate to the previously selected node and evaluated the resulting language expression within the active perspective. The result is displayed in the new Design Rationale window. That window also has buttons so that the user can modify the display in a number of ways. The display can be replaced by selecting previously saved results. (A button for saving the current results is located at the bottom of the window.) Another button allows the user to select a different query to be evaluated; it displays a list of all defined queries. A third button allows the user to create a new query. This is the point where something like programming may enter, although the interface for the language encourages reuse and modification of previously defined expressions (see Chapter 10). Finally, a last button allows the user to select a different perspective, thereby changing the display.

The PHIDIAS II interface provides an alternative display mode for design rationale and similar displays. Rather than showing an entire indented structure, it displays the top level of the outline form only (the unindented nodes). Every node that has hidden indented material is indicated with a small icon. Clicking on that icon displays the next level of indentation under that node. (This is similar to file directory displays in the Macintosh SYSTEM 7 and WINDOWS 3.1.) What is interesting here is that this mechanism is implemented with the HERMES language, not in a hard-wired, programmed-in way. That is, clicking on a node's icon causes the evaluation of the expression, `all associations` of that node in the result list. The availability of the language made it easy to implement this interface feature, and ensures that the feature can be flexibly modified by simply modifying the definition of the language expression (which does not require recompilation) and can be done by an end-user.

The critics in LHDE are passive agents, similar to the triggers in PHIDIAS. That is, the user must press a button to evaluate the critic rules. In LHDE, the critic rules are expressions in the language. (See Chapter 10 for the LHDE version of JANUS'

kitchen critics and for an analysis of privacy critics for lunar habitats.) No additional mechanisms are necessary because the language is designed to traverse and analyze the hypermedia representations of the design situation. In PHIDIAS II, the triggers for displaying design rationale on the selection and location of palette items is implemented using the HERMES language. For instance, the trigger for selection of chairs evaluates the expression, `discussion of chair selection issue`. As `discussion` is defined in the LHDE seed, this goes to the issue named `chair selection issue` in the issue base and displays all the related `issues`, `answers`, and `arguments`.

Of course, one could add additional components to a design environment built on HERMES. For instance, one could make critics fire automatically when a design unit they were defined for is moved, as in JANUS. One could define specification linking mechanisms as in KID, or formalization mechanisms as in X-NETWORK. Even if these mechanisms were borrowed from other systems, the HERMES substrate would pay off. Critic rules would still be defined in the HERMES language, without having to be programmed in LISP, and they could be associated with design units via general-purpose hypermedia links instead of special mechanisms. The specification linking would be greatly simplified in LHDE by defining domain distinctions as well as critic rules in the HERMES language, and then using the language to traverse the specification hypertext. Even the formalization mechanisms would be aided by using the HERMES language for formulating queries (as suggested in Chapter 7) and for providing a medium of formal (computer understandable) expression. The perspectives feature would also come in handy, allowing different versions of critics to be organized into perspectives and using these perspectives for making their critics more specific to the situation corresponding to that perspective (perhaps obviating the need for a separate specification mechanism).

The most important benefit of the HERMES substrate is the synergy possible with the hypermedia network, the perspectives organization, and the language expressivity. For instance, the HERMES substrate provides a useful basis for finally achieving the goals proposed as "future work" in the classic JANUS paper (Fischer, McCall, Morch, 1989), as discussed in the following paragraphs:

> (1) Within the argumentation system there is a pressing need for authoring to be integrated with browsing. (2) Allowing ad hoc authoring during browsing would enable the designer to annotate the issue base, record decisions on issues and generally personalize the argumentation. (3) This in turn would create the need for certain basic kinds of inference mechanisms. (4) For example, if the designer has rejected the answer "dining area" to the issue "What functional areas should the kitchen contain?" then the system should probably not display any issues, answers or arguments that presuppose or assume that the kitchen has a dining area.
>
> (5) Construction and argumentation might usefully be connected in a number of additional ways. (6) Catalog examples could be used to illustrate argumentation,

and argumentation could be used to help in selecting examples from the catalog (p.12; sentence numbering added).

(1) Integrating authoring with browsing. In the LHDE interface, authoring is integrated with browsing. At every step of hypermedia browsing, the navigation dialog in Figure 8-3 gives the user a choice of traversing out-going links, traversing in-coming (inverse) links, editing the current node or authoring or annotating the node. The editing option brings up an editor appropriate to the medium of the current node, with its content ready to be edited. (In cases of multiple contents, the contents are automatically placed in the editor consecutively.) The authoring option allows the user to create a new node and link it to the current node. Annotation is a typical application of this, where one links a text node to the current node with a link of type `annotation`. Adding the phrase, `with their annotations`, to a predicate will then include all the attached annotations in a given display. Of course, all authoring in LHDE takes place within the current perspective.

(2) Personalizing the argumentation. The authoring option in LHDE is also used for recording decisions in an issue base. Suppose you are browsing through a series of issues that correspond to the issues in KID's specification component. Then when you come to an `answer` that you wish to accept as a specification for your design, you can author a node that you attach to the `answer` with a `resolution` link. You define its content as the boolean value `true`. (This is easier to do in the LHDE interface than it sounds when described because the separation of nodes from their content is never apparent to the user, and the hypermedia linking is generally transparent.) In favoring the personalizing of the argumentation in the preceding quotation, the developers of JANUS did not consider carefully the implications of having many users "personalizing" the same homogeneous issue base. It is one thing for Rittel to have advocated including the deliberations of half a dozen opposing positions in a single issue base; quite another to accumulate the exploratory thoughts of arbitrarily many users, over long periods of time, following diverse and unrelated interests. This may not be a problem for a single-user system; however, LHDE is intended as a repository for extensive exploration. The perspectives mechanism is an important tool that allows "personalization" to scale up in LHDE and to function in a collaborative setting.

(3) Inference mechanisms. In HERMES, the inference mechanism is not some add-on function, but the embedded language itself. While the language does not allow fully general inference across large sets of production rules, it does allow people to encode dependencies. Conditionals, for instance, are used in a number of ways in LHDE. The evaluation of any object in the database can be made conditional upon an arbitrary expression in the HERMES language that evaluates to `true` or `false`. Queries incorporating such conditional expressions can also be defined as the content of nodes. Another approach is used in LHDE to preface display expressions with conditional expressions, as illustrated in point (4).

(4) Adaptive argumentation. In LHDE one can build up a `dining area conditional` as follows:

```
if resolutions of answers of the functional areas issue
  that contain "dining area" are true.
```

As would be clear when building this expression in the language interface (shown in Chapter 10), the phrase, `that contain "dining area"`, is applied to the `answers of the functional areas issue` prior to checking if the `resolutions` of the `answers` that pass through that filtering condition have the boolean content, `true`. Once this conditional expression has been defined, it can be used in the variety of ways suggested in point (3). For instance, if the design rationale included the display expression, `discussion of dining area issues`, then that expression could be modified to be: `if dining area conditional then discussion of dining area issues`. This would display the `issues`, `answers`, and `arguments` concerning dining areas if and only if the `dining area answer of the functional areas issue` had been resolved in the positive.

(5) Connecting construction and argumentation. Because the HERMES hypermedia substrate integrates the construction graphics and the design rationale text, graphical examples from the catalog can be linked to entries in the issue base. Assume that a particular kitchen layout is linked to an `issue` about dining areas with an `examples` link. Then you can amend the display expression above to include `dining area issues with their examples`. Depending on whether the LHDE or PHIDIAS II interface was being used, either the text and graphics would be inter-mixed in the outline indented form, or the graphic examples would be represented by an icon and clicking on that icon would display the graphic in situ or in another window.

(6) Connecting catalog and argumentation. In LHDE, catalogs are not fixed displays. They are defined by language expressions. These expressions can, of course, be modified with conditionals and other inferencing computations. Following are some sample catalog definitions illustrating a filtering of the content of the displayed catalog based on decisions in the argumentation (i.e., the issue base is treated as a specification component):

```
if dining area conditional then kitchens that contain
  dining areas

if safety is important then kitchens that are safe

if privacy is important then habitats that have parts
  that have privacy ratings

if privacy is important then privacy gradient catalog
```

The first of these evaluates the conditional that was defined earlier. If it is true, then `kitchens` are displayed if they contain subparts that are of node kind `dining areas`. The second makes use of an expression named `safety is important`, that checks the resolution of some issue related to safety. It then evaluates an

expression that performs an analysis of kitchen layouts similar to the safety-related subset of JANUS' critics. The third again begins with a specification conditional. It then accesses all habitats in the database. For each habitat, it goes through its subparts to see if any of them have a link of type `privacy rating`. As soon as such a link is found, the habitat is added to the list of items to be displayed. The last expression takes the idea of the third one further, critiquing the separation of parts of a lunar habitat based on the privacy ratings attached to its parts (see Chapter 10 for a detailed analysis of this last expression).

These examples of the synergy possible with the HERMES substrate have emphasized the use of hypermedia linking made possible by an integrated substrate. That is, all the objects inherit common functionality, including the ability to be linked together. The role of the language as a tool for traversing the hypermedia has also been emphasized. Expressions in the language can be defined to relate information from different components of a design environment. The utility of the perspectives mechanism has not been stressed as much. However, it can play a powerful role in personalizing the information, in coordinating sets of specifications, and in promoting collaboration. That theme will be taken up in the next chapter.

CHAPTER 9. INTERPRETIVE PERSPECTIVES FOR COLLABORATION

The HERMES substrate includes a mechanism for organizing knowledge in a design environment into a network of *perspectives*. These perspectives provide support for design as a process of interpretation and deliberation. They allow designers to interpret the design situation according to their individual and group interests. Perspectives provide a mechanism for creating, managing, and selectively activating different sets of design knowledge, such as critics, spatial relations, domain distinctions, palette items, and argumentation, so that alternative ideas can be deliberated and either adopted, rejected, or modified.

The perspectives mechanism organizes all the design information in the knowledge base. A designer always works within a particular perspective. At any time, the designer can select a different perspective by name. When a given perspective is selected ("active") then only information indexed for that perspective (or for a perspective inherited by that perspective) can be accessed, traversed, or displayed.

A new perspective can be created by assigning a name to it and selecting existing perspectives for it to inherit. Perspectives are connected in an inheritance network; a perspective can modify knowledge inherited from its parents or it can add new knowledge. Designers switch perspectives to examine a design from different viewpoints. Switching perspectives changes the currently effective definitions of critics, the terms used in these definitions, and other domain knowledge. For example, imagine that Archie was collaborating with Desi using the HERMES computer system. Then he could create `archie's habitat perspective` and select `desi's habitat perspective` to inherit from. This would allow him to build upon and critique Desi's work, without altering what is viewed by Desi in his perspective.

The organization of information by perspectives encourages users to view knowledge in terms of structured, meaningful categories that they can create and modify. It provides an extensible structure of knowledge contexts that can correspond to categories meaningful in the design domain. This eases the cognitive burden of manipulating potentially large numbers of alternative versions of critics, rationale, graphics, language expression definitions, and other design knowledge.

The perspectives mechanism allows items of knowledge to be bundled in various ways, which can overlap orthogonally or inter-connect. Common types of perspectives are:

* personal and group viewpoints of individual designers and teams
* topical groupings by content traditions (e.g., kitchen design)

* technical aspects by specialties (e.g., plumbing)
* historical versions (e.g., Archie's Monday morning habitat design)

For instance, `archie's habitat perspective` might include considerations specific to Archie's design, as well as incorporating many ideas from Desi's. If Desi and Archie are part of a larger team, then the team's perspective could display concepts and rationale from all its members, or it could select from and modify the knowledge inherited from multiple sources. Archie would also want to inherit knowledge from lunar habitat design traditions and related technical specialties. Then, as his design evolved, Archie could define perspectives for archiving versions of his work.

Lunar habitat design takes advantage of information from many technical disciplines and domain traditions: kitchen and bathroom design, low-gravity and vacuum considerations, electrical and lighting expertise, submarine and Antarctic isolation experiences. It can borrow selectively from both space station and Mars habitat prior designs. Each of these bodies of knowledge can be defined within a network of domains and subdomains that inherit, share, and modify knowledge from each other. Perspectives can also be used to save networks of historical versions of developing designs. The HERMES perspectives mechanism is a general—but hypermedia specific—implementation of contexts[4] that can be used to supply a variety of functionality to a design environment.

This chapter will present the HERMES perspectives mechanism in three sections. First, Section 9.1 offers a scenario to show how a design team using HERMES might approach the task documented in the protocol analysis of Section 3.2, "Perspectives on Privacy." Second, Section 9.2 describes the techniques used to implement the perspectives mechanism in HERMES. This will detail the hypermedia character of the implementation. Third, Section 9.3 discusses how the perspectives mechanism can provide computer support for cooperative work. This will include examples of interface features for displaying, browsing, and sharing knowledge in multiple perspectives representing different people, interests, or domains.

9.1. A SCENARIO OF COOPERATION

The work of lunar habitat designers was studied in order to learn about the work process of innovative cooperative design in a complex domain. Lunar habitat design seems to call for computer support because of the volume of technical information and governmental requirements, as well as because of the other-worldly setting in which the designers' tacit skills may be unreliable. It seemed wise to explore

---

[4] The terms *perspective* and *context* will be used interchangeably in this Chapter. Technically, the functionality of perspectives is implemented by defining contexts. As M. Gross suggested, perspectives are similar to the notion of "binding contexts" in programming languages: a definition is bound within the perspective in which it was created.

how lunar habitat designers work now without substantial computer support in order to envision new ways to support the old goals and to imagine how computer support would transform the tasks involved.[5]

The episode transcribed in Chapter 3 showed an important turning point in a design process: the application of the concept of privacy to the task at hand. The tacit notion of privacy was eventually operationalized with the idea of defining a privacy gradient, according to which public and private areas of a habitat are distributed based on their privacy ratings. The concept of privacy then provided a paradigmatic example for investigating the design rationale issue-base provided to lunar habitat designers by NASA: the Manned Systems Integration Standards (NASA, 1989a). Here it was seen that this important concept of privacy had largely eluded NASA's extensive efforts to provide propositional rules for the design of space-based habitation. Although privacy was acknowledged to be an important issue, NASA failed to provide support for designers to take privacy into account.

The present section will build on the discussion in the transcript and the critic definitions to show how HERMES can respond to the challenge of providing computer support for considerations of privacy. A scenario will show how lunar habitat designers could use the HERMES system to define a powerful set of privacy critics using the hypermedia links, perspectives, and language of HERMES. The detailed explanation of how the critics are evaluated by the system will be saved for Chapter 10.

Desi's perspective. Suppose that instead of sitting down together with pencil and paper, Desi and Archie had been part of a team that worked in a design environment built on the HERMES substrate. Desi, Archie, and two other team members (Phyllis and Sophia) are asked to design a lunar habitat for four astronauts to live in for 45 days. They decide to take turns working on the design in HERMES, starting with Desi.

Desi begins creating a perspective for his new work, which he names `desi's habitat perspective`. He defines this perspective to include (inherit) the information collected in a number of specialties and domains that he considers relevant to the design task. Then he selects two other lunar habitat perspectives and copies individual items of graphics and design rationale out of them for the lunar habitat shell, bunk-bed crew compartments, and a wardroom (dining and meeting room) arrangement. He inserts these into design rationale and graphics in his perspective. Then he adds some rectangles to represent the bathroom and galley (kitchen). The resulting layout is shown in Figure 9-1 (reproduced from Figure 3-2 of Section 3.2).

---

[5] This "dialectic of tradition and transcendence" in work-oriented design of computer support systems is a central theme of Ehn (1988). The transformation of tasks as a result of computer support is also emphasized by, for instance, Hutchins (1990) and Norman (1993).

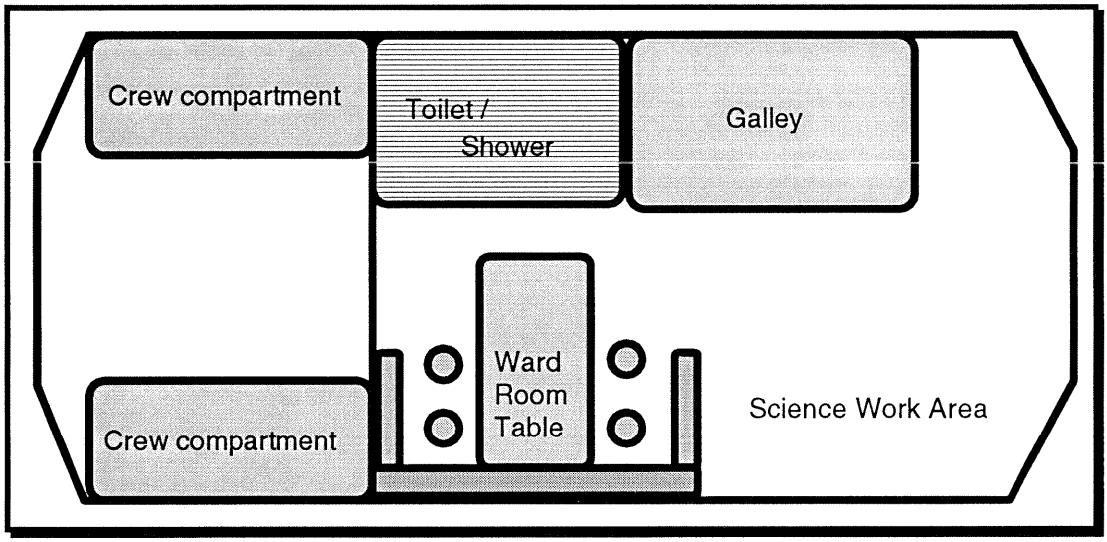


Figure 9-1. Desi's lunar habitat design.
An initial sketch has been proposed for the design team to work on.

The main functional areas of the habitat have been laid out in this sketch. This is an initial design concept. Because other team members will be reviewing this design and wondering why things are arranged the way they are, Desi adds some design rationale, arguing that the bathroom and galley have all been placed together in a "wet wall" configuration to minimize plumbing arrangements. Desi feels his design provides a good start for the team and he goes off to work on other projects.

Archie's perspective. Archie is interested to see what Desi has designed and to critique it from his own viewpoint. However, he does not want to destroy Desi's version. So Archie defines `archie's habitat perspective` as a new perspective and lists `desi's habitat perspective` as its inherited perspective. This means that Archie will start off with everything that is in Desi's perspective, but as he makes changes to it the changes will only be in effect within Archie's perspective and not within Desi's. The inheritance is active in the sense that if Desi subsequently modifies something in his perspective that Archie has not changed in his then the modification will show up in Archie's perspective as well (unlike if Archie had simply made his own copy of Desi's design at some given time).

Archie also inherits a number of additional perspectives with useful technical information. The hierarchy of perspectives incorporated in Archie's perspective—including those he inherits via Desi's perspective—are pictured in Figure 9-2 (below).

Archie is concerned with spatial adjacencies. He likes the way the crew compartments have been separated from the rest of the habitat to provide relief from the daily activity. However, he dislikes the acoustic proximity of the toilet (which flushes loudly) to the beds. Even worse, he finds the opening of the bathroom into the eating and gathering area potentially offensive. Archie is unsure of how to handle the

bathroom, so he switches to a perspective that he has not inherited, the perspective for residential (terrestrial) bathrooms and browses the issue-base section on the design and placement of bathrooms. This perspective inherits from several other cultural and domain perspectives, including European perspectives. Here he finds the idea that showers and toilets have rather different location and adjacency considerations in the European tradition.

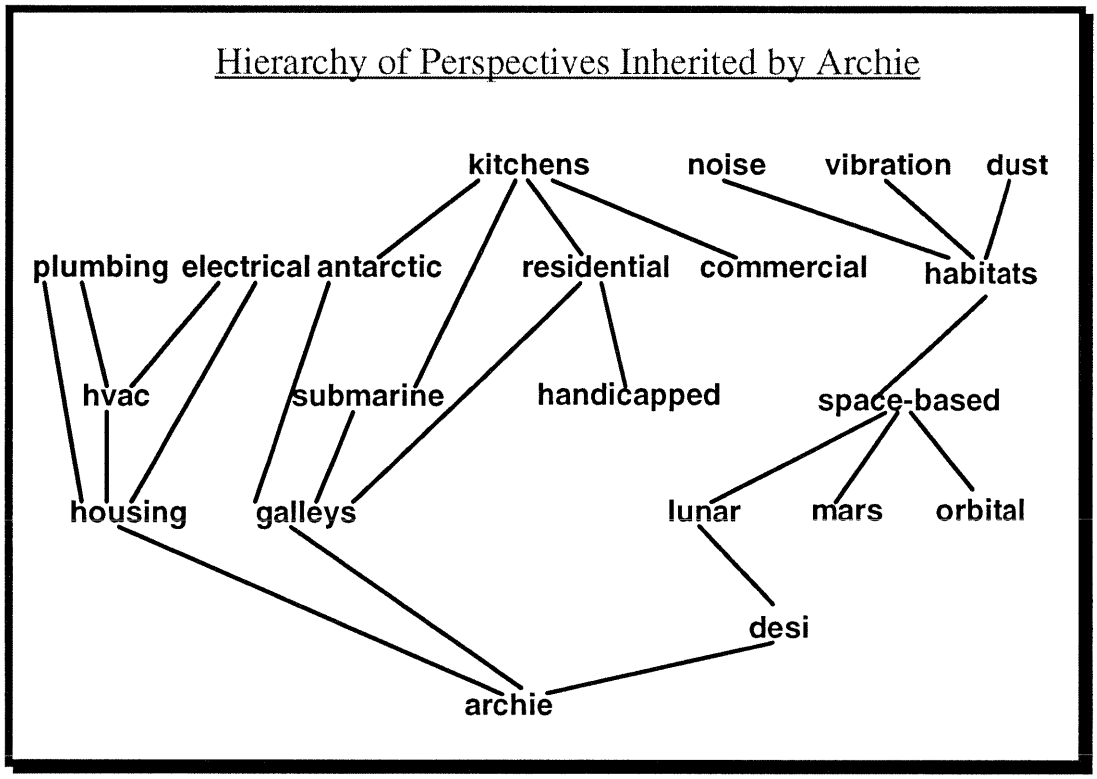


Figure 9-2. The hierarchy of perspectives inherited by Archie.
Note that Archie has access via Desi's perspective to information in the lunar, space-based, habitats, noise, vibration, and dust perspectives, as well as additional information related to housing and galleys.

Applying these ideas in his mind to how he projects life in the habitat, Archie concludes that the shower should be near the sleep areas, but the toilet should be near the other end of the habitat, by the entrance. Moving the shower gives him the idea of elaborating the separation of the sleeping and working areas by forming a dressing area incorporating personal stowage. He redesigns the galley based on other ideas he finds and feels he has reached a stopping point. (See Figure 9-3.) He copies the rationale from the bathroom perspective concerning the separate location of the shower and toilet, revising the rationale to apply to the lunar habitat.

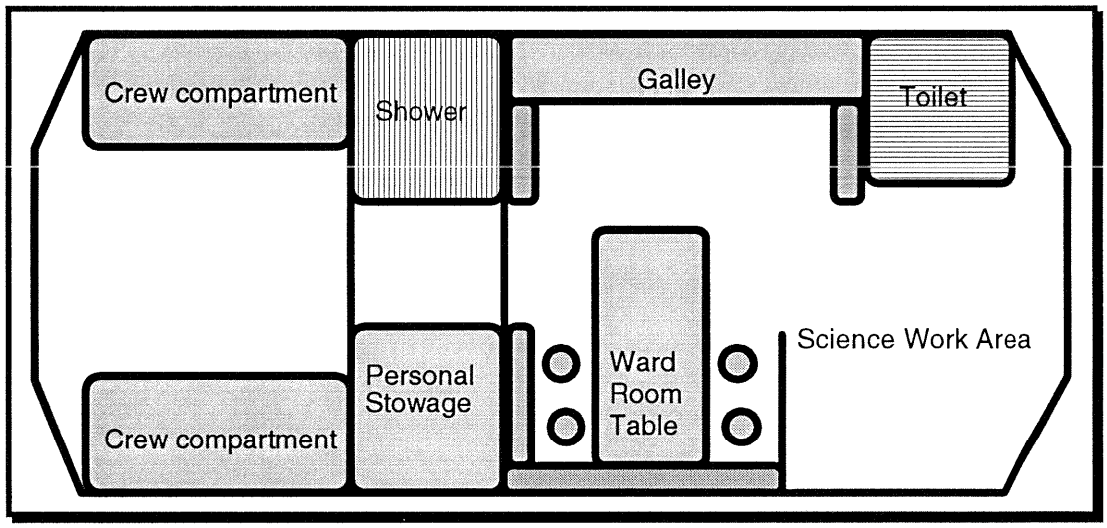


Figure 9-3. Archie's lunar habitat design.
The toilet and shower functions have been separated using the European perspective on bathroom design.

Archie revises the design rationale for the habitat. Within his perspective, he can modify or add to (annotate or author) anything in the issue bases he has inherited from Desi or from the other domains. He does this in preparation for the up-coming team meeting. Before the meeting, the team members each review Archie's design and its rationale by displaying it in HERMES. First, they discuss the over-all design. They like the creation of the dressing area between the shower and the personal stowage, but argue that it blocks traffic flow. A consensus is reached when Phyllis drags the dressing area to the other side of the crew compartments in the HERMES construction area.

As a group they deliberate about the issues in Archie's rationale section and agree that habitation issues must be the primary focus of their designing on this project. In particular, privacy is a key concept. In order to make the notion of privacy operational for evaluation by interpretive critics, they decide to label the parts of the habitat with privacy ratings. They agree on the following scale with values from 1 to 9:

```
very public:       1
quite public:      2
public:            3
somewhat public:   4
neutral:           5
somewhat private:  6
private:           7
quite private:     8
```

```
very private:      9
```

They define a link type, `privacy rating`, and use this type to link each area of the habitat to a node with one of the above numeric values (or their equivalent label). This process is facilitated by the HERMES interface: clicking on an area like the shower in the habitat brings up the same *Navigating the Hypertext* dialog seen in Figure 8-3 (in Section 8.3). Selecting the *Author or Annotate* option allows them to define a new numeric node with the value `8` or `quite private` and to connect it to the shower with a `privacy rating` link automatically. Figure 9-4 below shows the lunar habitat design the team has come up with, labeled with the agreed upon `privacy ratings`.

At the end of the meeting, Sophia and Phyllis agree to develop a suite of privacy critics that can be used for this and future lunar habitat design assignments.

Sophia's perspective. Sophia sets up her perspective to inherit all of Archie's work (and, indirectly, Desi's). Now Sophia must define the terminology to be used in her critics. She is interested in determining problem areas in which private areas are too near to public areas. By "too near", Sophia decides she means less than five feet. So she defines a "Measure" in the HERMES language named `too near` as:

```
closest distance is less than 5 feet
```

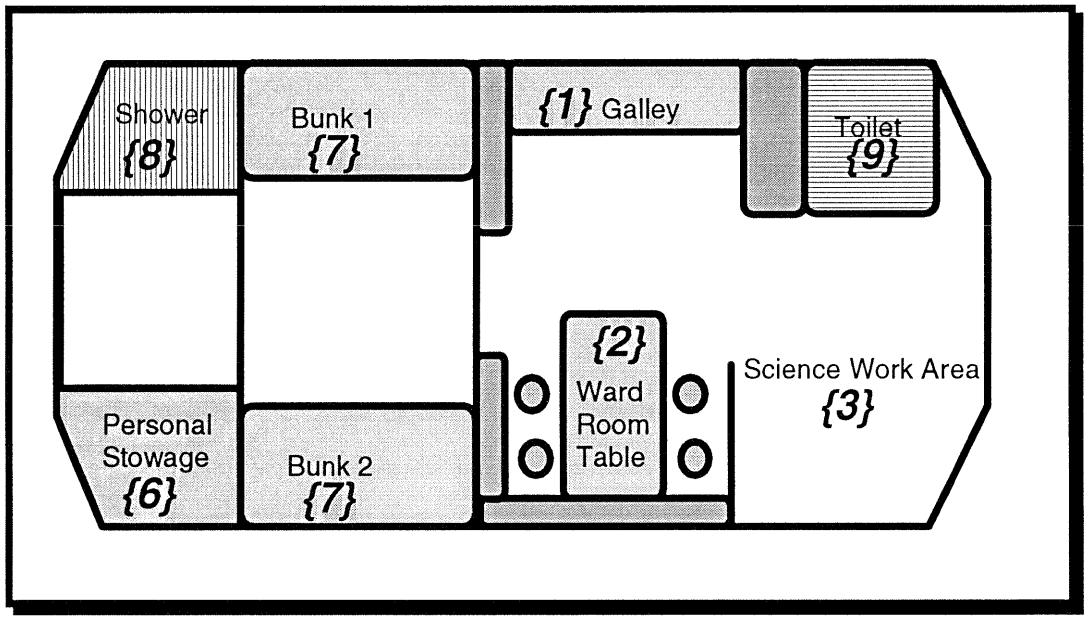


Figure 9-4. Archie's lunar habitat with its privacy ratings.

Then, she defines public and private areas in terms of the ends of the privacy scale:

```
public area: parts that have privacy ratings that are
  less than somewhat public
```

```
private areas: parts that have privacy ratings that are
  more than somewhat private
```

Next she defines the problem areas she is concerned with using these terms:

```
problem areas: private areas with public areas of that
  (last subject) that are too near those items
```

Then, Sophia defines a message for her critic to display if no problem areas are found:

```
privacy message: "Public and private areas are
  separated."
```

Finally, she can define her `privacy check` critic:

```
name with either name of problem areas or privacy
  message
```

This critic, `privacy check`, is a predicate that can be applied to any node or list of nodes in the database. When Sophia applies it to her lunar habitat design, it lists the name of the design and then lists all the `problem areas` in the habitat by their names; if no `problem areas` are found, it displays the `privacy message`. Figure 9-5 shows the output from applying `privacy check` to the design of `archie's lunar habitat` shown in Figure 9-4:

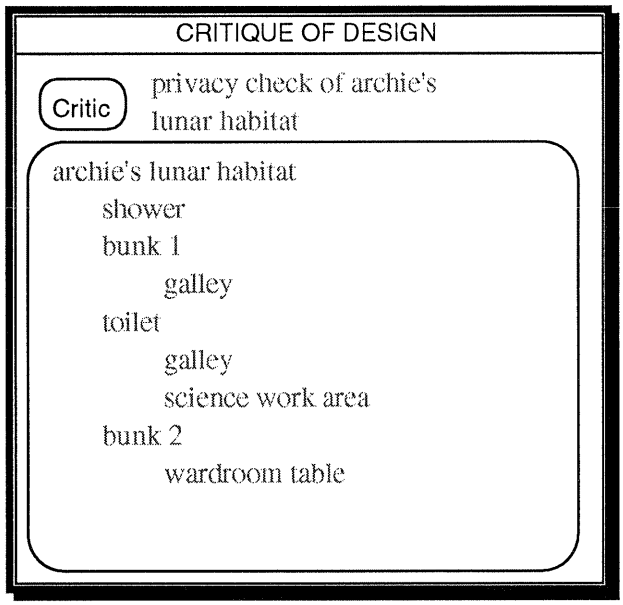


Figure 9-5. Output from the privacy check critic.

Note that all `private areas` are listed by name. Under each of them are the `public areas` that are `too near` to them. The way this critic is defined it supports the designer's review of the information. Sophia gets a complete listing of `private areas` from which she can check just what problematic adjacencies

each has so she can also make sure the critic is doing exactly the computation she wants it to.

Debugging of critics is an important process, particularly since much of the computation is implicit in the language expressions. The `privacy check` is a fairly complex critic that Sophia has developed and debugged gradually. Once she is sure it is working, she can use it as a basis for more complicated evaluations. For instance, the display of the lunar habitat design in HERMES does not actually include the `privacy ratings` that were shown in Figure 9-4. So Sophia decides she wants to print these values out along with the listing of areas. To do this, she defines a new critic that prints out both the `name` and the `privacy rating` of each listed area:

```
privacy display: name and privacy ratings of problem
  areas
```

The result of applying this critic to `archie's lunar habitat` is shown in Figure 9-6. (The names of the `privacy ratings` are shown in **bold**.)

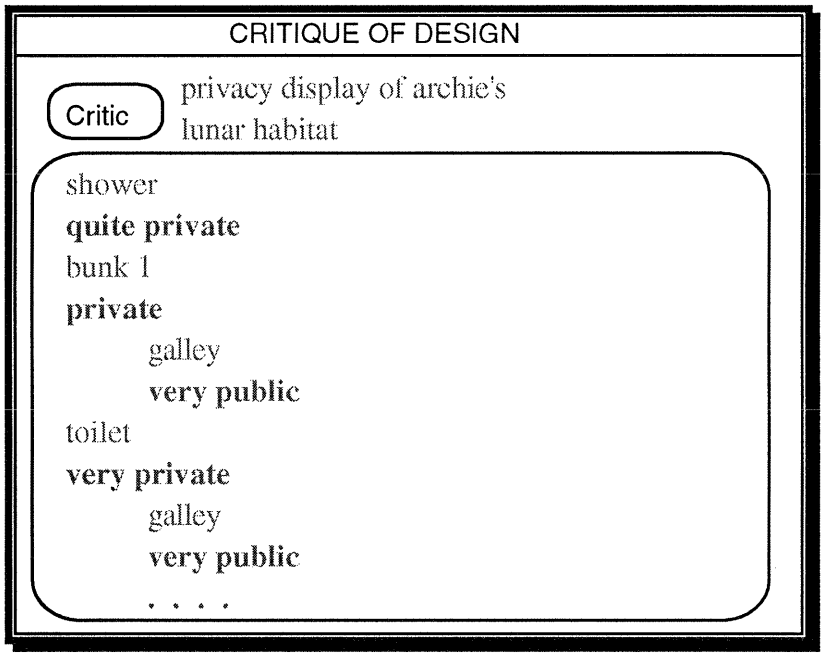


Figure 9-6. Output from the privacy display critic.

Now that Sophia has gotten her critics working the way she wants them to, she decides to make them general enough to apply to lists of objects. Then, as more habitats are developed in the HERMES database and are labeled with privacy values, designers can use Sophia's privacy critics to display catalogs of interesting habitats. This is illustrated in Figure 9-7. This way Sophia can quickly find examples of problem areas in past habitat designs to help her deliberate about when such adjacencies might in fact be acceptable.

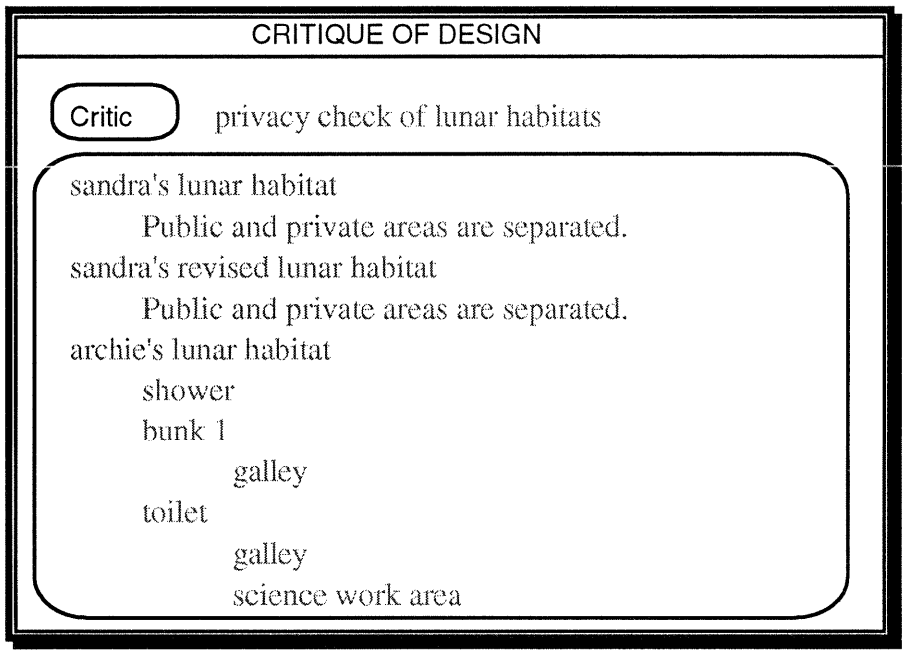


Figure 9-7. The privacy check critic applied to a list of all lunar habitats

Phyllis' perspective. Phyllis is a super-user of the HERMES language. To test its power, she tries to define a critic that involves a complex series of computations. By using an advanced feature of the language (explained in Section 10.3 below), she succeeds. Phyllis recalls previous discussions between Desi and Archie (from Chapter 3) that proposed the concept of a privacy gradient. That meant that the arrangement of the habitat should gradually change from private areas to public areas. To operationalize this notion, Phyllis introduces a test to see if any two areas of the habitat that are near each other differ in their privacy values by more than two.

Phyllis defines the following set of definitions to compute `problem parts` in her sense:

```
are incompatible: have privacy ratings that are more
  than privacy ratings of that (last subject) + 2 or
  are less than privacy ratings of that (last subject)
  -2

too near: closest distance is less than 3 feet

other parts: parts of inverse parts that do not equal
  that (last subject)

problem parts: name and privacy ratings of other parts
  that are too near that (last subject) and that are
  incompatible
```

These definitions illustrate the limits of the HERMES language, calling upon advanced features of the language that only experienced users of HERMES would feel

comfortable using to create new expressions. The wording of some of Phyllis' expressions are no longer intuitive because their computations refer outside of the expressions used to define them. In fact, the wording in such cases is designed to interrupt tacit understanding and to stimulate reflection on the explicit computational relations. Fortunately, this complexity is generally encapsulated in the names of the expressions so future users need not always be concerned with it.

Note that Phyllis has defined a measure with the same name (`too near`) as one of Sophia's, but with a different value. This is not a problem since they are working in independent perspectives (even though they inherit much of the same information from other perspectives.)

To complete the `privacy gradient critique`, Phyllis defines a format for listing problem parts and she specifies a message for the case in which no problem parts are found in a habitat:

```
privacy gradient listing: name and privacy ratings with
  problem parts

privacy gradient message: "The parts of this design are
  arranged along a privacy gradient."

privacy gradient critique: either privacy gradient
  listing of parts or privacy gradient message
```

Like Sophia, Phyllis wants to apply her critique to all `habitats` in the database. Note that in the following definition for this procedure Phyllis first filters the list of `habitats` to just those for which `privacy ratings` have been defined. This produces a list of habitats for which issues of designing for privacy are most likely to have been thought through and to provide relevant ideas and rationale. For these habitats, it is indicated which meet the criteria of following a privacy gradient and where the problem areas are in those that do not. A sample result is shown in Figure 9-8. Here is Phyllis' final critic rule or display expression, `privacy gradient catalog`:

```
name with privacy gradient critique of habitats that
  have parts that have privacy ratings
```

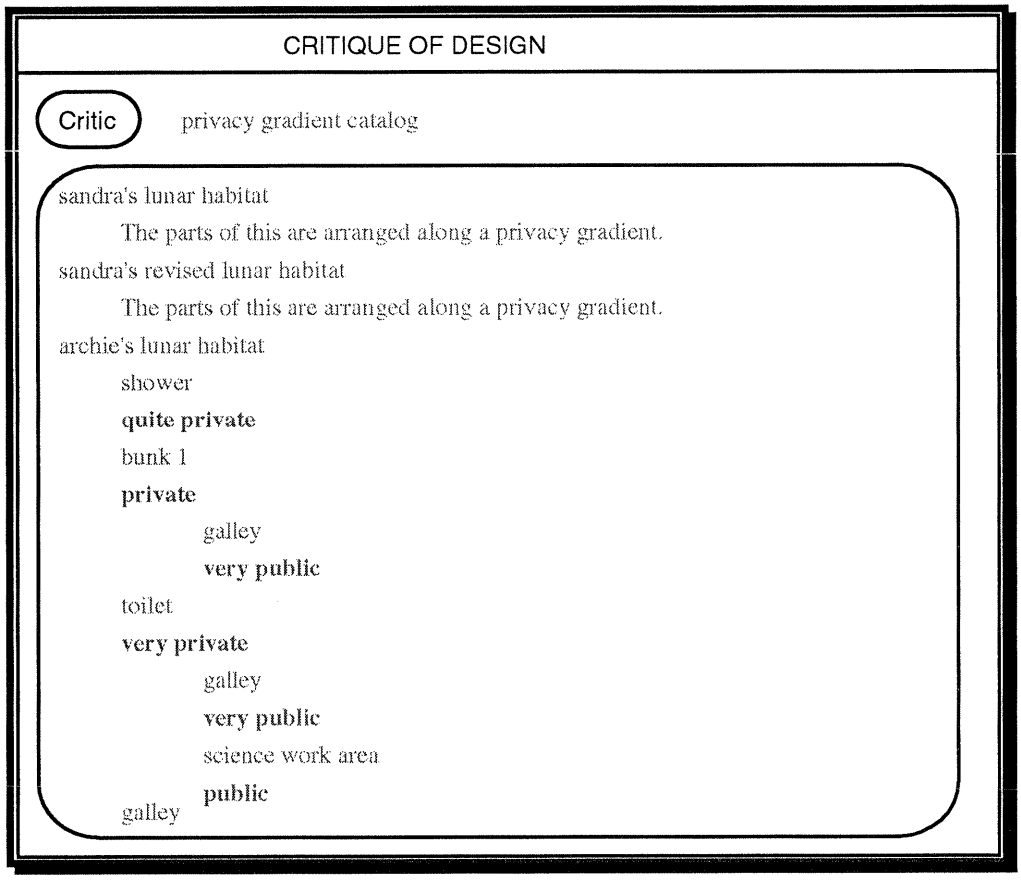


Figure 9-8. Output from the privacy gradient catalog expression.

The team perspective. When the team comes back together, they are enthusiastic about the power of the privacy critics to automate some complex analysis of habitats for them. Desi says, "I never tried to define anything in the HERMES language; I just make little adjustments to the display definitions and critics that I find already in the system. They usually meet my needs. But these new critics do things I could never do before. And I think I understand them well enough to use them and maybe even tweak them." "Yeah," chimed in Archie, "I never used the advanced syntax options for dealing with graphics and distances. Maybe I can learn how to do that by playing around with these privacy critics. Can you put them all in a perspective where we can experiment with them?"

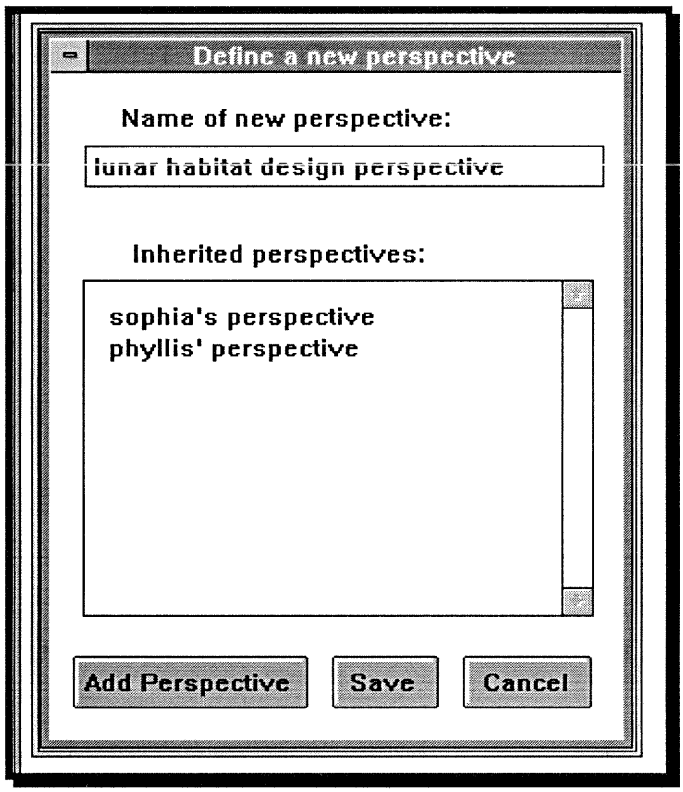


Figure 9-9. Creating a new perspective.

Sophia was happy to oblige: "Sure. The thing we need to be careful about is the definition of `too near`, because Phyllis and I disagree on that. Let's make the default for that 5 feet, okay?" She created a perspective called `lunar habitat design team` that anyone could inherit from to experiment with the critics or to pursue their design work further. She had the new perspective inherit from both the `sophia perspective` and the `phyllis perspective`, making sure she listed the `sophia perspective` first so that its definitions would override in case of conflicts, as with the definition of the expression `too near`.

Figure 9-9 shows the dialog box for creating the new perspective. Figure 9-10 shows the new hierarchy of defined perspectives.

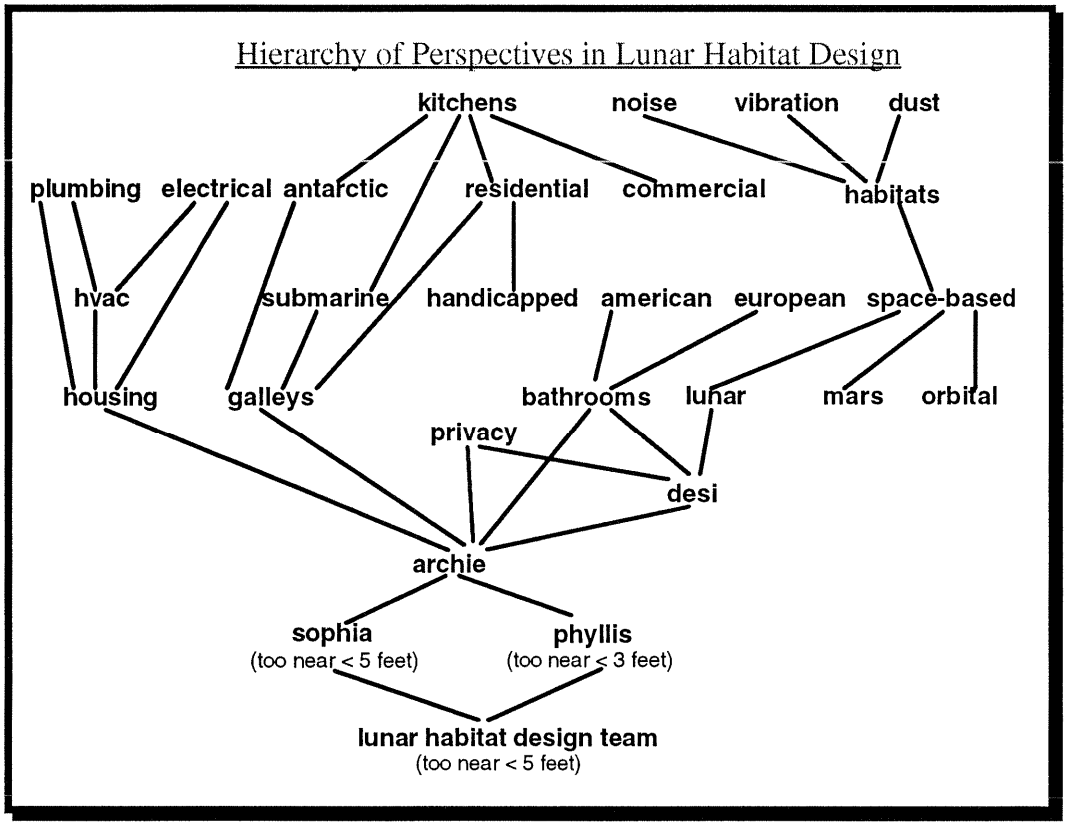


Figure 9-10. Hierarchy of perspectives inherited by the team.

9.2. A HYPERMEDIA IMPLEMENTATION OF PERSPECTIVES

This section discusses the implementation of the HERMES perspectives mechanism. The ten methods discussed below are used by the HERMES substrate internally. The user never needs to know how they work. Even people who build design environment components on top of the HERMES substrate do not need to be concerned with the details, but can simply call the methods. The purpose of this section is to describe some of the computation that takes place behind the scenes every time a designer retrieves, displays, navigates, modifies, critiques, or analyzes information in the system. It is an example of the active computation that supports the user's tacit design work.

As suggested in Chapter 7, the perspectives (or, equivalently, contexts) mechanism in HERMES is loosely based on the virtual copying of networks approach proposed by Mittal, et al. (1986) and the general copy-on-write technique discussed by Fitzgerald and Rashid (1986). More particularly, it was proposed by McCall (1991/92) for application to hierarchical networks of domain rationale in PHIDIAS. In HERMES, the perspectives mechanism has been expanded and generalized so that all information (e.g., graphics and other media, as well as definitions of language expressions) is accessible relative to the perspectives.

There are two parts to the perspectives mechanism. First, there is a hierarchy of defined perspectives that is maintained as a network of (context) nodes and (context) links. Second, every link in the hypermedia database contains lists specifying which perspectives may or may not be active for the link to be traversed. The question as to what perspective is the "active" one at any given time is answered by reference to a value maintained by the HERMES application.

The hierarchy of perspectives is quite simple. It looks much like the nodes and links pictured in Figure 9-10 above. When a new perspective is defined by a user through a dialog box like that in Figure 9-9, a new context node is created. It is linked to the context nodes it inherits from by a simple context link. As discussed in Chapter 8, context nodes and links are like regular nodes and links except that they have no node kinds or link types. Context nodes have just their names and their links to other contexts. Like any node in HERMES, they can be time-stamped and they can be linked to annotations or other attributes. This linking can be used for documentation or to implement security systems that restrict movement from one perspective to another. However, in the normal HERMES system all information can be accessed by all users; it is organized in perspectives to support timely access. Traversal of the context hierarchy is similar to normal hypermedia traversal, but it has been optimized for efficiency.

Links in HERMES consist of multiple sublinks between a given pair of nodes. Each sublink maintains four items related to the perspectives mechanism: (1) the original context in which the link was created, (2) a list of added contexts in which the link can also be traversed, (3) a list of deleted contexts in which the link should not be traversed, and (4) a "switch" context to which the active perspective should be changed when the link is traversed. This information supports ten methods for the virtual copying of nodes, links, or hypermedia networks, as discussed in this section.

When the system wants to traverse a link, it tests to see if any of the link's sublinks can be traversed. The test proceeds as follows: (a) If the currently active perspective or any of its inherited ancestors matches a context on the deleted list (3), then the sublink cannot be traversed. (b) If the currently active perspective or any of its inherited ancestors matches the original context (1) or a context on the added list (2), then the sublink can be traversed. If there is a switch context (4), then when the link is traversed the active perspective must be changed to the switched context. The inherited ancestors are checked through a breadth-first recursive search with a check for cycles in the inheritance network. Conflicts from multiple inheritance have no consequence since there is no content to the context nodes, the first match halts the search, and alternative paths are equivalent.

Recall from Chapter 8 that named nodes are separated from their contents. So, links connect pairs of named nodes and they also connect named nodes with their content. Because the contexts are checked during link traversal, they control both which named nodes are connected in the active perspective and what contents go with a given named node in that perspective. This is why it is possible for a given named

node (e.g., the language expression named "`too near`") to have different contents (different definitions) in different perspectives.

The following suite of ten methods implement the creation, deletion, and modification of links, nodes, and contents relative to perspectives. They are defined as object methods for VCopy nodes (see Section 8.2). They provide the following functions:

1. Copy the information from one context (perspective) into another.
2. Delete one node in a context that descends from another context.
3. Modify one node in a context that descends from another context.
4. Delete one link in a context that descends from another context.
5. Modify one link in a context that descends from another context.
6. Physically copy one node from one context into another context.
7. Virtually copy one node from one context into another context.
8. Reuse a subnetwork from one context in another context.
9. Virtual copy a subnetwork from one context into another context.
10. Lazy virtual copy a subnetwork from one context into another context.

Method 1: copy an entire context. Given the foregoing apparatus, the ten virtual copying methods can be explained. The simplest is to just copy all the contents of one perspective into a new perspective. For instance, Archie wanted to make his own copy of everything that was visible in Desi's perspective. This is done by defining the new perspective and having it inherit from the old one. Then, when the system checks a link to a node or to a node's contents when the new context is the active one, it will start by trying to match the new context and then will try to match its ancestors. The old context is its ancestor, so a match will be found when the new context is active if and only if it would have been found when the old context was active. Therefore, the same nodes and contents will be visible to Archie as to Desi. Of course, once Archie starts adding, modifying, or deleting nodes or links in his perspective, sublinks will start being labeled with Archie's new context and this will introduce changes between the two perspectives.

This approach is called *virtual copying* because the effect is to make it seem that all the information from one perspective has been copied into the other perspective. However, nothing has in fact been physically copied in the database. In fact, no nodes or links have been changed at all, except the addition of the new context node and its links in the perspectives inheritance hierarchy. Physical changes to the nodes and links only take place when there are changes made to the virtual copies. That is, if Archie deletes or modifies a node or link that was originally created by Desi, then changes must be made to ensure that the modifications or deletions show up in Archie's perspective but not in Desi's. On the other hand, if Desi changes something that has not been altered by Archie, then these changes should show up in

both perspectives. Under many circumstances, his last point is an advantage of virtual copying over physical copies—in addition to the great savings of memory and time.

The next four methods are for handling deletions or modifications to virtual copies in a descendant perspective.

Method 2: delete a node in a descendant context. To delete a node, simply add the name of the current perspective to the delete list of the sublink. For instance, to delete in Archie's perspective a named node or a content node that was virtual copied from Desi's perspective, leave its original context (Desi's) alone and add Archie's perspective to the delete list of the sublink of the link leading to the node. Then when traversal of that link is attempted in Archie's perspective, the delete list will prohibit the traversal, although it will still be permitted in Desi's perspective.

Method 3: modify a node in a descendant context. To modify a node, first create a physical copy of it in the new perspective and link it with a new link labeled with the current perspective as its original context. Then delete the old node in the perspective using method 2. Suppose Desi had defined `too near` as `closest distance is less then 5 feet` and Archie modified it to `closest distance is less then 3 feet`; the result is shown in Figure 9-11.

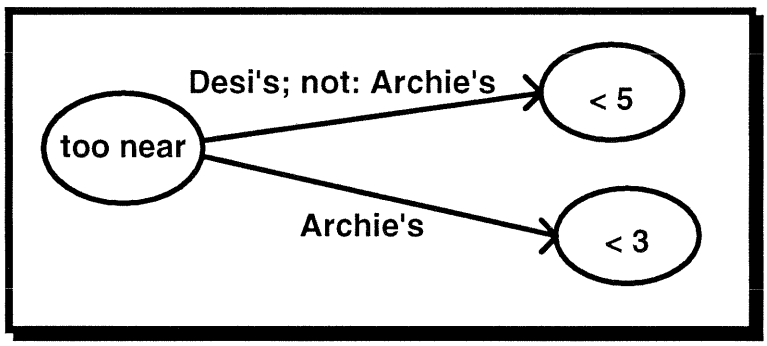


Figure 9-11. The result of modifying the virtual copy of a node.

Method 4: delete a link in a descendant context. This is identical to method 2. To make it so that a link will not be traversed in the descendent context is to make the linked node effectively deleted in that context.

Method 5: modify a link in a descendant context. This is similar to method 3, although no changes to nodes are made. Rather a new sublink of the original link is created. The original sublink and the new sublink are labeled as were the two links in method 3 (and Figure 9-11). Now there are two routes through the link to the node. One will be crossed in the ancestor context(s) the other in the descendent context.

Recall that display attributes and spatial transforms are stored in the sublinks, so which sublink gets traversed can make a significant difference in how the node at the end of the link is displayed. For instance, the node could be the graphics for a brick in a wall. If the wall consists of thousands of identical bricks, it could be made

up of thousands of virtual copies of the one graphic node, each reached by a different sublink having different spatial transforms to locate that copy in the wall. Such efficient vector graphics is a major benefit of the virtual copying scheme, although it is not a central concern of this dissertation.

The remaining methods handle cases in which one does not wish to copy an entire perspective, but rather just a single node of a linked network of nodes.

Method 6: physical copy one node into another context. One can always simply make a physical copy of a node from one context to another. The old node is not changed. The link from the new copy of the named node to the new copy of its content is labeled with the new perspective. This option can be used in place of virtual copying in cases where one does not wish the copy to change if its original prototype is changed in its old perspective.

Method 7: virtual copy one node into another context. This method uses the list of added contexts in the sublist. To copy a node from, say, Phyllis' perspective to an independent perspective, like Sophia's, simply add Sophia's perspective to the add list of the link between the node and its content. (The perspective hierarchy in Figure 9-12 is assumed in this and the following methods.)

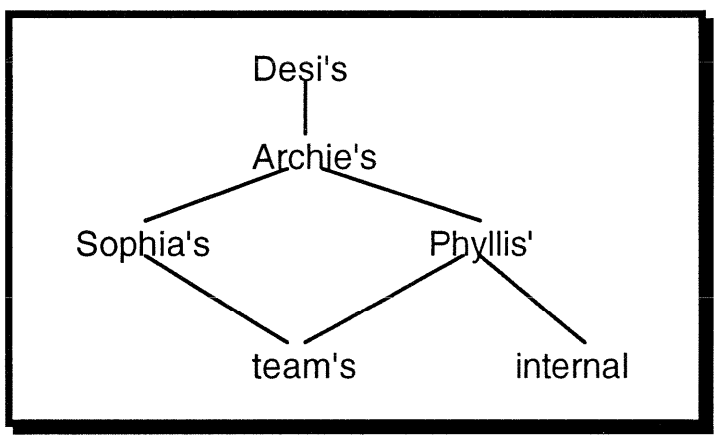


Figure 9-12. An illustrative perspectives hierarchy.

Method 8: reuse a subnetwork in another context. This method uses the switch context in the sublist. To virtual copy a network of nodes in, say, Phyllis' perspective so they can be traversed in an independent perspective like Sophia's, first create a new context and have it inherit from Phyllis' context. This context need not even have a name; since it is used internally, it can always be referenced directly by its internal object id. Although the number of such internally-defined contexts may proliferate with extensive virtual copying, they will never appear to the system users. Then create a link from where you want to enter this subnetwork in Sophia's perspective to the first node you want to traverse to in Phyllis' perspective. This link will have Sophia's perspective as its original context. Define its switch context to be the new internal context as in Figure 9-13. Then, what happens when you traverse

this link from Sophia's perspective is that your currently active perspective changes to the internal context. Since this context is a descendant of Phyllis' perspective, you can now freely traverse the subnetwork.

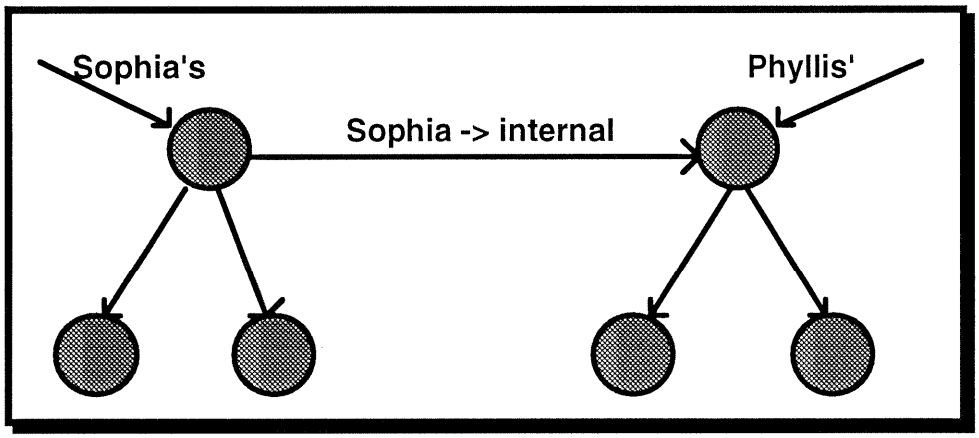


Figure 9-13. Switching contexts to traverse a subnetwork.
The network of nodes on the left is visible in Sophia's perspective; that on the right in Phyllis'. The link between them can be traversed in Sophia's perspective, but it switches the active perspective to an internally defined descendent of Phyllis' perspective so that the right-hand network will be visible.

Method 9: virtual copy a subnetwork into another context. This method is an extension of method 7 and an alternative to method 8. The disadvantage of this method is that it is more computationally intensive to set up. Whereas method 8 involves just adding an internal context to the perspectives hierarchy and creating a single new link with the switch context, method 9 involves inserting the current context into the add list of a sublist in every link of the subnetwork. If the subnetwork has thousands of nodes linked together, this can be an expensive operation, involving many disk accesses.

Method 10: lazy virtual copy a subnetwork into another context. This is a variation on method 9. Instead of traversing the entire subnetwork and inserting the current perspective into all the sublink add lists at once, only the link to the first node is treated. All links coming out of this node are then marked for future treatment. As each of these links is traversed in the future during normal operations, those links are treated and the links further down in the subnetwork coming out of their nodes are then marked for future treatment. This spreads out the costs and delays them until they are unavoidable. A further advantage is that prior to virtual copying each of the nodes as they are encountered, the user can be queried if the node should actually be included in the new perspective. This allows the user to browse through the network and selectively include just those nodes that are really desirable in the new perspective.

Method 10 uses the procedural attachment technique mentioned in Chapter 8. Every node in the system is capable of having an arbitrary procedure attached to it.

The nodes to be treated in the future by method 10 are marked by having the lazy virtual copying procedure attached to them. Then when they are traversed, the procedure is executed and it treats them and their further links appropriately. This is a form of delayed recursion.

The ten methods reviewed here (along with the context hierarchy and the procedure for checking links during attempted traversal) suffice for implementing the HERMES perspectives mechanism. They provide an efficient means for organizing information in over-lapping categories, such as hierarchies of personal and group viewpoints, of technical aspects, and of domain traditions. The virtual copying is also useful for efficient versioning schemes, CAD graphics, and information security systems. The following section will touch on some ways this mechanism can be used to support interpretation in collaborative design.

9.3. EVOLVING PERSPECTIVES

Supporting knowledge evolution. As knowledge in the database grows and changes, it must often be reorganized. The evolution of knowledge means that different designers are adding, deleting, and changing information in different perspectives. In a design environment without perspectives all the growth of knowledge would take place within a single, homogeneous knowledge base. When the organization of this knowledge became disorganized and contradictory it might be necessary for a reseeding process to take place. This could involve specialist programmers or knowledge engineers (that is, people other than the designers who normally use the system) to step in and impose order and consistency. They might extend some of the system functionality as well, but their main task would be to straighten out the organization of knowledge.

In HERMES, the perspectives mechanism can be used by the designers themselves to do some of the reseeding process in an on-going way. They can also use the language to extend the functionality of the system, defining, for instance, new analytic computations.

A paradigmatic task for supporting the evolution of perspectives and their knowledge is the merging of two unrelated perspectives. This was also identified as a critical task by the authors of the perspectives mechanism in the PIE system, reviewed in Chapter 7. In Section 9.1, above, the design team decided to merge the privacy critic work in `phyllis' perspective` with that in `sophia's perspective`, creating a new `lunar habitat design team` perspective. This is an example of reorganizing evolved knowledge. The new perspective might also be designated the `privacy perspective`. The point is that multiple independent efforts had created new knowledge in separate perspectives. Because the designers decided that this knowledge belonged together, they created a new category (perspective) for it and reorganized the knowledge accordingly.
Figure 9-14 shows the HERMES interface for doing this. It is similar to the schematic in Figure 9-9. Here, the new perspective is created by assigning it a name. Then

existing perspectives are chosen from a pick list (either as a sorted list or a hierarchical tree) to specify what information should be inherited. The inheritance takes place using Method 1 described in Section 9.2. In the particular scenario of Section 9.1, there was a multiple inheritance conflict in the definition of the expression, `too near`. Such conflicts are resolved through a breadth-first search of the inheritance tree. So the version of information in the most immediate ancestor perspective takes precedence. In case of two ancestors at the same level, the one named first in the dialog takes precedence. Note that this dialog allows one to review and modify the inheritance tree of existing perspectives as well as perspectives being newly created in the dialog.

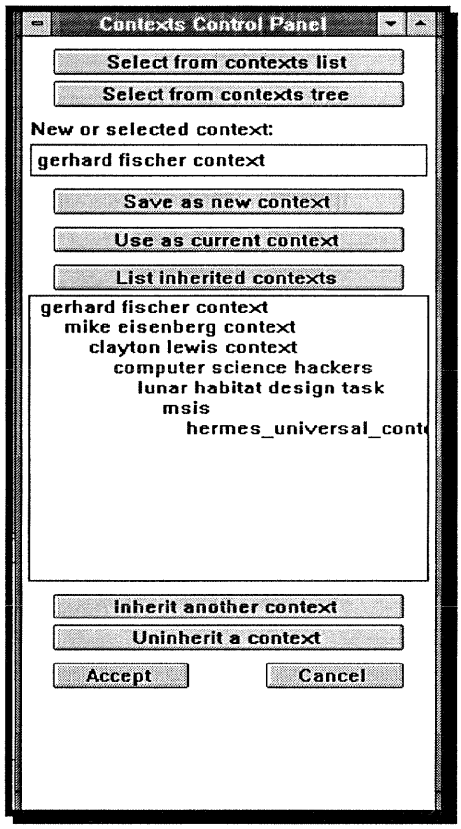


Figure 9-14. Interface for merging existing information into a new perspective.

Once the new perspective is set up, designers can browse through the information visible in the perspective and modify it. Information can be added, deleted or modified using the methods described in Section 9.2. This process of adding, deleting, and modifying applies to both named nodes and to their contents. It also applies to both individual nodes and to whole subnetworks of nodes. For instance, an issue in the design rationale could be wholly deleted or it could merely have its content changed in the new perspective. Furthermore, the networks of subissues, answers, and arguments underneath a given issue could be copied in from another perspective by one of several alternative methods already described in Section 9.2.

Of particular interest in merging design rationale and other information from different perspectives is the fact that multiple opinions can be preserved or suppressed at will. Figure 9-15 shows the same segment of design rationale as viewed in three perspectives, which inherit from each other sequentially (right to left). Two kinds of changes have been made in the subsequent perspectives: changes that overwrite the previous opinions and changes that add to the previous opinions.

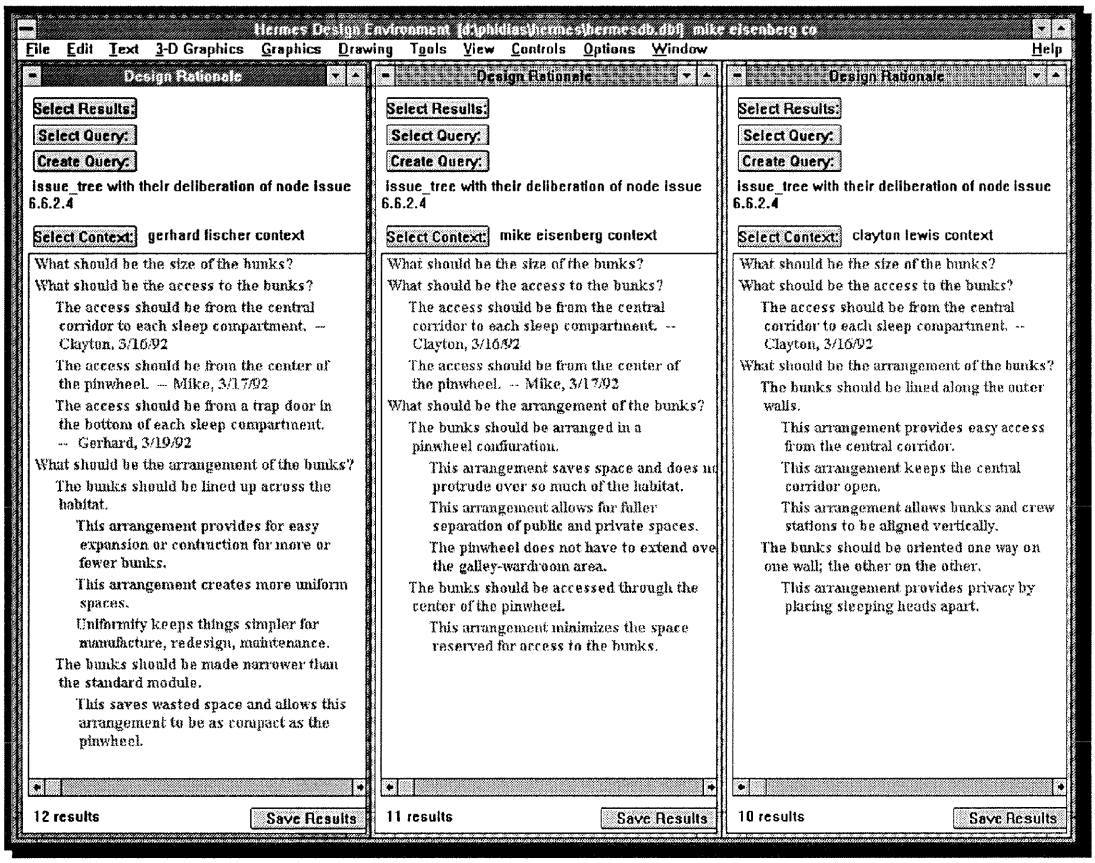


Figure 9-15. Three perspectives on a segment of design rationale.

In each perspective, the same three issues are raised. For the answer to the second issue—"What should be the access to the bunks?"—the middle perspective has added an additional answer to the original one and the perspective on the left has added a third answer to those two answers. So in the final perspective, which inherits from the other two, the three competing answers are all visible. However, the answers to the third issue—"What should be the arrangement of the bunks?"—replace each other. Here, the issue is answered differently in each perspective because the inherited answers were deleted or modified to the new answers. This shows how support for evolution of information can equally support the accumulation and deliberation of historical versions of information or the replacing and modification of information.

Another important concern for the evolution of knowledge is the need to support the demotion and promotion of items of information from a given perspective to one that is higher or lower in the perspective hierarchy. Assume that there is a hierarchy of domain traditions such as that on the right-hand side of Figure 9-10. From most general to most specific there are the perspectives: habitats, space-based habitats, and Mars or lunar habitats. Suppose that a particular network of design rationale had been formulated by a designer working in the `space-based habitat perspective` at some point in the past. In reviewing this information within the `lunar habitat design team` perspective the design team members use the language constructs discussed below to determine which context this rationale is defined in and they decide as a group that the rationale is general enough to be placed in the `habitats perspective`. Alternatively, they might decide that some other rationale is too specific to the moon and should be located in the `lunar habitat perspective`. By clicking on the top node of the subnetwork of rationale, they can bring up an interface dialog box (see Figure 9-16) that suggests a number of options for reorganizing the location within the perspectives hierarchy of the node and/or the network of nodes connected to it. These options are implemented with the methods described in Section 9.2.

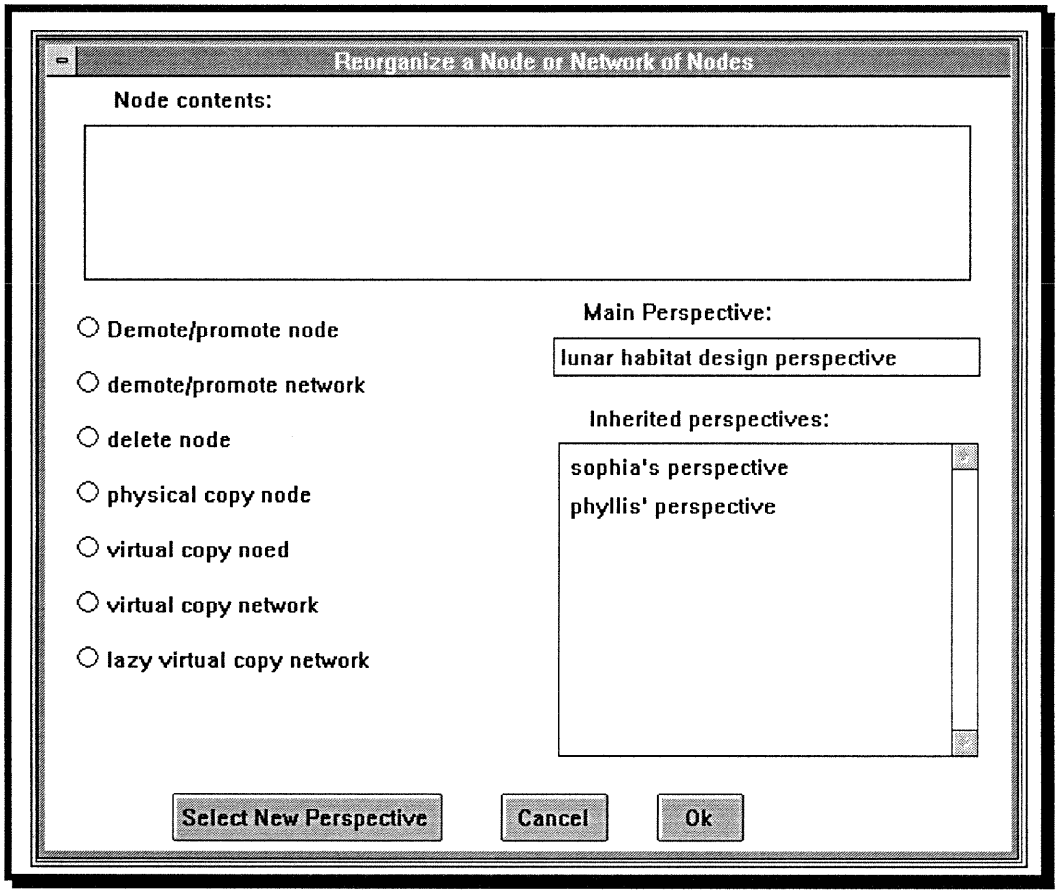


Figure 9-16. Interface for demoting or promoting a node or subnetwork of nodes.

Browsing perspectives. The perspectives mechanism simplifies the task of locating information in the rich knowledge base of an evolving design environment by partitioning the knowledge into useful categories. However, it also adds to the complexity of finding information because the knowledge being sought may not be visible in the current perspective even though it exists in the system. It may not be obvious what perspective to look in. Support must be provided for searching the network of perspectives and for browsing the knowledge available in the different perspectives.

LHDE provides a simple browser with an indented outline representation of the hierarchy of perspectives or a sorted list of the perspectives names as part of the interface for perspectives selection and new perspective creation. This may be adequate for people who are only interested in a handful of perspectives whose names they recognize. It may also suffice as long as the hierarchy makes intuitive sense, perspectives have descriptive names, and knowledge is distributed among the perspectives in a clear and systematic manner. As the knowledge base evolves, extended by multiple users, these conditions will likely not persist. Of course, users can switch to different perspectives and explore the information there with display queries and hypermedia navigation. Also, more sophisticated graphical browsers can be added to the system interface to better represent the network of perspectives.

The HERMES language also offers a more flexible and expressive solution to the problem of browsing the perspectives hierarchy and the knowledge bases in the various perspectives. As discussed in the next chapter, the language syntax falls into three primary classes: DataLists, Associations, and Filters. Each of these classes supports the formulation of expressions providing information about perspectives or contexts. (a) One can produce DataLists of objects that are visible in some arbitrary context other than the current active perspective. (b) One can list context information associated with a given object in the database. (c) One can filter a list of contexts in terms of their inheritance relations to other contexts or in terms of what objects are visible within them. This provides a useful suite of language functions for browsing the perspectives and exploring how they partition knowledge. Examples of these functions will now be given.

(a) The first function allows one to, in effect, switch perspectives within the evaluation of a language expression. For instance, if Phyllis wants to see what habitats are visible from Sophia's perspective then she can request a display of the following DataList:

```
habitats in sophia's perspective
```

This produces the same effect as if she had first switched contexts and then evaluated the expression, `habitats`. The same function allows Phyllis to apply her privacy critic to the habitats in Sophia's perspective rather than in her own:

```
privacy gradient catalog of habitats in sophia's
  perspective
```

By including this capability in the language, it can be used as part of a complex computation that may involve several context switches. Once defined, such a computation can be given a name and subsequent users of the expression do not have to worry about doing all the switching or remember what nodes are in which contexts.

(b) The second language function related to perspectives provides a special report on the context information associated with an item or a list of items. For each item, it provides the original context that it was defined in, the list of all added contexts in which it also appears, the list of all deleted contexts in which it does not appear, and the optional switch context. (Only named—user-defined—contexts are listed, not internally defined ones.) This way, one can find all the perspectives in which a given item is visible. In the following example, the `contexts` Association is applied to the result of a query:

```
contexts of habitats in hermes_universal_context
```

This example uses the function discussed in the previous paragraph to first switch to the special perspective, `hermes_universal_context`. This special perspective allows all knowledge in the database to be visible: it by-passes the context checking. So first all the habitats in the system are found, and then their context information is displayed.

(c) The third language function defines three Filters for lists of contexts. These filters allow only the contexts to be listed that inherit from a given context, are inherited by a given context, or allow a given item to be viewed. The following expressions illustrate the use of these three Filters:

```
contexts that inherit from desi's perspective

contexts that are inherited by archie's perspective

contexts that view more than five habitats
```

These expressions allow one to explore the structure of the perspectives hierarchy and of the way it organizes knowledge.

Perspectives fill in the layered architecture. Users of a design environment with a perspectives mechanism can build new structures for partitioning the knowledge base as it evolves. Thereby, the inheritance network of perspectives provides a mechanism for end-users to extend the effective structure of the layered architecture of the system. As discussed in Chapter 7, there is a gap (transformation distance-2) in the traditional design environment architecture (e.g., in JANUS and PHIDIAS) between the seeded representations of situations and the concrete task that is addressed during a given use of the system.

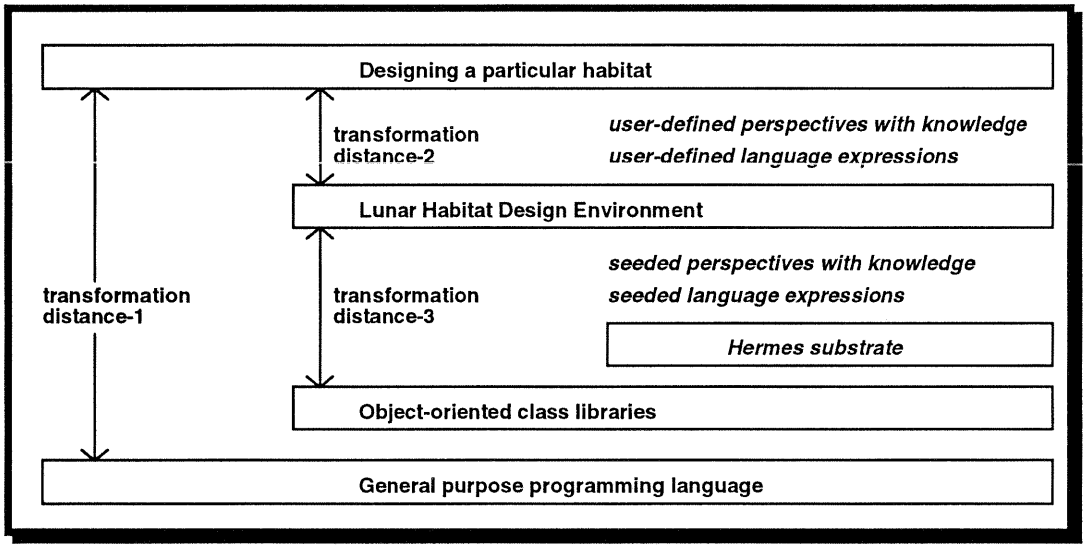


Figure 9-17. The layered architecture of design environments and HERMES. This figure extends Figure 7-2 in Chapter 7.

As shown in Figure 9-17, this gap is much smaller than that between the implementation programming language and the actual task domain, but it is not negligible.

In addition to providing palette items, catalog examples, and design rationale for the general problem domain, the seeded knowledge base in HERMES can partition this knowledge in a hierarchy of perspectives. Some of these perspectives can include knowledge that is specific to certain concrete tasks. This mediates between the general domain knowledge and specific tasks. In addition, end-users can extend the hierarchy to close the gap between the generic domain knowledge and novel tasks that arise. The extensibility of the perspectives hierarchy allows the gap to be narrowed as much as is needed to support interpretation in design by eliminating gaps in understanding that cause problems. As problems and knowledge evolve, the perspectives hierarchy can evolve under end-user control to meet the new demands and fill the shifting gaps.

In Chapter 10 it will be argued that the HERMES language can also be used as an extensible mechanism for end-users to progressively fill in the gap in the layered architecture. Definitions in the language exist within perspectives, so these two solutions work in tandem. Together, the HERMES substrate, its perspectives, and its language allow the major gaps in the layered architecture to be filled in to an arbitrarily fine degree and in an end-user extensible manner. Figure 9-17 illustrates this. From left to right in the figure are the original transformation distance between a general-purpose programming language and a task, the two problematic gaps in the traditional layered architecture of a design environment, and the fully layered architecture supported by HERMES.

Many of the features discussed in this section were originally suggested by lunar habitat designers and other NASA employees who have reviewed versions of HERMES. They have responded very favorably to the potential of the perspectives mechanism—as well as the hypermedia and language—to meet their everyday needs as designers facing complex, innovative, collaborative, knowledge-based tasks. To really know the extent to which the perspectives mechanisms can be used tacitly under realistic conditions will require extensive interface refinement and workplace testing. However, it seems plausible that the perspectives mechanisms can be effective in letting the computer manage a significant amount of the complexity of knowledge organization behind the scenes of the task at hand in which the designer is immersed.

CHAPTER 10. A LANGUAGE FOR SUPPORTING INTERPRETATION

The language presented in this chapter is designed as an integral part of the active computer support of human interpretation in design. It is structured for maximal plasticity so that designers can create and modify terms that express their ideas and their interpretations of their developing designs. At the same time, it must serve as a programming language used to instruct the computer in what computations to make. As part of a hypermedia substrate for design environments, it needs to provide expressive functionality useful for building user interface components and for exploring the hypermedia database.

If one thinks of a computationally active medium for design as incorporating a variety of "agents" that respond to events by computing information for messages and displays, then the HERMES language must serve as a *language of agents*. It must be able to analyze information in the database—using the customized terminology that particular designers defined within their perspectives—and format the results of computations on that information for display to the designers using the system. In the people-centered HERMES system, the agents do not change stored information, because such changes are left to the direct control of the human designers.

A central question addressed during the development of the HERMES language was how to make the language appropriate to the nature of the human-computer interaction that should take place in a design environment. The HERMES language grew out of the query language of the PHIDIAS design environment, discussed in Chapter 7. The PHIDIAS language was an attempt to provide a language that was "English-like" in appearance in the hope that it could be used by designers who had only a tacit understanding of what expressions in the language meant (i.e., what the expressions accomplished computationally). However, Part II argued that tacit understanding by itself was often insufficient; that interpretation required making some things explicit. That was one reason a language is needed at all. Designers cannot rely exclusively on pre-linguistic "human problem-domain communication" as illustrated by the JANUS system, but must sometimes be able to articulate their understanding in words. Language and explicit understanding are required to discover innovative interpretations, to share ideas with collaborators, and to create computer representations. On the other hand, explicit knowledge must be founded on tacit understanding and it is only required during creative interpretive acts, not when tacit understandings meet the needs. So PHIDIAS' approach to a tacitly understood language provides a promising alternative to traditional programming languages that require a sustained high degree of explicit awareness; but it is not sufficient by itself.

Of course, the scope of the original PHIDIAS query language was quite limited. The HERMES language extended that functionality to meet more of the expressive

needs of design environments and of the designers who use them. During this process, the evolving language was subjected to a series of programming walkthroughs (Bell, et al., 1991) to evaluate its usability for writing programs. A primary result of these walkthroughs—which are documented in detail in Appendix A—was the conclusion that significantly more support was needed for explicit understanding of computational issues. However, previous evaluations of the MODIFIER system summarized in Chapter 7 had shown that a purely explicit approach—even with significant support mechanisms in the interface—was not the answer either.

The theory of computer support from Chapter 6 suggests that *an adequate language must support a dynamic movement between tacit and explicit understanding*. (1) Routine *reuse* of expressions can be largely *tacit*. (2) Innovative *modification* requires a certain amount of *explicit* analysis. But even here, only the domain relationships and certain features of the representations need to be made explicit. Much of the computational "doctrine"[6] associated with general purpose programming languages does not need to be made explicit because it would only distract from the problem-domain concerns. Much of this can be kept tacit. The HERMES language represents an attempt to relieve the end-user of such programming doctrine as much as possible.

Relieving the end-user of technical doctrine of programming does not mean that designers using HERMES never need to worry about the explicit structure of the knowledge they are taking advantage of. On the contrary, the analysis of interpretation in this dissertation stresses the necessary role of explication in furthering normally tacit understanding. Rather, the attempt is merely made to minimize the amount of doctrine that must be learned that is unrelated to design. Designers are often predominantly visual, holistic, intuitive thinkers; the symbolic, detail-oriented, precise, mathematical character of programming language doctrine is particularly burdensome for many skilled designers.

Section 10.1 elaborates on the *principles* that have gone into the development of the HERMES language, including the necessity of supporting both *tacit* and *explicit* understanding. The uniqueness of the HERMES language is the way in which it strives to combine the problem-domain centered communicative goals of domain-specific design environments like PHIDIAS and JANUS with the computationally expressive goals of general purpose programming languages like PASCAL and LISP through this mix of tacit and explicit understanding.

Section 10.2 shows at an in-depth level how a number of the basic *mechanisms* of programming languages are available in the HERMES language in ways that require minimal explicit understanding of technical doctrine by system users: *Abstraction* is accomplished by ordinary naming, with no assignment statements.

---

[6] The term *doctrine* refers to guiding knowledge that must be understood in order to use a programming language. For instance, most general purpose programming languages require that programmers know doctrine about when and how to use iteration control structures. The programming walkthrough methodology is designed to assess what doctrine is required for a given task in a language.

Iteration takes place automatically without control structures. *Typing* is maintained by the implicit organization of the syntax options. *Recursion* is defined without explicit concern for halting conditions. *Variables* are generally avoided in favor of the application of successive operators; where necessary, deictic pronouns can be used to reference computational elements. *Quantification* operators can be applied directly to lists without use of explicitly bound variables. Other examples of the encapsulation of explicit mechanisms of computation in tacitly understandable forms are developed in Appendix B, where sample applications using them are also described. Of course, users of the HERMES language need to learn doctrine specific to the use of this language, but that is at a higher level of representation (closer to concerns of the problem domain) than doctrine for a general purpose programming language. Appendix C defines the complete syntax and semantics of the HERMES language.

Section 10.3 illustrates the use of the HERMES language for defining *interpretive critics*. Interpretive critics provide a final example of the synergy of HERMES' support for interpretation, exploiting the combination of the integrated substrate, perspectives, and the language. First, the *critics* from JANUS are redefined in the HERMES language. Then, the *privacy critics* from Chapter 9 are analyzed computationally. A number of the mechanisms discussed in Section 10.2 are shown at work here. This spells out in some detail one way in which HERMES can respond to the challenge from back in Chapter 3, to represent in a computer system Desi and Archie's concerns about privacy. The advantages of the HERMES approach are: definitions are made at a higher level of representation, the definitions can be more expressive, and alternative definitions can be organized in different perspectives.

10.1. AN APPROACH TO LANGUAGE DESIGN

The HERMES language is the result of following several principles arising from the theory of computer support and the review of design environment needs in Part II. These principles are:

1. Support a mix of tacit and explicit understanding.
2. Provide a people-centered approach.
3. Meet the needs of design environments.
4. Offer an end-user language for non-programmers.

This section will discuss how the HERMES language adheres to these principles.

1. Support a mix of tacit and explicit understanding. The HERMES language stresses different priorities than traditional computation-centered language designs, resulting in a different set of design decisions and a different character to the language. The contrast between the HERMES language and the FP functional programming language proposed by Backus (1978), on which the HERMES language is formally modeled, or the PASCAL procedural language in which it is implemented makes this point graphically.

Here is a task like that posed for the programming walkthroughs reported in Appendix A: Suppose you have a hypertext database with issues in nodes of two types: `question` and `problem`; answers to the issues in `answer` nodes connected by `answer` links; and arguments for the answers in `argument` nodes connected by `argument` links. (See Figure 10-1.) Now, you want to know: *which issues have four or more arguments associated with them* (via their answers).

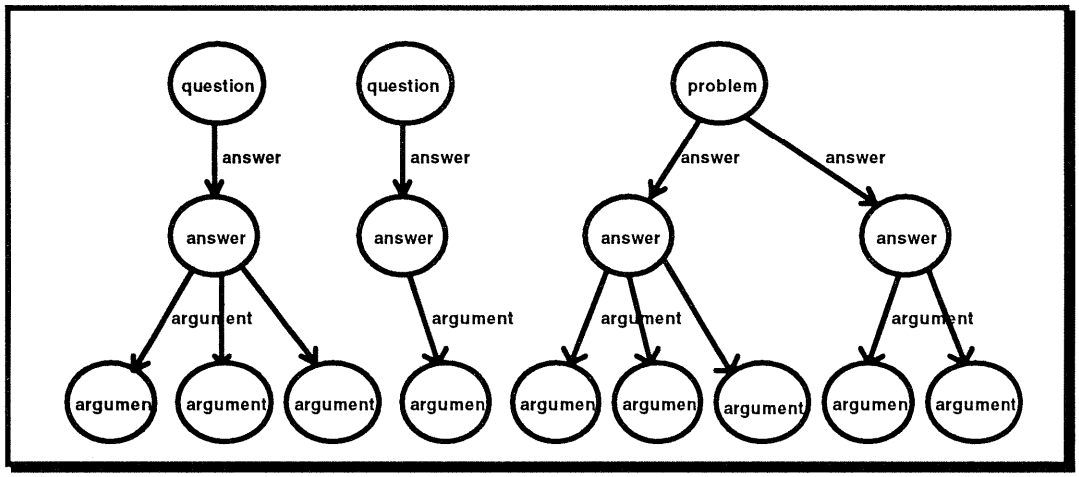


Figure 10-1. A database of design rationale.

This could be accomplished by first defining `issues` as `questions and problems` and defining `rationale` as `arguments of answers`. Then you could define the query using these newly defined terms as:

```
issues that have more than 3 rationale
```

Notice how this "program" in the HERMES language is a simple statement in domain terms of the desired results. All the computations that the computer must carry out to produce the query results are implicit: iterating through all the `questions` and `problems` in the database, following each of their `answer` links (if any) and their `argument` links (if any), accumulating and counting their `rationale` nodes, filtering out all the `issues` that do not fit the condition.

The statement of the query in the HERMES language contrasts with its formulation in other programming languages. First, it has an appearance that seems easier for non-programmers to understand tacitly than its equivalent in FP, even though the HERMES language is formally close to FP:

```
α(have-Q-R [>3, rationale]) ° issues
```

In this FP declarative statement, much of the computation has been explicitly symbolized in abstract mathematical formalisms of application, composition, and comparison. Even so, the functional approach of FP using successive composition of operators—which HERMES borrows from FP—avoids the step by step detail of a

procedural language. The following procedural pseudo-code shows what this query would require in a procedural language, and in fact how it is computed (behind the scenes) even in the HERMES system:

```
begin
list0 := empty list;
list1 := all nodes with Kind = question;
list2 := all nodes with Kind = problem;
list3 := list1 append list2;
for i = 1 to size of list3 do
   list4 := empty list;
   for each link type from node do
        if link type = answer
             then for each linked node do
                  for each link type from node do
                       if link type = argument
                            then add node to list4;
   if count of list4 > 3
        then add node to list0;
return list0;
end;
```

Traditional general purpose programming languages are based largely on mathematical models of fully explicit expressions. To name some of the most popular historical languages, FORTRAN is based on algebraic formulas, COBOL on business arithmetic, APL on matrix algebra, and LISP on symbolic logic. Assembly languages are necessarily closely modeled on the architecture of computer CPUs. Most recent languages are derived from combinations of these prototypes. Even Backus' FP language, which is an attempt to break away from the von Neumann and lambda-calculus models, is strongly influenced by APL—particularly in its outward appearance to the human programmer. All these languages have been developed under severe pressure to optimize usage of computer resources (memory locations and cycle time). This has led to the following problem: programming languages are necessary for empowering people to communicate with and through computers; however, the way in which the predominant languages are closely based on mathematical models make them difficult for many people in many situations to use to express themselves.

Natural languages that societies have historically developed for their own expression and interpersonal communication needs have very different characteristics from these programming languages. They tend to support informal, tacit, contextual, situated expression. Thus, they are very dependent on human intentional comprehension of semantics and communicative intent. They feature a highly

generative phrase structure and huge vocabularies that evolve historically. They develop under the constraint of cognitive ease for the human speaker and vocal brevity (Grice, 1975).

Now that computer resources are several orders of magnitude less scarce than in the past while human cognitive resources are being overwhelmed with the complexities of the information age, it seems time to consider designing programming languages or end-user languages in which some of the burdens are shifted to the computer. That is, while a mathematical basis for languages may be important for theoretical reasons, practical considerations of supporting the needs of users without burdening them unnecessarily suggest that the logical computational structure of the language should often be kept tacitly hidden in favor of a higher-level structure close to the user's explicit concerns. Computers increasingly have the power to manage the translation between these levels to relieve the user of that burden.

The goal of the HERMES language is to make communication with the computer system cognitively and interpretively easier for people. It tries to do this by hiding many computational details, leaving it up to the computer software to take care of them. It allows designers to build their own vocabulary incrementally, using terms familiar from their domain of work. The vocabulary can grow through a history of use, with different people developing different meanings for terms (in their own perspectives) and sharing these meanings (in common group perspectives). The language starts out with a shared basic vocabulary, established as a seeded vocabulary by the design environment builders. Terminology in the language can be reused and modified by subsequent system users, just as natural language words can take on new metaphorical meanings. The language is intended to support interpretation, explication, and interpersonal communication, not just formulaic statement. Section 10.2 will detail what is meant by hiding the computational structure of expressions.

2. Provide a people-centered approach. The slogan of a "people-centered" approach means that the computer system should be controlled by the people with whom it interacts at the points where judgmental decisions must be made that involve the exercise of intentionality. The HERMES language is designed to empower people to express their interpretations and judgments in ways that can affect the computer's actions and that can also be communicated to other people. By making the design environment programmable, the language lets designers using the system determine how displays, analyses, and critics used in the active computational environment are to be defined. Terms used in the definitions of these displays, analyses, and critic rules can be defined and modified by designers in accordance with their own interpretive perspectives.

"People-centered" also means that the system interacts with people in ways appropriate to human cognitive (interpretive) styles. HERMES features a language for designers (rather than trained programmers) to use. *The language is defined as a series of subset languages to facilitate learning by new users.* This way, people can work with the language at a level that is comfortable for them. When they need more explicit control in defining revised expressions to capture their precise interpretations, they will have a relatively easy path to exploring language features that are new to

them. They can simply move to the next stage of the language in a particular area of the language. For instance, if they need a new definition of a complex expression, they can expand the beginner's dialog box of syntax options to see the additional intermediate options or they can view the definition of an existing expression and modify it gradually. (An interface for doing this is discussed under point 4, below.)

First it should be noted that previously defined terms and expressions are used by designers most of the time. These can be simply selected from lists of relevant terms, even by a *novice*. Then there is a *beginner*'s version of the language that is similar to the PHIDIAS language, which proved easy to use for non-programmer users. This level of the language suffices for defining or modifying most common terms and queries. An *intermediate* level provides access to virtually all features of the language except those related to graphics. Finally, an *advanced* level can be used for graphics-related tasks, like defining interpretive critics. Most system displays and component interfaces are defined in the language, so they can be modified through use of the language. It would be possible to add a fully general *programming* level to the language by providing a programming language interpreter that could treat the syntax options of the HERMES language as predefined functions. This has not been done because the research focus of the HERMES language is to support interpretation in design and to make a language as interpretable as possible for non-programmers. This goal probably does not require a computationally complete language. So, the following levels of usage are supported by the HERMES language:

Novice. Even without defining any new expressions in the language, a novice can still use most of the HERMES system in a flexible way. It is, for instance, possible to define new link Types and node Kinds, although one cannot yet define new computed expressions that refer to them. One can also use all the previously defined (seeded) expressions in the language: DataLists, Predicates[7], conditions, queries, critics, etc. Thus, it is possible to define conditional nodes, conditional links, or virtual structures (queries embedded in nodes) without writing new expressions in the language.

Beginner. This version corresponds roughly to the original PHIDIAS language. It allows the user to define expressions, displays, and critics incorporating Filter clauses. With only 15% of the number of options of the full language, the Beginner syntax provides a good learning experience for most of the features and conventions of the HERMES language. This version of the language features Input Associations, a subset of Associations useful for eliciting design rationale or argumentation. For instance, if an Input Association, `deliberation`, is defined as `issues with their answers with their arguments`, then it can be used to control data entry. A special interface feature is designed to create new nodes following the patterns of user-defined Input Associations. Using the definition of `deliberation`, it will prompt for the text of an `issue` to be entered; then it prompts for one or more

---

[7] The definition and use of Predicates, conditional nodes, and virtual structures is described in Appendix B. DataLists and other syntax categories are defined and illustrated in Appendix C.

`answers` to that `issue`; for each `answer` it prompts for one or more `arguments`. Recursive Input Associations can also be defined that prompt for whole trees of data to as much depth as the respondent is willing to go.

Intermediate. The intermediate version of the language expands the computational power of the beginner version, without, however, including the complications introduced by graphics. This corresponds roughly to the level of complexity implemented for the version of the language used in the programming walkthrough (Appendix A) and in the academic advising application (Appendix B). Here, the simple data-entry Input Associations become a subset of the more powerful and complex Associations. Predicates are a modification of Associations to hide some of their complexity when displayed.

Advanced. This version adds the multi-media capability and more sophisticated programming options. Now all the computational capability of the language can be applied to nodes of any medium (e.g., vector graphics, sound, video, and bitmaps). This is necessary for implementing displays or critics that take into account graphical information (distances, spatial relationships, adjacencies, volumes, etc.). [This level of the language has been designed (see Appendix C), but not yet fully implemented.]

Programmer. Ultimately, one might want to give a user full programming power. In a research prototyping environment, one could simply hand over the source code. In a LISP environment, one can allow the user to enter programs as data that are then interpreted. However, in realistic cases where the source code is not made available and where speed is too much of a concern to use an interpreted language for building the system itself, other mechanisms must be developed. HERMES provides a form of procedural attachment implemented via dynamic link libraries (DLLs) in WINDOWS. This lets the user define a certain number of pre-named functions, using the full power of object-oriented PASCAL or C++. These functions can then be attached to nodes or links in the hypermedia database (see active objects in Section 8.2) and referred to by expressions in the HERMES language. [This level of the language has not been explored extensively, but is meant to be suggestive as a response to the limits of programming complex algorithms in the HERMES language.]

These levels of the language extend the idea from JANUS of a layered architecture, as discussed in Chapter 7. The layers of the language fill in the two gaps that appeared in Figure 7-2: the transformation distance-3 between the system building environment (LISP) and the design environment (JANUS), and the transformation distance-2 between the seeded design environment and the actual task domain (laying out a particular kitchen). The first of these gaps is filled primarily for system builders who are constructing a new design environment or adding new components to an existing one. When a design environment is built on top of the HERMES substrate, new components take advantage of the substrate functionality, including the language. As shown in Chapter 8, many functions are implemented as windows or buttons that evaluate expressions defined in the HERMES language. That means first of all that functionality can be defined using higher level terms in the

HERMES language without the system builder needing to work at the lower level implementation language. It also means that future end-users can revise the way those functions work by modifying the definitions of the terms used in the HERMES language, which is available to them at run-time as well. The second gap is filled primarily for designers using a design environment built on HERMES. They can simply use the terms, displays, and critics that have already been defined. If they need to modify something, the Beginner version of the language is available. If this is not sufficient, they can successively try more advanced versions of the language. This provides almost a continuity of layers to support a range of understanding from tacit work in the problem domain to explicit software programming in the underlying programming environment. (See Figure 9-17 in Section 9.3.)

3. Meet the needs of design environments. Chapter 7 cited the idea of programmable design environments proposed by Eisenberg and Fischer (1992). It was claimed there that HERMES could be viewed as the first implementation of this notion. In fact, the design and development of HERMES was driven by the desire to include programmability as a central feature of a design environment in order to empower designers to define, control, and extend the computational power of the software system in which they carry out their design work.

The desire to have the language refer to, analyze, critique, and display all the varieties of knowledge and representations in a design environment—including information from previous designs in a catalog, palette items for use in new designs, specification decisions, design rationale, domain distinctions, critic rules, etc.—forced the system to become more and more integrated. As the power of the language was extended from its original restriction to design rationale (in the original PHIDIAS query language), more of the knowledge was represented as hypermedia nodes that could be linked in one integrated knowledge base. New forms of knowledge were also added. For instance, conditional expressions could be defined to implement conditional links, conditional nodes, and critics. The increased generality of the system made it easy to add new media, like bitmaps, voice, and video as well.

As the language grew in range and power, the number of its syntax options in the language increased rapidly, despite extensive efforts to generalize and simplify the syntactic structure. In the end, the number of options increased by an order of magnitude. Most of these syntax options (those called "simple" options) directly reflect elements of the multimedia knowledge representation substrate. Many other syntax options (called "computed" options) define combinations of the primitives that are needed for useful computations. The appearance of expressions in the language is dominated by user-defined terms: names of objects, link types, node kinds, names of defined sub-expressions. Otherwise, there are just a few "helper" words that remind people of the functionality of the options. Little is left in its external appearance of the language's computational internal nature. Thus, the HERMES language appears to be a "new" language, although it is really basically the result of adapting a stripped-down functional programming approach to meet the needs of a design environment.

Despite its adherence to the notion of a programmable design environment, the HERMES language is very different from a programmable application like

SCHEMEPAINT (Eisenberg, 1992). In SCHEMEPAINT, the language is used for creation of new objects. In contrast, the HERMES language is "non-imperative" (Schmidt, 1986). Evaluation of expressions in the HERMES language do not change state: they do not *create* anything new. They navigate through the hypermedia database and collect lists of existing objects. Of course, by means of user interface features, these lists can themselves be saved as new objects. Also, interface features can be designed that use language expressions to organize, modify, or even create objects. For instance, a design rationale prompting component in the interface can elicit and store new argumentation using the Input Association syntax options as explained in Appendix C. *The language is primarily geared to the diverse information retrieval needs of designers.*

Design environments have a variety of data retrieval, manipulation, and display needs. In a hypermedia-based system like HERMES, these needs can generally be categorized into three groups: (a) to generate lists of information, (b) to selectively choose items from lists, and (c) to navigate through the inter-connected network of the database. This corresponds to the three categories of operations that Abelson & Sussman (1985) emphasize for functional computer programs: to enumerate, map, and filter lists or streams of information.[8] The HERMES language syntax provides three primary classes of terms to operationalize these functions: DataLists, Filters, and Associations, as indicated in Table 10-1:

Table 10-1. Correspondence of language uses, operations and classes of terms.

	uses	operations	HERMES language
(a)	generate lists	enumerate	DataList
(b)	selectively choose	filter	Filter
(c)	navigate network	map	Association

(a) Many forms of lists must be *generated* (enumerated) in a design environment. In a system built on top of the HERMES substrate, virtually all displays in the user interface are constructed dynamically from such lists. The HERMES language is designed above all to provide a flexible means for defining lists of items stored in the database and useful for interpretive tasks in the represented domain. In this sense, the HERMES language is a database query language. The HERMES language is optimized for expressing queries in this environment and for retrieving the requested information efficiently in useful formats. Unlike SQL (a general purpose query language for relational databases), it is designed for an object-oriented, multimedia database in which items are linked together in hypertext style. It differs from SQL in

[8] The suggestion to interpret operations in the HERMES language as the processing of streams of information in this sense was suggested by both C. Lewis and M. Eisenberg, independently.

being non-relational and hypermedia-specific. Among the information listings available through the HERMES language are general queries and the basic displays used in design environments, such as design rationale issue-base views, catalogs of past designs, palettes of design components. An example of a DataList that computes the items for a display of some rationale created by Archie is:

```
issues that have creator archie
```

(b) *Filtering* functions of the language are important for implementing critics and for making all displays relative to design decisions encoded in specifications, constructions, or design rationale. For instance, using the language one can define a display of all catalog items that pass a Filter referring to the existence of specific palette items in a certain construction, `answers` resolved in the design rationale, or selections made in a specification listing. Perspectives provide another filtering mechanism in HERMES, allowing only nodes that are defined in the currently active perspective to be processed by the language. The two filtering mechanisms can be combined in expressions in the language like:

```
issues in context desi's habitat perspective that have
  creator archie
```

(c) *Navigation* through the hypermedia database (mapping) is also accomplished with the HERMES language. A good example of such navigation is shown in Figure 10-1 with the expression:

```
issues that have more than 3 rationale
```

Here, the expression `rationale`, defined as `arguments of answers`, navigates from each `issue` node across its `answer` links to new nodes and across their `argument` links.

The three major syntax categories of the HERMES language (DataLists, Filters, Associations) provide the three primary functions required for design environments: (a) definitions of lists of nodes, (b) expressions for filtering out nodes not meeting stated criteria, and (c) operations to traverse various kinds of associations. These support the situated, perspectival, and linguistic character of interpretation by naming representations of things in the design situation, filtering out objects for display based on viewing criteria, and providing expressions for exploring semantic associations. Objects in each of these three categories can be either (1) reused or (2) refined by combining expressions in useful ways. This defines the six primary syntactic classes; four other classes provide auxiliary terms and features. The syntactic classes are listed with brief descriptions in Table 10-2.

Table 10-2. Major syntactic classes of the HERMES language.

	syntactic class	*description*
a-1	Datalists	options for identifying hypermedia nodes.
a-2	Computed Datalists	permitted combinations of language elements that determine sets of nodes
b-1	Filters	operations characterizing nodes for selection
b-2	Computed Filters	permitted combinations of language elements that define filter conditions
c-1	Associations	links and other associations of nodes
c-2	Computed Associations	permitted combinations of language elements that determine non-primitive Associations
d-1	Media Elements	nodes of various media: text, numbers, booleans, graphics, sound, video, etc.
d-2	Computed Media Elements	permitted combinations of media elements, e.g., arithmetic or boolean computations
e-1	Pre-defined Terminology	connective terms, measurement primitives, fixed values for attributes and types
e-2	Computed Terminology	namable quantifiers and numerical comparisons

The central *syntax classes* of the HERMES language are (a) *DataLists*, (b) *Filters*, and (c) *Associations*. In addition, (d) the Media elements define several syntax classes, one for each kind of allowable multimedia content in the hypermedia database that is traversed by the language: *Character, Number, Boolean, Graphic, Image, Pen, Sound, Video, Animation,* and *ComputedView*. (e) The Terminology options provide the connective terms for joining multiple items together and for counting items, as well as certain definitions useful for graphical computations; these include three syntax classes for user-definable options: *Count, Quantifier, Measure*; and eight syntax classes that are system-defined: *Connective, Combination, Distance, Units, Dimension, Attribute, Value,* and *LanguageType*. In addition there are three hypermedia classes that are part of the syntax: *Contexts*, *NodeKinds* and *LinkTypes*. The syntax classes are divided into Simple and Computed options. The Simple options define a single operation for producing a result. The Composite options define legal combinations of applying one operation to another. This defines the operator algebra that is at the heart of the HERMES language. It is discussed below. Table 10-3 (below) provides sample options from each of the classes listed in Table 10-2 (above).

The DataList, Filter, and Association options constitute the majority of the syntax options. The Simple options are all defined as primitive operators. For instance, Simple DataLists return a node or list of nodes as their result. DataList, Filter, and Association (both Simple and Computed) evaluation functions all take a

DataList as input and return a new DataList as a result. This DataList result format acts as a stream of data items that passes through the operators to generate new items, filter out items that were there, or map from the old items to associated new items. Because of this uniform format, any of the operators can be applied successively to the results of any other operators. This allows the unlimited nesting of phrases and application of operators that makes the HERMES language highly generative.

Table 10-3. Examples of syntactic options for the HERMES language.

	syntactic class	***example***
a-1	Datalists	all database items of a specified NodeKind
a-2	Computed Datalists	items of a *DataList* that pass a specified *Filter*
b-1	Filters	items that are of a specified NodeKind
b-2	Computed Filters	items that pass *Filter1* and also pass *Filter2*
c-1	Associations	a Link Type (e.g., children)
c-2	Computed Associations	*Association1* with their *Association2*
d-1	Media Elements	a real number (e.g., 3.14)
d-2	Computed Media Elements	the total of all numbers in a specified *DataList*
e-1	Pre-defined Terminology	closest distance between two graphic items
e-2	Computed Terminology	a *Distance* is greater than a specified *Number* (e.g., too near: closest distance is less than 5 feet)

In HERMES, only certain combinations of applications are permitted, as defined by the Computed options. If the Simple options were incorporated as predefined functions in a general programming language like FP or SCHEME, then any combinations of operators could be evaluated. However, a judgment has been made in designing HERMES to limit the combinations to semantically meaningful and useful options. That accounts for the seeming proliferation of options. In fact, however, the majority of options are nothing but combinations of other options applied to each other. For these combination options, the semantics are trivially defined, as shown in Appendix C in which the denotational semantics and the corresponding implementation code for the evaluation function of one such combination option is shown. The HERMES language is a carefully *constrained* language, designed to promote relatively tacit usage by structuring the choice of operation combinations to avoid many problematic expression definitions and to guide the language user.

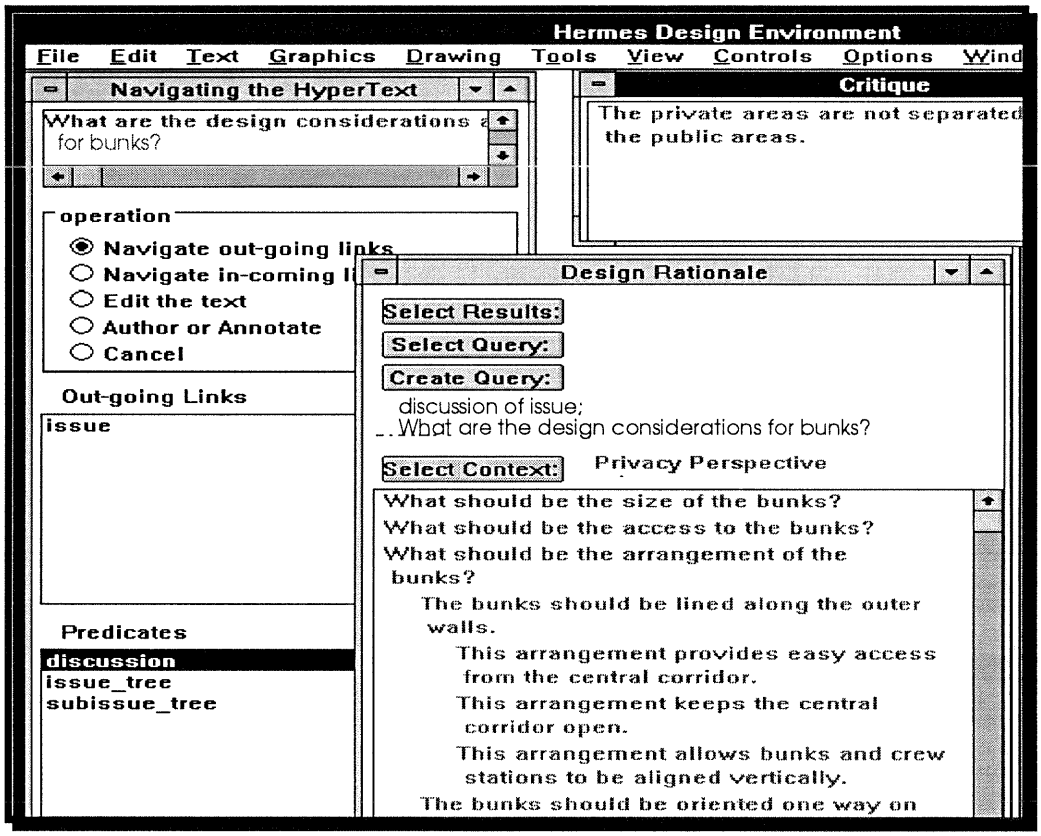


Figure 10-2. An example of hypermedia navigation.

4. Offer an end-user language for non-programmers. The use of the language in HERMES can be made appropriate for non-programmers in many ways through interface features. Some examples were already given in Chapter 8. Consider how *navigation* through the hypermedia database (mapping) is accomplished with the HERMES language. An example of such navigation is shown in Figure 10-2 (above). A textual node has been selected in a *Design Rationale* window by clicking on it. This brings up the *Navigating the Hypertext* window. The selected node has been displayed in the top of this window and the default option of "Navigate out-going links" has been chosen. The list of "Out-going Links" displays "issue", indicating that the selected node is associated with out-going links of type `issue` to other nodes. The list of *Predicates*[9] displays three terms that have previously been defined in the HERMES language; these terms are all defined with expressions that include `issue` as their initial traversal, so they are relevant to the selected node that has `issue` links. If the user had selected "issue" under "Out-going Links", then a new Design Rationale window would have been displayed listing all the nodes navigated to by following `issue` links from the original selected node. In the case shown in the figure, the user has instead selected the Predicate `discussion`. `Discussion` is defined in the HERMES language (either in the seed or by a previous user) as a series

[9] Predicates are a special form of computed Association. They are explained in Appendix B.

of link navigations beginning with `issue` links. So the display produced in the new window is an indented list resulting from the navigations defined by the language expression named `discussion`.

The example just given illustrates a number of points about language usage in HERMES. First, expressions (like `discussion`) can be reused without explicit concern for their detailed definition, particularly if their name indicates their function adequately. Second, rather complex displays can be defined relatively easily. If one wanted to, one could modify the definition of `discussion` or define a new term based on it. The new term could use Filter conditions to eliminate items selectively as well. For instance, one could define a new Predicate `bunk discussion` as: `discussion that contains 'bunk'`. Then the list of Predicates displayed in the *Navigating the Hypertext* window would include `bunk discussion` and selection of this option would result in a display that only listed items including the word "bunk". Third, language usage can be integrated into the user interface so that it feels like tacit navigation through hypermedia rather than explicit querying with a language. The use of this language need not have the look and feel of programming, even when new expressions are being defined for accomplishing arbitrarily complex computations.

When an expression must be explicitly programmed, interface support is available to reduce the cognitive burden of recalling syntax options, strict formats, expression names, or terminology spellings. As part of the attempt to reduce programming errors that would frustrate a non-programmer, a direct manipulation interface is provided for use, reuse, modification, and creation of expressions in the HERMES language. Strictly speaking, this is not part of the HERMES substrate, but belongs to the interface of a design environment built on top of the substrate. It is presented here simply to suggest one solution to the problem of supporting people to use the HERMES language with minimal cognitive overload.

* By presenting all relevant options on the computer screen at each stage and requiring expressions to be built up by choosing from these dialog options, the user is relieved of having to remember the various legal options.
* Similarly the problem of entering the precise proper format and spelling is solved. Novice programmers are particularly frustrated by punctuation and spelling errors during program input.
* The interface presents definitions of terms in a readable format. Given that expressions in the HERMES language often read much like English, it is important to avoid the impression that the system can understand arbitrary English formulations. The restriction to a visible menu of choices makes the restrictions clear and unavoidable.
* The same dialog boxes that are used for defining new expressions encourage the reuse of previously defined expressions. Old definitions can be reviewed with the dialogs to see their internal structure, and the definitions can then be modified and reused.

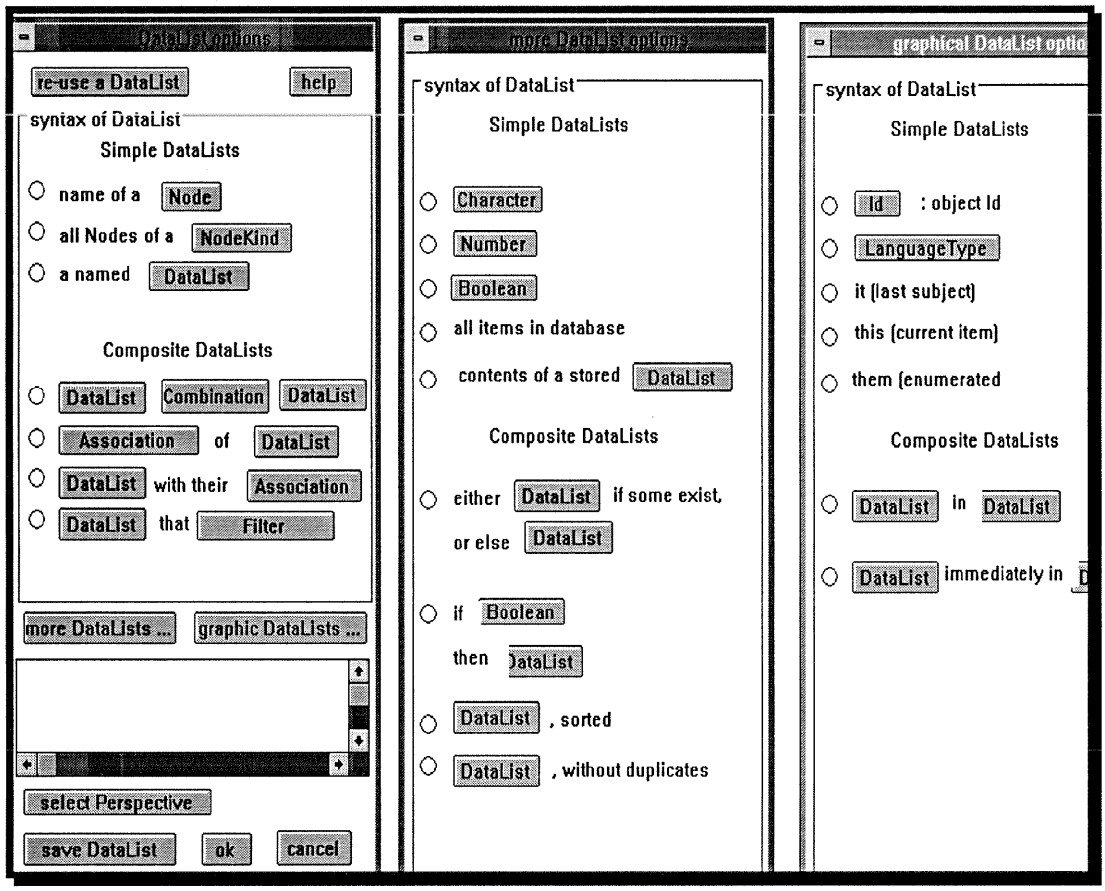


Figure 10-3. Dialog boxes for defining DataList expressions.

Figure 10-3 shows the three dialog boxes for defining a DataList expression. This is typically the starting point for defining expressions in the language, such as queries or critics. It is also possible to start with other dialogs to define conditional expressions, numerical computations, and so on. The leftmost dialog, labeled "DataList options," is the first dialog to appear under conditions in which one needs to define a new DataList. If one wants to select a previously defined DataList expression—whether defined as part of the HERMES seed, by other system users or by the current user—then a pick-list of the names of all defined DataLists is used instead.

This programming interface incorporates the breakdown of the language into a series of levels for users with different degrees of experience in using the language. This is an example of the mixture of support for tacit and explicit understanding. Even when the system user needs to make interpretations explicit and state them in the programming language, this burden is softened by providing a direct manipulation, construction kit interface for defining expressions and by providing a layered architecture of many levels of successive complexity.

The leftmost dialog presents the *Beginner's version of the syntax*. The heart of this dialog is the list of seven small circles, the radio buttons for selecting one of the three Simple DataList options or one of the four Composite DataList options. The first of these options allows the user to simply select a node from a pick-list, which will be displayed when this option is chosen. The second option defines a DataList consisting of all nodes in the database that are of a specified NodeKind. If this option is selected (by clicking the mouse on its NodeKind button), a pick-list of the names of all defined NodeKinds is displayed. The third option retrieves a DataList expression that has been previously defined and saved.

The Composite DataList options lead to other dialog boxes to define constituent parts of the composite expression. The first of these, for instance, has three buttons: "DataList", "Combination", "DataList". If this option is selected, then each of these buttons must be pressed with the mouse before the new expression can be saved. Pressing the first (or the third) of these buttons brings up another copy of the same dialog box so that the constituent DataLists may be selected. For instance, to define a DataList named `issues` as `questions and problems`, use this option and press the first button. When the new DataList dialog appears, select the first Simple DataList option and choose `questions` from the pick-list that appears. Then click the "ok" button at the bottom of the new dialog to confirm this choice. The new dialog will disappear. Then press the "Combination" button. This will bring up a dialog listing the five Combination options. Simply select the `and` option and click the "ok" button. Then define the DataList for the third button as `problems`. Now the expression `questions and problems` will show in the small window near the bottom of the DataList dialog box. Press the "save DataList" button below there. A small dialog will ask for the name of the new DataList that has just been defined. Type the word `issues` and the new expression will be part of the HERMES system.

Below the list of options in the DataList dialog are buttons labeled "more DataLists" and "graphic DataLists". These bring up the dialogs with the *Intermediate and Advanced DataList options*, respectively. They are also shown in Figure 10-3. They work the same way as the options in the first dialog, which remains on the screen and controls the overall expression definition process. This is how a user advances from the Beginner to the Intermediate or Advanced levels of the language based on their specific needs.

One more button should be mentioned in the basic "DataList options" dialog box. That is the uppermost button: "reuse a DataList". Pressing this button brings up a pick-list of defined DataLists. When one is selected from this list, the list disappears and the definition of the selected DataList appears in the dialog. First it appears in the display window in its narrative format. But it also appears in the options in the sense that the option that was used for defining it is now selected in the dialog. Pressing the buttons for that option will bring up dialogs that are also already displaying the constituent parts. This provides a way of exploring the structure of a defined expression. If anything is changed on the subsidiary dialogs and confirmed and saved,

then the definition of that expression will be modified accordingly (within the currently selected perspective).

While the foregoing description and its accompanying figure may seem complicated, that is partly because it is harder to describe this process explicitly in words, covering most of the various possible actions, then it is to use the direct manipulation interface to make selections needed to accomplish a specific task.

The HERMES language represents an attempt to push a particular approach to language design as far as possible. This is what makes the HERMES language distinctive. The approach is motivated by the theory of computer support of interpretation. It includes an effort to balance support for tacit and explicit understanding, promoting tacit activity whenever possible, even during the accomplishment of explicit programming tasks. It tries to develop an expressive and extensible programming language for people like designers who may not be experienced at computer programming. To do this, it hides many of the computational mechanisms (as described in the following section), and constrains the syntax of the language. Of course, there are trade-offs involved in keeping mechanisms hidden and in limiting options to enforce meaningfulness of expressions. The HERMES language has great flexibility and expressivity. It is infinitely generative and arbitrarily complex. But it is far from being Turing complete. There are many definitions of lists that cannot be expressed in it, but that are relatively straight-forward to program in PASCAL or LISP, for instance. Subsequent examples and analysis in the remainder of this chapter and in the Appendices should show the language's ability to formulate easily and tacitly the expressions most useful for design environments, as well as pointing out the limitations that can arise in more complex circumstances.

10.2 ENCAPSULATING EXPLICIT MECHANISMS IN TACIT FORMS

The HERMES language has been designed to minimize the amount of programming language doctrine required as explicit knowledge by people writing and reading expressions in the language. This has been accomplished by hiding a number of programming language mechanisms in the syntax options of the HERMES language so they can be used with only a tacit understanding of their functioning. This section illustrates what is meant by this approach. A number of important areas of programming language doctrine that require explicit understanding for the use of languages like LISP or PASCAL are incorporated in the syntax options, evaluation processes, and user interfaces of the HERMES language in ways that can be used without explicit understanding.

1. Abstraction in HERMES takes place by simply giving a name to an expression that has been defined; there is no explicit assignment statement. All expressions in the HERMES language can be named and the names may be used wherever the corresponding expression could be used.

2. Iteration is implicit in HERMES. For instance, in the example of displaying the `discussion` of an item from the Design Rationale window, all the `issue`

links from the item's node were followed. Normally, this would be expressed in a traditional programming language as some form of iteration (a `for`-loop or a recursion). However, in the HERMES language it is not expressed at all, but merely assumed in the declaration calling for a list of the issues.

3. Typing of expressions is enforced by restrictions on the allowable syntax options, without the user needing to be aware of this.
4. Recursion is an important technique for navigating successive links in hypermedia networks using the HERMES language, but users need not be aware of it or worry about halting conditions.
5. Variable binding occurs as a natural result of the syntax of expressions, but explicit variables are not used.
6. The syntax allows one to quantify expressions by asking for all items, checking for at least one item, and so on, but the computational implementation of these is hidden from the user.
7. Conditionals can be stated in explicit if / then format or implied through more tacit formats.

1. **Abstraction**. The importance of *abstraction* has been emphasized by many leading language designers (e.g., Liskov, et al., 1977; Cardelli & Wegner, 1985; Abelson & Sussman, 1985; Wirth, 1988). From a people-centered perspective, the importance of abstraction in definition of expressions in a programming language is that a complex expression can be hidden under an easy-to-use name that corresponds to terminology in the application domain. That means that the name can be used in a relatively tacit way once it has been defined explicitly.

To use an example from Chapter 3, it may be quite difficult to define an operational definition of privacy for lunar habitats. Such a definition must be spelled out in complete and explicit detail (see Section 10.3 below for such a definition) so that the computer system can use it. However, once defined, the definition can be stored under the name `privacy`. From then on, designers using the system can make use of the term `privacy` without being concerned about all the computational details. In fact, the term may have been defined as part of the system's knowledge seed by the software developers or by some intermediate knowledge engineer, so that the lunar habitat designers never need to be concerned with (or develop the skills to understand) the technical details of implementation. This is a case in which the analysis of interpretation provides a new argument about the role of abstraction, an old technique. Based on this argument, HERMES in fact places considerable emphasis on this form of abstraction in the design of the HERMES language, allowing every object stored in the computer memory—including all defined expressions and sub-expressions in the language—to be referred to by user-defined names.

The use of abstraction in HERMES supports extensibility of the language. Not only can link Types and node Kinds be user-defined (as they already were in PHIDIAS), but so can the language's namable terminology elements: Count, Quantifier, and Measure. For instance, a new Count term, `several`, could be defined as: `more`

`than 2 and less than 7`. By naming the expression "`several`," a user can then make use of this term without worrying about its precise definition. Of course, if another user interprets the term `several` differently, the definition can always be explored and modified through the interface to the language. Similarly, Measure terms can be added to the language, like "`set off from`": `central distance is less than 24 inches and closest distance is more than 4 inches`

In Section 9.1's scenario in which the notion of privacy is made operational, the scale of privacy values from 1 to 9 was abstracted by the definition of the following Number terms:

```
very public:       1
quite public:      2
public:            3
somewhat public:   4
neutral:           5
somewhat private:  6
private:           7
quite private:     8
very private:      9
```

These definitions allow designers to think in tacit problem domain terms rather than in the explicit quantitative terms required by the computer. The abstractions are constructed tacitly by simply supplying a name when an expression is defined; there are no explicit assignment statements in the language.

2. **Iteration**. Much traditional programming language doctrine has to do with iteration: `for` loops versus `while` control statements, recursive list processing, sequential comparisons, etc. In HERMES, there are no explicit control structures for iteration. Yet, iterating through lists (e.g., all the nodes of such and such a description, or all the links of a certain Type from a node) is ubiquitous in its central task of navigating through hypermedia. In the HERMES language, the various iterative tasks of the design environment and of its hypermedia substrate are encapsulated in primitive syntax options.

A Simple DataList can be defined by a NodeKind or LanguageType; these options iterate through the database index to retrieve all nodes of the specified category. A Simple Association can be defined by a Link Type; this option iterates through all the links of the specified Type from a given starting node. The option, `all associations`, iterates through all of a node's out-going links; `inverse` Association iterates in-coming links; `parts` iterates content links. Each of these options returns the list of all nodes at the other end of these links. The options are implemented with iteration control structures, but the user need not be aware of this.

Simple Filters also iterate through lists of nodes. They test each node successively to see whether it meets some condition. The nodes that meet the condition are returned. For instance, the expression, `arguments of answers of the bunk locations issue that are of kind pro-argument`, will be evaluated by iterating through all the nodes at the ends of the specified argument links and testing which of them are of node Kind `pro-argument`.

Several of the Number options are also implicitly iterative, returning a count of elements in a list, a minimum, maximum, total, or product of the values. One Number option even iterates through all combinations of two or three graphical objects to return a list of the distances between them. To test for an acceptable work triangle in a kitchen, a designer can simply take the minimum value of this list of distances, without needing to worry about the details of iterating through all the combinatorial possibilities if there are multiple stoves, sinks, or refrigerators in the kitchen.

3. **Typing.** The constrained syntax of the HERMES language provides an implicit typing system. Like the strong typing system of languages like PASCAL, it avoids syntactic combinations that would be meaningless or cause conflicts. However, it is enforced "behind the scenes" so users do not have to be aware of it as a typing system. Types are not declared explicitly by the user.

The Simple syntax options are categorized in the syntax classes discussed in the previous section, such as DataList, Filter, and Association. These 25 classes are the *types* of the HERMES language. A typical Computed syntax option combines terms from several of these classes. For instance, one Computed DataList option is: DataList Combination DataList. This joins any two expressions of type DataList with any expression of type Combination, like `and` or `or`.

Notice that each of the computed syntax options listed in Appendix C refers to one or more syntax classes (or types). Legal combinations of these types are defined by the options of the syntax. This is a convention of the language; it would be possible to define combinations of individual options or to distinguish between categories of options like Simple and Computed—but that would be a different language. For instance, example a-2 in Table 10-3 allows a DataList to be defined as a Filter applied to a DataList. This means that any expression of type Filter can be applied to any expression of type DataList and the result will be a legal expression of type DataList. The DataList used as a component in this definition may itself be a Computed DataList composed of several components. By applying these rules repeatedly, one can build up well-defined expressions of arbitrary nested complexity.

The set of defined legal options has been carefully designed to permit the construction of a broad range of expressions to meet the needs of people using a hypermedia-based design environment. While generality of expression has been a priority, an attempt has also been made to exclude combinations that would lead to problems for the users. Another constraint has been to keep the sheer number of options as small as possible. Of the 110 options defined, only a small number will be

used most of the time; many are for advanced techniques primarily necessary for internal use building interface functionality or for complex graphical computations.

4. **Recursion**. Recursion was already available in the PHIDIAS query language. A simple example is the definition of `issue trees` as: `issues with their issue trees`. Here, the definition recursively incorporates its own name. This is useful for navigating hypermedia networks to arbitrary depth. The evaluation proceeds from a node across all its `issue` links to new nodes, across their `issue` links, etc. The recursion terminates at nodes that have no `issue` links. This graceful termination condition is built into the implementation of the Simple Association option, Link Type. Therefore, users of the language do not need to be concerned about explicitly stating a halting condition for the recursion, a step that frequently causes bugs for novice programmers. The implementation supports what a naive user would tacitly expect, or at least what one would come to expect after having been exposed to some sample recursive definitions in the language.

5. **Variables**. The HERMES language experiments with how far a programming language can go without the use of explicit variables. Variables are perhaps the first serious barrier that most programming poses for people who are not mathematically inclined or experienced. Lack of explicit variables differentiates the HERMES language clearly from procedural languages (that use variables for iteration counters, subroutine parameters, array indices, etc.), functional languages (that use variables for lambda parameters), and logic languages (that use variables for quantification).

HERMES makes use of operator application, applying successive operations directly to the results of previous operations without need for abstract variables to relate the operations to the operands. This works smoothly in simple cases and supports tacit expectations. When expressions are nested several levels deep, the relations of what operations are to be applied to which operands can become confusing. (Several examples of this are given in Section 10.3 and in Appendix B, in which moderately complex applications in the HERMES language are discussed.) For these cases, three special "deictic variables" have been defined. These are not abstract variables, but terms that perform much the same concrete role as deictic pronouns in natural languages.

The deictic variables of HERMES are the following Simple DataList options: `that (last subject)`, `this (expression)`, and `those items`. They are used within an expression to refer to a node or list of nodes that has been previously computed. They disambiguate the application of Predicates and allow intermediate results of computations to be displayed or reused without recomputing them. Examples of the use of the `it` and `them` deictic variables will be seen in the analysis of the privacy critics in the following section. The `this (expression)` variable can be useful in defining recursive terms; `issue trees` can be defined as: `issues with their this (expression)`, where `this (expression)` refers to the term `issue trees` that is itself being defined.

It should be noted that the lack of variables is a trade-off in the design of the HERMES language. It is intended to reduce the cognitive overhead of the use of

explicit variables. However, it probably introduces the most severe restriction in the expressibility of the language for relatively complex computations, making critics like the `privacy gradient critique` in Section 10.3 and the `advice` critic in the academic advising application in Appendix B difficult to construct and comprehend. However, the language is not meant primarily to be used for building computationally complex systems, but rather for supporting the incessant reuse and modification of relatively simple definitions of terms needed for displaying, analyzing, and critiquing hypermedia representations of designs.

6. **Quantification**. The HERMES Quantifier type is provided to support quantification. As just discussed, it does not use the explicit bound variables of predicate calculus or PROLOG. Three examples show how it is used:

```
chairs that are near to at least one table in archie's
  habitat

issues that have no answers that include "bunk"

if all privacy ratings of parts of archie's habitat are
  more than quite private
```

As should be clear from these expressions, the computation of a quantity like *all* is carried out internally by the implementation of this syntax option and need not be an explicit concern of the user.

7. **Conditionals**. Conditionals are important in a design environment. They are, for instance, used for critic rules, conditional links, and conditional nodes. In addition to the standard syntax form for conditionals, `if Boolean then DataList1, else DataList2`, HERMES offers the following form: `either DataList1 or DataList2`. The second format is more supportive of a tacit approach. Its evaluation first computes DataList1. If it returns something, that is returned as the result of the whole conditional expression; if it returns no nodes, then DataList2 is computed and its results are returned for the conditional. For instance, if one wants to list the answers to an issue if there are any and give a warning message otherwise, one can define the following conditional expression:

```
either answers of my issue or "There are no answers to
  my issue."
```

The implementation of this option takes care of the checking of whether there are any results of the first part and deciding whether or not to compute and return the results of the second part.

10.3 DEFINING INTERPRETIVE CRITICS

Interpretive critics. Interpretive critics in the Lunar Habitat Design Environment (LHDE) built on HERMES play much the same role as critics in JANUS and triggers in PHIDIAS, as discussed in Chapter 7. In LHDE the critics are not active

the way that JANUS' critics were, although a different design environment built on the HERMES substrate could make use of the same mechanisms as JANUS to activate critics associated with a design unit whenever an instance of that unit is created or moved in a design construction. In LHDE and PHIDIAS II (which is also built on HERMES), critics are tied to user interface buttons to provide PHIDIAS-style triggers. Interpretive critics can be used whenever a user has them evaluated by means of any interface mechanism. That is, designers can define and evaluate interpretive critics very freely, without necessarily having them tied to design units in a palette component or to predefined buttons in a construction interface. Interpretive critics are, thus, more general than the critics and triggers of the related systems they were inspired by.

Interpretive critics are defined using the HERMES language. They can take advantage of all of the expressive power of the language. Basically, a critic is any expression in the language that analyzes the state of the hypermedia database. Typically, a critic looks for certain features in a graphical construction and displays a message or takes some other action depending on whether the feature is found or not. The message can include design rationale or examples explaining the reasoning behind the critic definition. It might, for instance, include a selection of items from the design rationale, through which the designer can browse, e.g.:

```
privacy check of habitats and deliberation of privacy
  issue
```

By using the HERMES language, interpretive critics can be more general, more expressive, and more complex than JANUS critics. They are not restricted to spatial relations of individual design units in the palette or to a single construction area. They can analyze, for instance, multiple habitats in the database, evaluate global characteristics of designs (like number of parts or absence of particular parts), and make their analysis dependent on other conditions in the database. Examples of complex critics are the privacy critics described in Chapter 9 and the academic advising critic discussed in Appendix B.

Because the whole language can be used and the whole database accessed, critics can be made dependent upon information in other designs, in an issue base, or in a distinct specification component (as indicated in Chapter 8). The critics can play an important role in integrating diverse pieces of information in the system.

Critics in HERMES are called *interpretive* because of the synergy which they engender between the HERMES language and the mechanism of interpretive perspectives. This is best explained with an example. Suppose Desi defined a critic named `refrigerator access` as:

```
if refrigerators are too near doors then refrigerator
  access message
```

Now, if Desi had defined `too near` as `closest distance is less than 5 feet` but Archie had modified `too near` to be `closest distance is less than 3 feet`, then the `refrigerator access` critic will be

"interpreted" differently in Archie's perspective then in Desi's. Since the language allows critics to be built up to arbitrary levels of complexity, a critic like the `academic advising` critic (in Appendix B) may be dependent upon the definition of many sub-expression, which may be defined differently in different perspectives. The point in the example is that Desi and Archie have different interpretations of what it means for something in the kitchen to be too close to something else. In another domain (e.g., molecular chemistry or astronomy) the term `too near` might need to be redefined more drastically. The perspectives mechanism assures that the evaluation of an interpretive critic will always interpret the terms and sub-expressions of the critic's definition within the context of the current active perspective.

Comparison with JANUS critics. HERMES critics are defined in the high level representations made available through the language. That is, they can be defined using vocabulary that is close to the problem domain, without needing to think in the explicit functional manner of the LISP syntax used by MODIFIER. All of the critics used in systems like JANUS and MODIFIER can be concisely stated in the HERMES language. Following are definitions of terms used for defining these critics:

```
next to:      closest distance is less than 4 inches
far from:     closest distance is more than 30 inches
close to:     central distance is less than 60 inches
near:         closest distance is less than 12 inches
set off from:     close to and not next to
work triangle distances:    list of closest distance in
   feet among sink, stove, refrigerator
```

Using these terms, the equivalent of JANUS' critic rules can be concisely and readably defined as follows in the HERMES language:

```
all stoves are set off from sinks
no stoves are next to refrigerators
all stoves are far from all doors and windows
all dishwashers are next to sinks
all refrigerators are far from all windows
refrigerators are close to doors
sinks are near windows
the minimum work triangle distances are less than 23
```

In MODIFIER, the critic rules are meant to be available to and modifiable by the end-user. However, they are written in LISP. Thus, a designer wishing to modify a critic rule in MODIFIER must be at least somewhat familiar with the complexities of LISP doctrine, including its non-intuitive Polish notation. In addition, conventions of MODIFIER's property sheets must be understood and used to make explicit computational decisions. For instance, the HERMES critic,

```
all stoves are set off from sinks
```

appears in MODIFIER's property sheets as:

```
not_next_to (stove , sink )        apply to: all

near (stove, sink)                 apply to: one
```

The parentheses of LISP in MODIFIER's critics are replaced in HERMES by an implicit nested phrase structure that is familiar to people from natural language. This nesting is unambiguously determined at definition time through the tacit use of the interface to the language discussed above. Figure 10-4 shows the explicit phrase structure for the critic rule just discussed. Note that this diagram not only expands the definition of `set off from` (which has been abstracted in the rule statement), but also indicates the clauses `at least one` and `in kitchen`, which are computationally important but are implicit in the expression that the user sees and manipulates. That is to say, both the structure of the critic and substantial contents of it are kept implicit and are hidden from the user's explicit understanding, in much the sense that the explicit phrase structure of normal speech is not usually an object during ordinary communication.

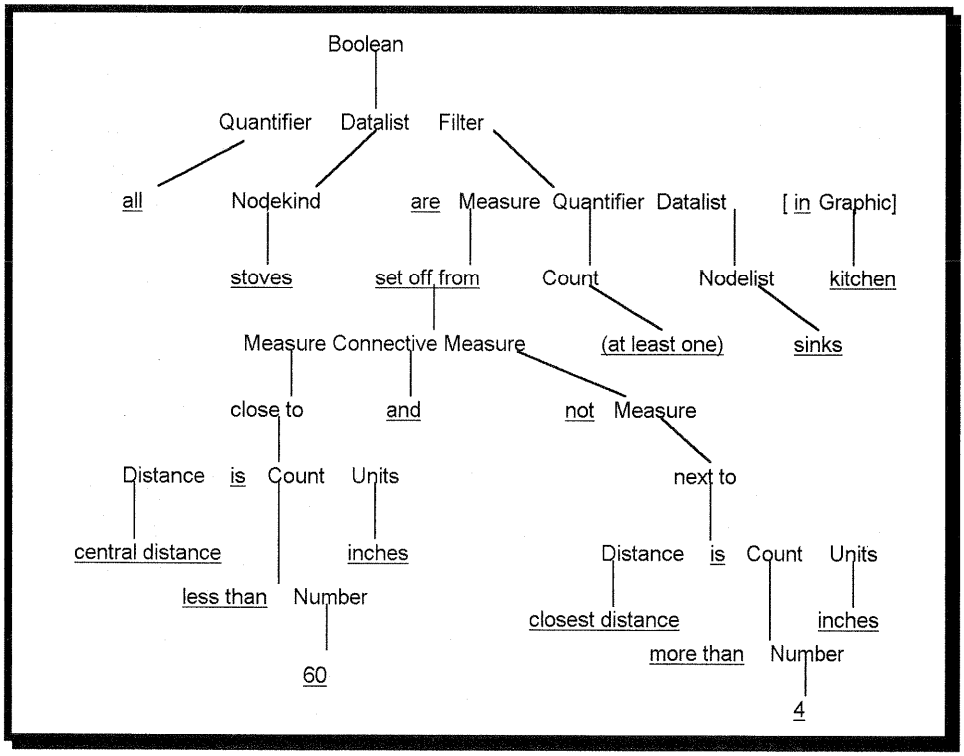


Figure 10-4. Phrase structure of a HERMES critic rule.
The critic rule can be read from the leaves of the tree: `all stoves are set off from [at least one] sinks [in kitchen]`. *Phrases in brackets are implicit. The phrase* `set off from` *can be expanded as:* `central distance is less than 60 inches and not closest distance is more than 4 inches`.

The point of this diagram is not to show how complex interpretive critics are internally, but on the contrary to show how rules that are inherently quite complex can be expressed in apparently relatively simple expressions (like, `all stoves are set off from sinks`), which hide much of the complexity that the user ordinarily does not need to be concerned with.

Analysis of the privacy critics. The privacy critics developed in the scenario of Chapter 9 provide a good example of a complex definition in the HERMES language. A close look can reveal both some of the advantages of using the language and also some of the difficulties.

The task of the privacy critic is to determine if public areas of a lunar habitat are too near to private areas. So first the notions of privacy and nearness must be operationalized and applied to areas within habitats. A `privacy values` scale from 1 to 9 is established and these Number values are given names from `very public` (1) to `very private` (9). Links of type `privacy rating` are attached to various parts of the habitats and connected to nodes with appropriate `privacy values`. The Measure term `too near` is defined as:

```
closest distance is less than 5 feet
```

Now it is possible to define public and private areas:

```
public area: parts that have privacy ratings that are
  less than somewhat public

private areas: parts that have privacy ratings that are
  more than somewhat private
```

These are Computed Associations or Predicates. They look at all the parts of whatever DataList they are applied to. These parts are then Filtered by checking if they have `privacy ratings` links and furthermore if the nodes connected by such links lead to values greater or less than the values named `somewhat private` or `somewhat public`. Any parts found that have at least one such link will be returned by these expressions.

It would be more efficient to make these definitions for `immediate parts` (i.e., top level parts of the habitats) rather than all `parts` (including subparts, all the way down to primitive graphical polygons). That would save considerable traversal of the hierarchies of graphical objects making up the habitats. However, that would require that the person defining the expression knew that all the relevant public and private parts were defined as top level parts of the habitats. If the designer defining this expression had also constructed the habitat graphic this would be possible. For the sake of generality that has not been assumed in this discussion.

Note that a given part might have multiple `privacy rating` links (even in the same perspective). The definitions above only require one such link meeting the Filter condition. Thus, a given part could be returned as both a `public area` and a `private area`. Such an anomaly would quickly show up as a `problem area` in

the critic results. In general, the ability of the definitions to deal reasonably with such multiple-definitions is an aspect of robustness in the HERMES language. It is discussed in Appendix B under the topic of defeasible reasoning.

The next step is to create a display of `problem areas`, that is, `private areas` that are too near to `public areas`. This can be accomplished with the following definition of `problem areas`:

```
private areas that are too near public areas of that
  (last subject) with those items
```

The idea here is to select one `private area` of a habitat at a time and for each one to then iterate through all the `public areas` of the same habitat and list the `public areas` that are `too near` to the selected `private area`. Both `private areas` and `public areas` are Associations that operate on the same habitat (DataList) when the overall `problem areas` Association is applied to a habitat or a list of habitats.

The Filter syntax option used in the definition of `problem areas` has the following form: Measure [Quantifier] DataList [in Graphic]. The Measure has already been defined and stored with the name `too near`. The optional [Quantifier] defaults to an implicit "`at least one`". The DataList that the `private area` is to be measured to is each `public area` of whichever habitat is currently being operated on by the `problem areas` Association. To define a DataList consisting of these `public areas`, the deictic variable, `that (last subject)`, is used to refer to the habitat to which the `problem areas` Association is applied. This deictic refers to the most recently defined "subject" to which operators are being applied, namely the "subject" of the `problem areas` Association. Here the term "subject" refers to the DataList that is the input to the evaluation of an expression. A stack of recent subjects is maintained in order to implement this deictic variable. The parenthetical explanatory phrase, "`(last subject)`", departs from the tacit feel of the language in order to alert the reader to think explicitly about the computational structure of operator application in this case because a reference is being made to some term outside the immediate expression—namely to the subject to which this expression will be applied.

The optional [in Graphic] phrase defaults to `in that (last subject)`, which, again, refers to the "subject" of the `problem areas` Association. That means that the measurement of distance between the `private area` and the `public area` is computed within the graphical habitat. Unless a graphical object is explicitly named as the context for distance measurements, the assumption is made that the last explicitly named subject should serve this role. The necessity of naming (tacitly or explicitly) a graphical context for measurements arises from the generality of the HERMES language, which can be referring to any object in the database, rather than to the content of a unique construction area as assumed in JANUS and PHIDIAS.

Finally, in the definition of `problem areas` the deictic variable `those items` refers to the most recently enumerated items, namely the `public areas`

that are enumerated for each `private area` and that satisfy the Filter condition. During the testing of the Filter condition, the successful enumerated items are stored on a special list that can be referenced by the special deictic variable "`those items`". Thus, the `with those items` phrase following the Filter phrase retrieves the list of `public areas` that are `too near` a given `private area` and adds them to the result list of the critic following that `private area` and indented under it.

Now that the computational heart of the `privacy check` critic has been defined, the critic can be assembled. First, a privacy message is defined to be displayed in the case that no problem areas are found for a given habitat. This is simply a Character node with the contents:

```
"Public and private areas are separated."
```

This node can be named `privacy message` or it can be linked to the `privacy check` critic itself. If it is named, the critic is defined as:

```
name with either name of problem areas or privacy
  message
```

If it is linked with a link of type `message`, then the critic is defined as:

```
name with either name of problem areas or message of
  this (expression)
```

In the latter case, the reference to the `privacy message` is replaced by a computation, `message of this (expression)`, using the deictic variable `this`. The variable `this (expression)` refers to the current object itself, so `message of this (expression)` follows the `message` link from the definition of this critic to the Character node whose content is the required message. Again, the use of parentheses signals the need for some explicit reflection by the reader.

The `privacy check` critic uses the implicit if / then construction, `either / or`, in which the first phrase is used if it produces any results, otherwise the second phrase (in this case, simply displaying the message) is used. The principal work done by the definition of `privacy check` is to display the names of graphical objects, rather than displaying them as graphics. `Privacy check` is a Computed Association that is applied to a DataList of one or more habitats. So it first displays the name to the habitat to which it is being applied, then (indented under that name, because of the `with` conjunction) it computes the list of `problem areas` of that habitat and displays the names of all the items in the resultant list (including the names of the `public areas` that are indented in the list under the `private areas`). If the resultant list was empty for a given habitat, the `privacy message` is displayed instead.

In the scenario, a variation on `privacy check` named `privacy display` was defined:

```
name and privacy ratings of problem areas
```

This critic displays the privacy ratings as well as the names of all items in the list computed by the problem areas Association.

Recall from Chapter 3 that the lunar habitat designers eventually settled on a concept of *privacy gradient* in the transcribed session. That meant that they wanted the arrangement of the habitat to change gradually from private areas to public areas. To operationalize this notion, one could introduce a test to see if any two areas that are near each other differ by more than a value of, say, plus or minus two. This introduces explicit arithmetic computations into the definitions of a critic. It also introduces a complicated comparison of each habitat part with all the other parts of the habitat. The following set of definitions can be used to compute habitat parts that are incompatible in this sense of a privacy gradient.

In the Chapter 9 scenario, the designers ended up with a critic called privacy gradient catalog. It goes through all habitats in the database, selecting those for which privacy ratings links are attached to some parts. For those habitats, it displays their name and an analysis of how they meet the defined privacy gradient considerations:

```
name with privacy gradient critique of habitats that
  have parts that have privacy ratings
```

For each habitat that has privacy ratings, the privacy gradient critique is displayed. This is similar to the privacy display, above, in that it computes problem parts using a privacy gradient listing Association, or else displays a privacy gradient message. Here are the definitions to handle this:

```
privacy gradient critique: either privacy gradient
  listing or privacy gradient message

privacy gradient listing: name and privacy ratings of
  parts that have privacy ratings with their problem
  parts

privacy gradient message: "The parts of this design are
  arranged along a privacy gradient."
```

The privacy gradient listing Association iterates through the parts of a habitat and for each part lists (indented) their problem parts. The definition of problem parts is the tricky part. It uses three further definitions: too near, other parts, and are incompatible. The Measure, too near, is the same as it was in the privacy check critic, except that in the current perspective it has been modified from 5 feet to 3 feet:

```
problem parts: name and privacy ratings of other parts
  that are too near that (last subject) and that are
  incompatible

too near: closest distance is less than 3 feet
```

```
other parts: parts of inverse parts that do not equal
  that (last subject)

are incompatible: have privacy ratings that are more
  than privacy ratings of that (last subject) + 2 or
  are less than privacy ratings of that (last subject)
  -2
```

The definition of `other parts` requires some explanation. Within the `privacy gradient listing` expression, the Association `problem parts` must be applied to `parts` (of a habitat). The definition of `problem parts` centers on the definition of `other parts`. However, what is wanted is "other parts" of the habitat, not other parts of the selected part of the habitat, which is what would result from the application of `problem parts` to `parts`. Therefore, within the definition of `other parts`, the computation must get back to the habitat by tracing backwards the `part` link between the habitat and its part. This is accomplished by the construction, `inverse parts`. Once the computation is back at the habitat, it can find the other parts by navigating all the `parts` (i.e., graphical content) links of the habitat. Of course, the computation of "other parts" should exclude the part from which the computation began in order to avoid comparing that part with itself. This is accomplished with the Filter, `that do not equal that (last subject)`, in which the deictic variable `that (last subject)` refers to the last "subject" of application, namely the original `part` iterated in the `privacy gradient listing` expression.

The definition of the Filter, `are incompatible`, uses the same `that (last subject)` variable in order to compare each of the `other parts` with the original `part`. This Filter also introduces explicit arithmetic in order to judge whether the `privacy ratings` of these two parts differ by more than 2 on the privacy scale. This comparison completes the operationalization of the idea of a privacy gradient as it occurred in the lunar habitat design transcript.

The definition of `privacy gradient catalog` with all its preliminary definitions is a relatively formidable task. If one undertakes figuring it out from scratch, it might well seem that the task is easier to do in a traditional programming language. This seems especially true to people who are experienced in programming. It may well be that such a task pushes the HERMES language to near its limits. On the other hand, a design environment built on the HERMES substrate might support reuse and modification sufficiently to make the HERMES alternative preferable. First, much of the defining could have been done in the seeded set of language definitions, providing a well thought-out collection of building blocks for complex tasks involving privacy. If this was not available in the original seed, a reseeding process could take place when the privacy issue is raised as an important concern. Then an experienced programmer or a HERMES local developer could step in and provide a set of privacy-related definitions for everyone to use.

As stated at the outset of this chapter, the HERMES language has been developed to push its approach to supporting a mix of tacit and explicit understanding as far as possible and to explore its limits. The `privacy gradient critique` expression provides an important test of these limits. On the one hand, it shows that the task that appeared extremely challenging back in Chapter 3 can in fact be accomplished using the HERMES language. On the other hand, it shows that such a task may strain the limits of the language. The limits of the language are explored further by examples in Appendix B. More thorough experience will have to await the building of robust design environments on the HERMES substrate and their use by a community of designers.

CHAPTER 11. CONTRIBUTIONS

The topic of this dissertation has been the problem of providing computer support for cooperative design given the nature of tacit and explicit understanding. But at a meta-level, an important theme has been the role of theory in software design. Often, work in cognitive science and artificial intelligence proceeds with little reference to philosophy, which is given lip service as one constituent of these interdisciplinary endeavors. Of course, preconceptions abound in such work, but they are either treated as self-evident common sense or addressed through discussions of individual concepts whose inner coherence remains outside the investigation.

This dissertation is an attempt to take *theory* seriously in computer science. Rather than first creating a software artifact whose theory is at best only tacitly available retrospectively,[10] and then subjecting the artifact to controlled user testing to determine its effectiveness, the approach followed here is to formulate a set of explicit theoretical principles to motivate an approach to computer support of design and then to present a package of prototyped functionality to illustrate that approach. Together, the theory and the examples are meant to provide cogent arguments for the deliberation of central issues in software design of systems to support innovative, collaborative design work in exploratory domains.

Of course, several preconceptions have been at work here, too. However, the major assumptions have been systematically reflected upon in the process and explicated or modified as need be. It has been assumed, for instance, that software to support professional designers should be based on an understanding of the structure of their work processes. As a guiding idea, the design process was viewed (or pre-viewed) as a process of interpretation (Chapter 1). Two approaches were then taken to explore this work process: one by looking at some of the best available descriptions of the way designers work (by Alexander, Rittel, and Schön in Chapter 2), and the other by looking at a concrete example of designers working (on lunar habitat design in Chapter 3). To make this theory even more explicit and general, it was then put into the framework of a philosophy (Heidegger's hermeneutics in Chapter 4). An explicit theory of computer support for interpretation in design was built on top of the results of the preceding investigations (Chapter 5 and 6). The theory developed in this way was then used to evaluate related software systems

---

[10]Carroll and associates have made a case for considering artifacts as themselves implicit expressions of theories, as though guiding philosophies were unnecessary. This case has been made specifically in terms of software artifacts in the realm of human-computer interaction, and has even been related to hermeneutics (Carroll & Campbell, 1989; Carroll & Kellogg, 1989). While they persuasively point out problems with the traditional assumptions about the relation of psychological theory to design practice, they overlook the spiral character of understanding, in particular the guiding role of (often tacit) philosophical beliefs and conceptual frameworks.

meant to support design (Chapter 7). Finally, the theory served to motivate and justify design decisions in the HERMES software (Chapters 8, 9, and 10).

While this approach stresses theory, it does not ignore the need for empirical grounding or iterative testing. The design methodologies reviewed all grew out of either reflection on professional practice or consideration of experimental findings. The study of lunar habitat design pursued as part of the dissertation took on the flavor of participatory design (Ehn, 1988; Greenbaum & Kyng, 1991) by having the software designers and the design professionals working together on a lunar habitat design, and by involving the two groups in dialogue about the design work and about possibilities for computer-based support of this work. Although it was never reflected in the Chapter 3 transcripts, the lunar habitat designers have been involved in on-going evaluation of the HERMES system and its functionality as part of their role as corporate sponsors of the funded research. In addition, the design of HERMES is a response to empirical experience with the related design environments on which it is based, as well as on a series of programming walkthroughs to evaluate the HERMES language design (reported in Appendix A).

Evaluation and refinement of the Lunar Habitat Design Environment (LHDE) and PHIDIAS II built on top of the HERMES substrate are expected to continue indefinitely. Clearly, the greatest need for future work is to build a robust design environment that exercises all of the functionality of the HERMES substrate and to gain experience in the utility of this functionality through use by professional designers. Unfortunately, that is beyond the scope of the present effort. For one thing, it will involve identifying real-world projects in which a system like LHDE makes commercial sense in order to get professionals to invest significant time in using preliminary versions. The support of lunar habitat design has served as a fruitful application domain in developing HERMES, but a specific project must now provide a practical context for further participatory development and workplace evaluation.

It is useful to view the unfolding of this dissertation as a hermeneutic process, in which a vague preconception of interpretation in design becomes increasingly clearer through precisely the kind of interpretive process that has been analyzed in the dissertation. The concept of interpretation has been elaborated through an investigation of the role of interpretation in design. The guiding perspective was the intuition that *interpretation is the central category for founding a theory of computer support.* This perspective was tied through a process of reflection to its explicit roots in Heidegger's philosophy, but also to the almost forgotten role of interpretation in the related systems that HERMES grew out of. In a sense, the dissertation embodies a moment of reflection in which the effort to build systems of computer support ran up against the limits of multi-faceted, domain-oriented, knowledge-based systems; made explicit the role of interpretation in design; and then, using this, proposed a system that *integrates the facets in a hypermedia substrate, extends the notion of domain-orientation with perspectives*, and uncovers the basis of explicit computer knowledge representations in the *expressing of tacit human preunderstanding in language.*

In looking back over what has been accomplished in this dissertation, it is clear that no final answers have been given. The analysis of interpretation remains

unclear and incomplete in many ways. The theory of computer support is no more than a beginning in an attempt to provide rationale for a new direction in artificial intelligence. The design of HERMES is suggestive of promising functionality, but this promise remains largely untested. Nevertheless, whatever the limits of this work, it does seem to have made significant contributions on three primary levels: on a philosophical level (11.1), on a theoretical level (11.2), and on a system building level (11.3).

11.1 CONTRIBUTIONS TO A PHILOSOPHY OF INTERPRETATION

At least since Dreyfus (1966; 1972; & Dreyfus, 1986), the relevance of Heidegger's philosophy to AI has been debated. Unfortunately, most of the discussion by computer scientists has relied on secondary sources, especially pre-publication drafts of Dreyfus' (1991) commentary on Heidegger. So one contribution of this dissertation has been to return to the original text of Heidegger (1927) and to systematically apply that text to the context of computer support for interpretation in design. The result has been an *analysis of interpretation* that is frequently more detailed and rigorous than alternative presentations. This represents a contribution to Heideggerian scholarship as such, not just from a computer science perspective.

Of course, according to the philosophy there is no "correct" interpretation of a text unrelated to a background of concerns. The confrontation of the Heideggerian text with the problematic of design and computer support for design had important consequences. Examples from design methodology and from lunar habitat design provided not only a concreteness to Heidegger's abstractions, but a more realistic context than Heidegger's own craft-oriented glimpses of the lonely carpenter absorbed in his hammering. Design shifted the emphasis to collaborative work. It also moved (thanks largely to Schön's insights) from use of the physical artifact to the more conceptual design of artifacts. In particular, this brought to the fore the role of discovery over that of laying out what was implicitly disclosed. This clarified and extended the analysis of interpretation, removing certain ambiguities that Heidegger glossed over.

Perhaps most importantly, the effort to apply Heidegger's philosophy to computer system building not only forced a precision of concept, but resulted in the operationalizing of many of the ideas. This is, of course, a common benefit to philosophy of mind when it is applied in AI. In this case, the result was a computer model of human interpretation as situated, perspectival, and linguistic. However, in addition to the model, there is an extensive recognition of the limits of the model and the need to involve people in the operation of the model. These limits are shown to be consequences of the Heideggerian analysis. So philosophy benefited from its meeting with computer science.

11.2 CONTRIBUTIONS TO A THEORY OF COMPUTER SUPPORT

The central contribution was to identify the key concept for a theory of computer support: interpretation. Although Winograd & Flores (1986), for instance, talked a lot about interpretation, they ranged across Heidegger's (1927) framework and focused on its critique of technical rationality. Ironically, their proposed software example, the COORDINATOR program, suffered from a lack of respect for the importance of interpretative control by the users. They failed to take seriously the fact that there is no objective structure to a domain and that people should be supported in defining their own analyses, interpretations, and terminologies from their own perspectives. Support for interpretation is the ingredient missing from most traditional AI programs. This dissertation contributes the antidote: a *recognition of the central role of interpretation and the impossibility of fully automating it.* It is difficult to convey the potential importance of this contribution; that is why so many pages of the dissertation have been devoted to this theme.

The proposed theory of computer support is built squarely on the analysis of interpretation. This gives the theory a coherence and consistency missing from other theoretical frameworks in computer science (other than those based on strictly formal logical grounds). It demonstrates how philosophy (again, other than logic) can be put in the service of computer science.

Knowledge-based system design inevitably raises the question of the nature of knowledge. Some contributions have been made here. First, *the varieties of knowledge or information have been categorized* in terms of their origins in various phases of the process of interpretation. This includes not only tacit and explicit understanding, but also shared understanding and captured computer representations. Second, the idea of domain knowledge has been critiqued. Not only does knowledge in a design domain change as the related technologies and styles change and as the expertise of the field matures and grows, but every designer and every design team has their own domain knowledge. It is not simply that they each have different pieces of an underlying knowledge. Rather, to know is to know from a perspective, so *there is no objective body of domain knowledge independent of what people know in their own ways*, within their many perspectives. Third, the role of language in expressing knowledge has been emphasized. *The emergence of interpersonal or operationalized knowledge from tacit experience takes place through discourse and assertion* within situated interpretation. Correspondingly, an end-user language has an important role to play in computer support.

11.3 CONTRIBUTIONS TO A SYSTEM FOR INNOVATIVE DESIGN

The effort to illustrate the functionality called for by the theory resulted in three major contributions to building computer support for innovative design: (a) a hypermedia knowledge representation substrate, incorporating: (b) a system of perspectives and (c) an end-user language. The design of each of these features has been thought through, both in terms of the functionality required by the theory and in terms of their usability in a practical computer system for design professionals. Each has also been prototyped in executable code and subjected to testing to confirm the implementability of the ideas. Various versions of these features, along with auxiliary functionality have also been incorporated in a series of design environments that have been shown to lunar habitat designers for feedback.

(a) The *hypermedia substrate* incorporates the power of the fine-grained hypertext in the original PHIDIAS system, provides an efficient and scalable object-oriented database for persistence, incorporates multi-media nodes, and integrates the perspectives and language into the fundamental node and link structure. This hypermedia offers an extremely powerful and flexible knowledge representation system, whose control by the user is limited primarily by the lack of a fuller user interface. Adaptability by the user—or plasticity of representation—is critical according to the theory. The HERMES hypermedia contributes an example of a substrate for supporting such adaptability.

(b) The *perspectives mechanism* is a contribution to Computer Supported Cooperative Work. It allows individuals to organize their own versions of knowledge representations and to share them. This provides a tool for supporting the evolution of knowledge by starting with systematically organized domains and allowing users to inherit and modify these and to organize meaningful new domains. The virtual copying approach is an inherently efficient mechanism, which encourages consistency by eliminating unnecessary duplication of representations in multiple copies.

(c) The *HERMES language* is a contribution to end-user programming languages and programmable design environments. It suggests ways of reducing the programming doctrine that users have to learn or keep in mind. Much of the traditional programming language doctrine is suppressed by keeping the corresponding features tacit in the HERMES language. Also, the appearance of expressions in the language supports tacit understanding by making heavy use of user-defined domain terminology and by following several syntactic conventions of natural language. At the same time, when the computational structure of an expression must be made more explicit to be understood or modified, this can be done to some extent through interface displays and to some extent by exploratory execution. A programming language paradigm that was implicit in PHIDIAS' query language has been pushed forward, extended, and modified to the point of a powerful end-user language that can play key roles in a system to support interpretation.

Computer technology can contribute to human emancipation. By providing computationally active media of external memory, it can significantly extend cognitive capabilities within an increasingly complex world. However, that requires a people-centered approach in which machine computations are at the service of human judgments and interpretation. Mainstream software approaches have developed within a social context dominated by the interests of military, government, and multinational corporations, resulting in computer applications that replace people or that dictate how they think and work. This dissertation has tried to present design rationale to oppose the bureaucratic interests, a theory to guide people-centered software development, and example mechanisms for giving people innovative, shared control over software computations.

BIBLIOGRAPHY

Abelson H, Sussman G (1985) *Structure and Interpretation of Computer Programs.* Cambridge: MIT Press.

Adorno TW (1964). *Jargon der Eigentlichkeit: Zur deutschen Ideologie.* [The Jargon of Authenticity]. Frankfurt am Main: Suhrkamp.

Adorno TW (1966). *Negative Dialektik* [Negative Dialectics]. Frankfurt am Main: Suhrkamp.

Alexander C (1964) *Notes on the Synthesis of Form.* Cambridge: Harvard University Press.

Alexander C (1971) The State of the Art in Design Methods. In Cross N (1984). 309-316.

Alexander C, Ishikawa S, Silverstein M, Jacobson M, Fiksdahl-King I, Angel S (1977) *A Pattern Language: Towns, Buildings, Construction.* New York: Oxford University Press.

Alexander C, Poyner B (1966) The Atoms of Environmental Structure. In Cross N (1984). 123-133.

Backus J (1978) Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM. 21.* (8). 613-641.

Bell B, Citrin W, Lewis C, Rieman J, Weaver R, Wilde N, Zorn B (1992).The Programming Walkthrough: A Structured Method for Assessing the Writability of Programming Languages. Technical Report CU-CS-577-92 January 1992. Department of Computer Science. University of Colorado.

Bernstein B (1992) Euclid: Supporting Collaborative Argumentation with Hypertext. Technical Report CU-CS-596-92 January 1992. Department of Computer Science. University of Colorado.

Bluth BJ (1984) *Space Station/Antarctic Analogs.* Washington, D.C.: NASA.

Bluth BJ (1986) Lunar Settlements—A Socio-economic Outlook. *37th Congress of the International Astronautical Federation,* Innsbruck, Austria, October 4-11. Oxford: Persimmon Press.

Boborow DG, Goldstein IP (1980a) *An Experimental Description-Based Programming Environment: Four Reports.* Technical Report CSL-81-3. Palo Alto, CA: Xerox Palo Alto Research Center.

Boborow DG, Goldstein IP (1980b) Representing Design Alternatives. In Boborow & Goldstein 1980a. 19-29.

Boborow DG, Winograd T (1977) An Overview of KRL, A Knowledge Representation Language. *Cognitive Science* 1 (1), 3-46. In Brachman & Levesque (1985).

Boeing Aerospace Company (1983) *Space Station/Nuclear Submarine Analogs.* Granada Hills, CA: National Behavior Systems.

Boland RJ Jr., Maheshwari AK, Te'eni D, Schwartz DG, Tenkasi RV (1992) Sharing Perspectives in Distributed Decision Making. *CSCW '92 Proceedings.*

Bourdieu P (1977) *Outline of a Theory of Practice.* Oxford: Oxford University Press.

Bourdieu P (1991) *The Political Ontology of Martin Heidegger.* Stanford: Stanford University Press.

Brachman R, Levesque H (1985) *Readings in Knowledge Representation.* San Mateo: Morgan Kaufmann.

Buchanan BG, Shortliffe EH (1984) Human Engineering of Medical Expert Systems. In Buchanan BG, Shortliffe EH (Eds.) (1984) *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, MA: Addison-Wesley. 599-612.

Budde R, Züllighoven H (1990) *Software-Werkzeuge in einer Programmierwerkstatt: Ansätze eines hermeneutisch fundierten Werkzeug- und Maschinenbegriffs*. [Software Tools in a Programming Workshop: Approaches to an Hermeneutically-based Concept of Tools and Machines]. München: Oldenbourg Verlag.

Bush V (1945) As We May Think. *Atlantic Monthly*. 176 (1), 101-108. Reprinted in Greif I (1988) *Computer-Supported Cooperative Work*. San Mateo, CA: Morgan Kaufmann.

Cardelli L, Wenger P (1985) On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys. 17*. (7). 471-522.

Carroll JM, Campbell RL (1989) Artifacts as Psychological Theories: the Case of Human-Computer Interaction. *Behavior and Information Technology*. Vol. 8, no. 4, 247-256.

Carroll JM & Kellogg WA (1989). Artifact as theory-nexus: hermeneutics meets theory-based design, *Proceedings of the Conference of Human Factors in Computing Systems*, Austin, 7-14.

Compton WD, Benson CD (1983) *Living and Working in Space: A History of Skylab*. Washington, DC: NASA.

Conklin J, Begeman M (1988) gIBIS: A Hypertext Tool for Exploratory Policy Discussion. *Proceedings of the Conference on Computer Supported Cooperative Work*. New York: ACM. 140-152.

Coyne R (1991). Inconspicuous Architecture, *Gadamer Action & Reason: Conference Proceedings*. Australia: University of Sydney. 62-70.

Coyne R, Snodgrass A (1991) What Is the Philosophical Basis of AI in Design? Working paper. Faculty of Architecture, University of Sydney.

Cross N (1984) *Developments in Design Methodology*. New York: Wiley.

Dayton T, et al. (1993) Skills Needed by User-centered Design Practitioners in Real Software Development Environments: Report on the CHI '92 Workshop. *SIGCHI Bulletin. 25* (3) 16-31.

Design Edge (1990) *Initial Lunar Habitat Construction Shack*. Design control specification. Houston, TX.

Descartes R (1641) *Meditations of First Philosophy*. Indianapolis: Hackett. 1979

Donald M (1991) *Origins of the Modern Mind: Three Stages in the Evolution of Culture and Cognition*. Cambridge: Harvard University Press.

Dreyfus H (1966) *Alchemy and Artificial Intelligence*. Rand paper P3244. The Rand Corporation.

Dreyfus H (1967) Phenomenology and Artificial Intelligence. In Edie J (ed.) (1967) *Phenomenology in America*. Chicago: Quadrangle. 31-47.

Dreyfus H (1972) *What Computers Cannot Do*. New York: Harper and Row.

Dreyfus H (ed.) (1982) *Husserl, Intentionality, and Cognitive Science*. Cambridge: MIT Press.

Dreyfus H (1985) Holism and Hermeneutics. In Hollinger R (Ed.) (1985) *Hermeneutics and Praxis*. Notre Dame, IN: University of Notre Dame Press. 227-247.

Dreyfus H (1990) Heidegger's History of the Being of Equiptment. In Dreyfus H, Hall H (eds.) (1991) *Heidegger: A Critical Reader*. Oxford: Basil Blackwell. 173-185.

Dreyfus H (1991) *Being-in-the-World: A Commentary on Heidegger's Being and Time, Division I*. Cambridge: MIT Press.

Dreyfus H, Dreyfus S (1986) *Mind Over Machine*. New York: Free Press.

Ehn P (1988) *Work-Oriented Design of Computer Artifacts*. Stockholm: Arbetslivscentrum.

Eichold A (1992) Lunar Base Planning Criteria. NEA grant final report. Washington, DC: NEA.

Eisenberg M (1992). SchemePaint: A Programmable Application for Graphics. Technical Report CU-CS-587-92. Computer Science Department, University of Colorado at Boulder.

Eisenberg M, Fischer G (1992) Programmable Design Environments and Design Rationale. AAAI'92 Workshop on Design Rationale Capture and Use. San Jose, CA. July 15, 1992. 81-90.

Engelbart D (1963) A Conceptual Framework for the Augmentation of Man's Intellect. In Howerton, P (Ed.) (1963) *Vistas of Information Handling*. (Vol. 1). Washington, DC: Spartan Books. Reprinted in Greif I (Ed.) (1988) Computer-Supported Cooperative Work. San Mateo, CA: Morgan Kaufmann.

Ericsson KA, Simon HA (1984) *Protocol Analysis: Verbal Reports as Data*. Cambridge: MIT Press.

Fischer G (1989) Creativity Enhancing Design Environments. *Proceedings of the International Conference "Modeling Creativity and Knowledge-Based Creative Design"*. Heron Island, Australia. 127-132.

Fischer G (1991) Supporting Learning on Demand with Design Environments. *Proceedings of the International Conference on the Learning Sciences August 1991*. Evanston, IL. 127-132.

Fischer G, Girgensohn A (1990) End-User Modifiability in Design Environments. *Human Factors in Computing Systems, CHI '90 Conference Proceedings (Seattle, WA)*. New York: ACM.

Fischer G, Grudin J, Lemke A, McCall R, Ostwald J, Reeves B, Shipman F (1991a). Supporting Indirect, Collaborative Design with Integrated Knowledge-Based Design Environments. Submitted to *Human-Computer Interaction*.

Fischer G, Lemke A, McCall R, Morch A (1991b) Making Argumentation Serve Design. *Human-Computer Interaction* (Special Issue on Design Rationale). *6*, (3 & 4), 393-419.

Fischer G, McCall R, Morch A (1989) Janus: Integrating Hypertext with a Knowledge-based Design Environment. *Proceedings of Hypertext '89*. Pittsburgh, PA: ACM, 105-117.

Fischer G, McCall R, Ostwald J, Reeves B, Shipman F (1993c) Seeding, Evolutionary Growth and Reseeding: Supporting Incremental Development of Designs and Design Environments. Submitted to *AAAI'93*.

Fischer G, Nakakoji K (1992) Beyond the Macho Approach of Artificial Intelligence: Empower Human Designers—Do Not Replace Them. *Knowledge-Based Systems Journal. 5*, (1), 15-30.

Fischer G, Nakakoji K, Ostwald J, Stahl, G, Sumner T (1993a) Embedding Computer-Based Critics in the Contexts of Design. *Proceedings of InterCHI '93. Conference on Human Factors in Computing Systems*. Amsterdam. April 1993. 157-164.

Fischer G, Nakakoji K, Ostwald J, Stahl, G, Sumner T (1993b) Embedding Critics in Design Environments. *The Knowledge Engineering Review*, Special Issue on Expert Critiquing. Fall 1993.

Fitzgerald F, Rashid R (1986) The Integration of Virtual Memory Management and Interprocess Communication in Accent. *ACM Transactions on Computer Systems, 4*, (2), 147.

Floyd C, Züllighoven H, Budde R, Keil-Slawik R (1992) *Software Development and Reality Construction*. Heidelberg: Springer Verlag.

Foley J, van Dam A, Feiner S, Hughes J (1990) *Computer Graphics: Principles and Practice*. Reading, MA: Addison-Wesley.

Fodor J (1981) Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology. In Haugland (1981). 307-338.

Freud S (1917) *A General Introduction to Psychoanalysis*. New York: Washington Square Press. 1952.

Gadamer H-G (1960) *Wahrheit und Methode*. Tübingen: Mohr. Translation: Gadamer H-G (1988) *Truth and Method*. New York: Crossroad.

Gadamer H-G (1966) Die Universalität des hermeneutischen Problems [The universality of the hermeneutic problem]. In Gadamer HG (1967) *Kleine Schriften I Philosophie Hermeneutic*. Tübingen: Mohr. 101-112.

Gadamer H-G (1967) Rhetorik, Hermeneutik und Ideologiekritik [Rhetoric, Hermeneutics and Ideology Critique]. In Gadamer HG (1967) *Kleine Schriften I Philosophie Hermeneutic*. Tübingen: Mohr. 113-130.

Girgensohn A (1992) *End-User Modifiability in Knowledge-Based Design Environments*. Ph.D. dissertation. Department of Computer Science. University of Colorado at Boulder.

Goldstein IP, Boborow DG (1980) Descriptions for a Programming Environment. *Proceedings of the First Annual Conference of the National Association for Artificial Intelligence*, Stanford, CA. 1-6.

Greenbaum J, Kyng M (1991) *Design at Work: Cooperative Design of Computer Systems*. Hillsdale, NJ: Lawrence Erlbaum.

Greif I (Ed.) (1988) *Computer-Supported Cooperative Work*. San Mateo, CA: Morgan Kaufmann.

Grice HP (1975) Logic and Conversation. In Cole P, Morgan J (1975) *Syntax and Semantics 3: Speech Acts*. New York: Academic Press. 41-58.

Habermas J (1967) Zur Logik der Sozialwissenschaften [On the Logic of the Social Sciences]. *Philosophiphische Rundschau*. Beiheft 5, February, 1967.

Habermas J (1968) *Erkenntnis und Interesse* [Knowledge and human interests]. Frankfurt a. M.: Suhrkamp Verlag.

Habermas J (1985) *Der philosophische Diskurs der Moderne: Zwölf Vorlesungen* [The Philosophical Discourse of Modernity]. Frankfurt am Main: Suhrkamp.

Halasz F (1988) Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems. *Communications of the ACM*. Vol. 31, No. 7. 836-852.

Harnad S (1993) Grounding, Situatedness, and Meaning. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*. Boulder, CO. 169-174.

Haugland J (Ed.) (1981) *Mind Design*. Cambridge: MIT Press.

Hegel GWF (1807) *Phänomenologie des Geistes*. Translation: Hegel GWF (1967) *Phenomenology of Mind*. New York: Harper & Row.

Hegel GWF (1833) *Grundlinien der Philosophie des Rechts* [Principles of the philosophy of right]. Leipzig.

Heidegger M (1927) *Sein und Zeit*. Tuebingen: Niemeyer. Translation: Heidegger M (1962) *Being and Time*. New York: Harper & Row.

Heidegger M (1947) Brief Über den "Humanismus" [Letter on Humanism]. In Heidegger M (1967) *Wegmarken*. Frankfurt a.M.: Klostermann.

Heidegger M (1950) Ursprung des Kunstwerks [The origin of the work of art]. In Heidegger M (1950) *Holzwege*. Frankfurt a.M.: Klostermann.

Heidegger M (1951) *Erläuterungen zu Hölderlins Dichtung*. [Commentary on Holderlin's Poetry] Frankfurt a.M.: Klostermann.

Heidegger M (1953). Wissenschaft und Besinnung [Science and reflection]. In Heidegger M (1954) *Vorträge und Aufsätze*. Pfullingen: Neske.

Heidegger, M. (1971) *Poetry, Language, Thought*. Trans. A. Hoftadter. New York: Harper & Row.

Heidegger M (1975) *Der Grundprobleme der Phänomenologie* [Basic problems of phenomenology]. *Gesamtausgabe* vol. 24. Frankfurt a.M.: Klostermann.

Heidegger M (1979) *Prolegomena zur Geschichte des Zeitbegriffs* [Introduction to the history of the concept of time]. *Gesamtausgabe* vol. 20. Frankfurt a.M.: Klostermann.

Hewitt C (1971) *Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot*. Ph.D. Thesis. June 1971. Reprinted in AI-TR-258 MIT-AI Laboratory, April 1972.

Hinton G, Anderson J (1989) *Parallel Models of Associative Memory*. Hillsdale, NJ: Lawrence Erlbaum.

Hutchins E (1990) The Technology of Team Navigation. In Galegher P, Kraut R, Egido C (Eds.) (1990) *Intellectual Teamwork*. Hillsdale, NJ: Erlbaum. 191-220.

Illich I (1973) *Tools for Conviviality*. New York: Harper & Row.

Johnson JA, Nardi BA, Zarmer CL, Miller JR (1993) ACE: Building Interactive Graphical Applications. *Communications of the ACM. 36* (4). 41-55.

Kant I (1787) *Kritik der reinen Vernunft*. Translation: Kant I (1929) *Critique of Pure Reason*. New York: St. Martin's Press.

Kazmierski M, Spangler D (1992) Lunatechs II: A Kit of Parts for Lunar Habitat Design. Unpublished project report, College of Environmental Design, University of Colorado at Boulder.

Kolodner J (1984) *Retrieval and Organizational Strategies in Conceptual Memory*. Hillsdale, NJ: Lawrence Erlbaum.

Kuhn T (1962) *The Structure of Scientific Revolutions*. Chicago: University of Chicago Press.

Kunz W, Rittel H (1970) Issues as Elements of Information Systems. Working paper 131. Center for Planning and Development Research, University of California, Berkeley.

Kunz W, Rittel H (1984) How to Know What is Known: Designing Crutches for Communication. In Dietschmann HJ (Ed) (1984) *Representation and Exchange of Knowledge as a Basis of Information Processes*. North-Holland: Elsevier. 51-60.

Lakoff G (1987) *Women, Fire, and Dangerous Things*. Chicago: Univ. of Chicago Press.

Lee J (1990) SIBYL: A Tool for Managing Group Decision Rationale. *Proc. CSCW*. LA: ACM Press.

Lee J, Lai K-Y (1991) What's in Design Rationale? *Human-Computer Interaction*. 6. 251-280.

Lefebvre H (1991) *The Production of Space*. Oxford: Blackwell.

Liskov B, Snyder A, Atkinson R, Shaffert C (1977) Abstraction Mechanisms in CLU. *Communications of the ACM. 20*. (8). 564-576.

Marshall C, Halasz F, Rogers R, Jannsen W (1991) Aquanet: A Hypertext Tool to Hold your Knowledge in Place. In *Hypertext '91*. 261-275.

Marx K (1844) *Texte zu Methode und Praxis II*. Germany: Rowohlt. 1966.

Marx K (1867) *Das Kapital.* Hamburg: Meissner. Translation: Marx K (1977) *Capital.* New York: Vintage.

Mead GH (1934) *Mind, Self, and Society.* Chicago: University of Chicago Press.

Merriam-Webster (1991) *Webster's Ninth New Collegiate Dictionary.* Springfield: Merriam-Webster.

Merleau-Ponty M (1945) *Phenomenologie de la Perception.* Paris: Gallimard. Translation: Merleau-Ponty M (1962) *Phenomenology of Perception.* London: Routledge & Kegan Paul.

McCall R (1986) Issue-Serve Systems: A Descriptive Theory of Design. *Design Methods and Theories.* Vol.20, no. 3, 443-458.

McCall R (1987) PHIBIS: Procedurally Hierarchical Issue-Based Information Systems. *Proceedings of the Conference on Architecture at the International Congress on Planning and Design Theory.* New York: American Society of Mechanical Engineers. 17-22.

McCall R (1989) Mikroplis: A Hypertext System for Design. *Design Studies,* 10 (4), 228-238.

McCall R (1991) PHI: A Conceptual Foundation for Design Hypermedia. Design Studies. 12 (1), 30-41.

McCall R, Bennett P, d'Oronzio P, Ostwald J, Shipman F, Wallace N (1990a) Phidias: Integrating CAD Graphics into Dynamic Hypertext. In Rizk A, Streitz N, Andre J (eds) (1990) *Hypertext: Concepts, Systems and Applications* (Proceedings of ECHT '90). Cambridge: Cambridge University Press. 152-165.

McCall R, Morch A, Fischer G (1990b) Supporting Reflection-in-action in the Janus Design Environment. In Mitchell W, McCullough M, Purcell P (eds) (1990b) *The Electronic Design Studio.* Cambridge: MIT Press. 247-260.

McCall R, Schaab B, Schuler W, Mistrik I (1983) *Mikroplis User Manual.* Heidelberg.

McCall R (1989/90) Development of a Design Environment Integrating Dynamic Hypertext with CAD. Funded proposal to Colorado Institute for Artificial Intelligence.

McCall R (1990/91) Intelligent Hypertext as an Alternative to Expert Systems. Funded proposal to Colorado Institute for Artificial Intelligence.

McCall R (1991/92) Virtual Copies of Hypermedia Networks in a System for Design of Space-based Habitats. Funded proposal to Colorado Institute for Artificial Intelligence.

McCall R (1992/93) Intelligent Hypermedia Graphics in the Design of Space-based Habitats. Funded proposal to Colorado Advanced Software Institute.

McCall R (1993/95) Computer-Supported Knowledge Capture for the Design of Space-based Habitats. Proposal to Colorado Advanced Software Institute.

Minsky M (1985) *The Society of Mind.* New York: Simon and Schuster.

Mittal S, Boborow DG, Kahn KM (1986) Virtual Copies At the Boundary Between Classes and Instances. *OOPSLA '86 Proceedings.* 159-166.

Miyake N (1986) Constructive Interaction and the Iterative Process of Understanding. *Cognitive Science. 10.* 151-177.

Moore GT, Fieber JP, Moths JH, Paruleski KL (1991) Genesis Advanced Lunar Outpost II: A Progress Report. In Blackledge RC Redfield CL Seida SB (Eds.), *Space -- A Call for Action: Proceedings of the Tenth Annual International Space Development Conference.* San Diego, CA: Univelt, 55.

Nakakoji K (1993) *The Role of a Specification Component.* Ph.D. dissertation. Department of Computer Science. University of Colorado at Boulder.

Nardi B, Miller J (1990) The Spreadsheet Interface: A Basis for End User Programming. *Proceedings of Interact '90.* 977-983.

NASA (1989) *Space Station Freedom Man-Systems Integration Standards.* NASA-STD-3000 Volume I. Revision A. December 14, 1989. NASA.

NASA (1989) *Space Station Freedom Man-Systems Integration Standards.* NASA-STD-3000 Volume IV. Revision A. December 14, 1989. NASA.

Nielsen J, Frehr I, Nymand NO (1991) The Learnability of HyperCard as an Object-oriented Programming System. *Behavioral Information Technology. 10* (2) 111-120.

Nilsson N (1980) *Principles of Artificial Intelligence.* Palo Alto: Morgan Kaufmann.

Nobel DF (1984) *Forces of Production: A Social History of Industrial Automation.* New York: Knopf.

Norman D (1993) *Things That Make Us Smart.* Reading, MA: Addison-Wesley. In preparation.

Norman D, Draper S (1986) *User Centered System Design: New Perspectives on Human-Computer Interaction.* Hillsdale, NJ: Lawrence Erlbaum.

Palmer R (1969) *Hermeneutics: Interpretation Theory in Schliermacher, Dilthey, Heidegger and Gadamer.* Evanston: Northwestern University Press.

Papert S (1980) *Mindstorms: Children, Computers, and Powerful Ideas.* New York: Basic Books.

Plato (348 BC) *The Collected Dialogues of Plato.* E Hamilton & H Cairns (Eds.). New York: Pantheon. 1961.

Polanyi M (1962) *Personal Knowledge.* London: Routledge & Kegan Paul.

Putnam H (1967) The Nature of Mental States. In Block N (Ed.) (1980) *Readings in Philosophy of Psychology* (Vol. 1). Cambridge: Harvard University Press. First published as Psychological Predicates. In Capitan WH,. Merrill DD (Eds.) (1967) *Art, Mind and Religion.* Pittsburgh: University of Pittsburgh Press.

Putnam H (1988) *Representation and Reality.* Cambridge: MIT Press.

Quillian MR (1967) Word Concepts: A Theory and Simulation of Some Basic Semantic Capabilities. *Behavioral Science 12,* 410-430. In Brachman & Levesque (1985).

Raybeck D (1991) Proxemics and Privacy: Managing the Problems of Life in Confined Settings. In Harrison AA, Clearwater YA, McKay CP (Eds.) (1991) *From Antarctica to Outer Space: Life in Isolation and Confinement.* New York: Springer Verlag. 317-330.

Redmiles D (1992) *From Programming Tasks to Solutions—Bridging the Gap Through the Explanation of Examples.* Ph.D. dissertation. Department of Computer Science. University of Colorado at Boulder.

Reeves B (1993) *The Role of Embedded Communication and Embedded History in Collaborative Design.* Ph.D. dissertation. Department of Computer Science. University of Colorado at Boulder.

Resnick L (1991) Shared Cognition: Thinking as Social Practice. In Resnick L, Levine J, Teasley S (Eds.) (1991) *Perspectives on Socially Shared Cognition.* Washington, DC: APA. 1-22.

Richardson J (1991) *Existential Epistemology: A Heideggerian Critique of the Cartesian Project.* Oxford: Claredon Paperbacks.

Rilke RM (1912) *Duino Elegies.* Translation: Boston: Shambala. 1992.

Rittel H (1972) Second-generation Design Methods. In Cross (1984). 317-327.

Rittel H, Webber M (1973) Dilemmas in a General Theory of Planning. *Policy Science. 4*, 155-169. Alternative version as Rittel H, Webber M (1973) Planning Problems are Wicked Problems. In Cross (1984). 135-144.

Rorty R (1977) *Philosophy and the Mirror of Nature*. Princeton: Princeton University Press.

Schaab B, McCall R, Schuler W (1984) Mikroplis -- ein Textbank-Management-System. *Nachrichten für Dokumentation. 35* (6). 254-259.

Schank R (1982) *Dynamic Memory*. Cambridge: Cambridge University Press.

Schön D (1983) *The Reflective Practitioner*. New York: Basic Books.

Schön D (1985) *The Design Studio*. London: RIBA Publications.

Schön D (1992) Designing as Reflective Conversation with the Materials of a Design Situation. *Knowledge-Based Systems*, **5**, (3). 3-14.

Schutz A (1970) *Reflections on the Problem of Relevance*. New Haven: Yale University Press.

Searle J (1980) Minds, Brains, and Programs. *The Behavioral and Brain Sciences, **3**.*

Searle J (1983) *Intentionality: An Essay in the Philosophy of Mind*. Cambridge: Cambridge University Press.

Shipman F (1993) *Supporting Knowledge-Base Evolution Using Multiple Degrees of Formality*. Ph.D. dissertation. Department of Computer Science. University of Colorado at Boulder.

Simon H (1973) The Structure of Ill-structured Problems. *Artificial Intelligence. 4*. 181-200.

Simon H (1981) *The Sciences of the Artificial*. Cambridge: MIT Press.

Smith, BC (1991) The Owl and the Electric Encyclopedia. *Artificial Intelligence. 47*. 251-288.

Smolensky P, Fox B, King R, Lewis C (1987) Computer-Aided Reasoned Discourse, or How to Argue with a Computer. In Guindon R (Ed.) (1987) *Cognitive Science and its Implications for Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum.

Snodgrass A, Coyne R (1990) Is Designing Hermeneutical? Working paper. Faculty of Architecture, University of Sydney.

Stahl G (1975a) *Marxian Hermeneutics and Heideggerian Social Theory: Interpreting and Transforming Our World*. Ph.D. dissertation. Department of Philosophy. Northwestern University.

Stahl G (1975b).The Jargon of Authenticity: An Introduction to a Marxist Critique of Heidegger. *Boundary 2, III*, (2). 489-498.

Stahl G (1976) Attuned to Being: Heideggerian Music in Technological Society. *Boundary 2, IV*, (2). 637-664.

Stahl G (1991) A Hypermedia Inference Language as an Alternative to Rule-Based Expert Systems. Technical Report CU-CS-557-91. Computer Science Department, University of Colorado at Boulder. 1-23.

Stahl G (1992) A Computational Medium for Supporting Interpretation in Design. Technical Report CU-CS-598-92. Computer Science Department, University of Colorado at Boulder. 1-39.

Stahl G (1993a) Supporting Situated Interpretation. *Proceedings of the Cognitive Science Society: A Multidisciplinary Conference on Cognition*. Boulder. CO. June 18-21, 1993. 965-970.

Stahl G (1993b) Supporting Interpretation in Design. Submitted to *Journal of Architecture and Planning Research.* Special issue on Computational Representations of Knowledge.

Stahl G, McCall R, Peper, G (1992) Extending Hypermedia with an Inference Language: An Alternative to Rule-Based Expert Systems. *Proceedings of the IBM ITL Conference: Expert Systems (October 19-21, 1992).* 160-167.

Suchman L (1987) *Plans and Situated Actions: the Problem of Human Machine Communication.* Cambridge: Cambridge University Press.

Suchman L (1993) Response to Vera and Simon's Situated Action: A Symbolic Interpretation. *Cognitive Science. 17.* (1). 77-86.

Suchman L, Trigg R (1991) Understanding Practice: Video as a Medium for Reflection and Design. In Greenbaum & Kyng (1991). 65-90.

Sussman G, McDermott D (1972) *From PLANNER to CONNIVER: A Genetic Approach.* Montvale, NJ: AFIPS Press.

Tafforin C (1990) Relationships Between Orientation, Movement and Posture in Weightlessness: Preliminary Ethological Observations. *Acta Astronautica. 21.* 271-280.

Toulmin S (1958) *The Uses of Argument.* Cambridge: Cambridge University Press.

Vonnegut K (1952) *Player Piano.* New York: Avon.

Vygotsky LS (1978) *Mind in Society.* Cambridge: Harvard University Press.

Weizenbaum J (1976) *Computer Power and Human Reason: From Judgment to Calculation.* New York: Freeman & Co.

Winograd T, Flores F (1986) *Understanding Computers and Cognition: A New Foundation for Design.* New York: Addison-Wesley.

Winston PH (1981) *Artificial Intelligence.* Reading, MA: Addison-Wesley.

Wirth N (1988) From Modula to Oberon. *Software—Practice and Experience. 18.* (7). 661-670.

Wittgenstein L (1953) *Philosophical Investigations.* New York: Macmillan.

Wixon D, Holzblatt K, Knox S (1990) Contextual Design: An Emergent View of System Design. *CHI '90 Proceedings.*

Woods WA (1975) What's in a Link: Foundations for Semantic Networks. In: Brachman RJ, Levesque HJ (1985) *Readings in Knowledge Representation.* San Mateo, CA: Morgan Kaufmann. 217-242.