HUMAN-COMPUTER INTERACTION, 1990, Volume 5, pp. 305-344

Designing the Design Process: Exploiting Opportunistic Thoughts

Raymonde Guindon
Microelectronics and Computer Technology Corporation

ABSTRACT

This study shows that top-down decomposition is problematic in the early stages of design. Instead, an opportunistic decomposition is better suited to handle the ill-structuredness of design problems. Designers are observed interleaving decisions at various levels of abstraction in the solution decomposition. The verbal protocols of three professionals designing a software system of realistic complexity are analyzed to determine the frequency and causes of opportunistic decompositions. The sudden discovery of new requirements and partial solutions triggered by data-driven rules and associations, the immediate development of solutions for newly discovered requirements, and drifting through partial solutions are shown to be important causes of opportunistic design. A top-down decomposition appears to be a special case for well-structured problems when the designer already knows the correct decomposition. Two cognitive models are briefly discussed in relation to opportunistic design. Finally, implications for training, methods, and computational environments to support the early stages of design are outlined.

Author's present address: Raymonde Guindon, Computer Science Department, Stanford University, Stanford, CA 94305.

CONTENTS

The proper study of mankind is the science of design.
Simon, 1969/1981, p. 159

1. A STUDY OF SOFTWARE SYSTEM DESIGN

In the early stages of software system design, a designer must transform an informal, incomplete, and ambiguous specification of the requirements into a high-level design expressed in a formal or semiformal notation. The requirement specifications define the external, functional, and performance requirements, as well as exception handling and acceptance criteria. A high-level design, sometimes called external design, refines those requirements (i.e., what the system should do) and describes the main software subsystems, the information flow and control, the conceptual data structures, and the interfaces between subsystems (i.e., how the system will satisfy these requirements; Fairley, 1985). The most expensive errors to correct in a software development project are those made during high-level design (Dunn, 1984).

Yet, high-level design has seldom been empirically studied and is poorly supported by software tools and environments available today.

This study is a descriptive one that examines the applicability of top-down approaches to the early stages of software design. This section presents an analysis of high-level design tasks based on Simon's (1973) notion of ill-structured problems. It then reviews previous cognitive studies of design and informal observations of practitioners and shows that they disagree with respect to the applicability of the prescribed top-down design and the waterfall model. Sections 2.1 to 2.4 present the method and the categories of design behaviors coded in the protocols. Section 3 presents results from the protocol analysis. Section 3.1 presents data on the dynamics of the design process and on the frequency of deviations from a top-down process. Indeed, the design process greatly deviates from the top-down approach. Section 3.2 determines the types of design activities that deviate from a top-down process and their frequency. Section 3.3 illustrates many examples of these deviations and analyzes their causes and underlying mechanisms. Because design tasks belong to the domain of bounded rationality, these deviations are explained in terms of the interplay between basic psychological mechanisms and the designer's intendedly rational endeavors (Newell & Card, 1985). The analyses show that these deviations are not special cases due to bad design habits or performance breakdowns but are, rather, a natural consequence of the ill-structuredness of problems in the early stages of design. Section 4.1 discusses these findings in relation to previous studies and reconciles differences in results. Section 4.2 presents a definition of opportunistic design and discusses two cognitive models to account for the results in this study. Section 4.3 presents implications for training, methods, and environments to support the early stages of design. This article focuses on how designers apply their knowledge and is based on two earlier preliminary articles (Guindon & Curtis, 1988; Guindon, Krasner, & Curtis, 1987). Another related article (Guindon, in press) describes the sources of knowledge exploited by experts during design.

1.1. Design as an Ill-Structured Problem

Simon (1969/1981) described the relation between *natural sciences* and a *science of design.* Natural sciences are concerned with how things are in the world. A design process, however, is not a natural phenomenon, even though it is strongly influenced by natural phenomena such as cognitive, social, organizational, and physical laws. Humans design the design process. A science of design should be concerned with how the design process ought to be to best accommodate the environment's constraints, including the designer's own cognitive and social constraints. Two topics of a science of design are

particularly relevant to this study. The first is the structure of complex artifacts and their impact on the design process; the second is the set of principles that should be followed to control the selection and ordering of actions during design.

Simon (1973) characterized design problems as ill structured. Three important sufficient features of ill-structured problems are: (a) incomplete and ambiguous specification of goals, (b) no predetermined solution path, and (c) the need for integration of multiple knowledge domains. These features make design problems particularly difficult.

As practitioners acknowledge, an intrinsic aspect of system design is the incompleteness and ambiguity of the requirements or goals (e.g., see Meyer, 1985; Parnas & Clements, 1986; Swartout & Balzer, 1982). In other words, software design problems have poorly defined goals and no well-defined criteria to evaluate the solution (Simon, 1973). Therefore, design involves problem structuring. Problem structuring is the process of discovering missing information, such as problem goals and evaluation criteria, and using it to define a problem space (Simon, 1973).

System design often involves novelty. Even though the designer may be thoroughly familiar with the design process itself, there may not be any precedent in the literature for the system to be designed—It may be a new technology. More frequently, the system may simply involve some novelty in an otherwise well-understood problem. The novelty may range from a novel combination of requirements for a familiar type of system in a familiar problem domain to an unfamiliar type of system in an unfamiliar problem domain. As a consequence, there is often no predetermined solution path from the requirements to the finished artifact (Newell, 1969; Nii, 1986; Reitman, 1965; Rittel, 1972; Simon, 1973). Thus, system design frequently requires the creation of new solutions interleaved with the application of known solutions.

The design of a software system typically involves the integration of multiple sources of knowledge—the problem domain, software system architecture, and computer science. For example, the design of a control system for the elevators in a building includes integrating knowledge about how users might make vastly different requests for service in real situations, about servicing asynchronous requests, about interactions between software and hardware, about control and communication schemes for multiple processors operating concurrently, and about scheduling.

Structuredness is not a dichotomy but rather a continuum (Simon, 1973). Problems fall everywhere in the continuum. System design is less structured than such problems as checkers or Tower of Hanoi. In particular, software system design is generally less structured than program and algorithm designs. The goals of programs and algorithms tend to be better specified and often require knowledge from fewer sources. Moreover, structuredness is not

only a problem feature, but also a psychological feature: The knowledge a designer has about similar or related problems can readily impose a great deal of structure to a problem. Reitman (1965) pointed out that design problems are problems with a large number of open constraints—parameters whose values are left unspecified in the problem statement. Solving an ill-structured problem is partly a process of resolving these constraints. Simon (1973) emphasized that one of the main roles of the problem solver is to increase the structuredness of the problem by resolving these constraints: Simon wrote, "There is much merit to the claim that much problem solving effort is directed at structuring problems, and only a fraction of it at solving problems once they are structured" (p. 187). Thus, problem structuredness is also a function of how many structuring activities the problem solver has already performed. Therefore, one expects design behaviors to vary according to the incompleteness and ambiguity of the problem specification, the amount of knowledge from different domains that need to be integrated, how familiar the designer is with a particular problem, how many structuring activities have already been performed, and the interactions between these variables.

1.2. Empirical Evidence for Prescriptive Design Models

An example of a prescriptive model of the software design process is the top-down model. In top-down design, aspects of the overall system are designed first; then the system is progressively decomposed into subsystems at increasingly greater levels of detail. Stepwise refinement is a specialization of top-down approaches in which the designer must demonstrate that each successive addition to the design postpones detailed design decisions as long as possible (see Dahl, Dijkstra, & Hoare, 1972; Wirth, 1971). Top-down design is related to a popular paradigm in planning research. Sacerdoti's (1975) NOAH program implements a successive refinement approach to planning. In NOAH, problems are specified in terms of high-level goals that determine general actions and are successively expanded into lower level goals that determine more elementary actions.

Jeffries, Turner, Polson, and Atwood (1981) studied two novices and four experts designing a book-indexing program—Given a set of words and the source text of a book, generate an index for the book. Jeffries et al. argued that the usual order in which a designer should attempt subproblem solution is top down, breadth first, ensuring that all information about the current state of the design is available to the next lower level of abstraction. Jeffries et al. observed that both experts and novices tend to apply a top-down, breadth-first decomposition. Jeffries et al. acknowledged that this similarity in solution decomposition was probably due to the straightforwardness of the problem, which required only upper undergraduate level computer science

knowledge. Indeed, the book-indexing problem is relatively well structured in terms of Simon's analysis: It has well-defined goals; it presents little novelty to the designers; and it requires the integration of few knowledge sources. Nevertheless, Jeffries et al. observed some deviations from a top-down, breadth-first decomposition when a subproblem appeared critical, very difficult, or had an immediately known solution. They also reported that the protocol of one of their designers, a professional systems analyst with more than 10 years of experience, was interspersed with digressions that related to subproblems at other levels and at other positions in the problem.

Adelson and Soloway (1984, 1985) studied three experts and two novices designing systems with which they had differing familiarity—an electronic mail system with seven well-defined mail operations, a library record-keeping system, and an interrupt handler. They observed that the expert designers' development of the solution was systematic and balanced—The designers developed each component of the design solution so that none of them was defined in significantly more detail than the others at the same level of abstraction. Adelson and Soloway explained that this is achieved by a process that compares the current solution with the goal and selects a subproblem to solve only at such a level of detail and granularity that balanced development is enforced. Adelson and Soloway speculated that without balanced development, it would be impossible to mentally simulate a solution because its parts would have been defined at different levels of detail. The expert designer of the interrupt handler, however, violated a balanced development strategy by exploring in detail the only unfamiliar part of the artifact, a brand of chip. Adelson and Soloway suggested that designers undertake unbalanced development and detailed explorations only when they already have a mental model of the system they are building. In terms of Simon's (1973) analysis, the electronic mail and the library record-keeping systems had well-defined goals and presented little novelty to the designers. On the other hand, the unfamiliar part of the interrupt handler presented novelty to the expert.

Kant and Newell (1984) studied two PhD-level computer scientists designing an algorithm to an unfamiliar problem—the convex hull inclusion problem. Given a set of points in a plane, the algorithm had to generate the smallest subset of these points that, when connected in a convex polygon, contained all the other points. The designers were very skilled in algorithm design, but they had never solved this particular problem. The subjects quickly adopted a problem-solving schema, such as divide and conquer or generate and test, to elaborate the algorithm in the algorithm-design space. The rest of the time was mostly spent successively refining the initial problem-solving schema. Interestingly, the subjects also worked, though much less frequently, in a geometry space. This process sometimes led to the discovery of new solutions. Kant and Newell suggested that the interplay between problem solving in the two problem spaces permits the process of

discovery. In terms of Simon's (1973) analysis, the convex hull problem had well-defined goals, required the integration of knowledge from the algorithm and geometry domains, and was novel for these subjects.

Another important point is that these studies seem to consider the specification of the design problem as a fully completed process preceding design. This agrees with the waterfall models of the software life cycle where requirement specification should be completed before design (e.g., Royce, 1970). Problem specification is defined as the part of problem structuring where incompletely specified design goals and evaluation criteria are specified. Problem specification relies on knowledge from the domain of application (e.g., for the convex hull problem, the domain is geometry). Kant and Newell (1984) reported that their subjects sometimes used a geometry problem space for two purposes: (a) When solution retrieval fails, the designer tries test case execution, and (b) when comparing the current algorithm solution to the goal. However, Kant and Newell did not report inferences of requirements—design goals or evaluation criteria for the quality of the solution—as a result of using the geometry problem space. Jeffries et al. (1981) described a strategy called problem solving by understanding: When subjects were unable to develop a solution for a subproblem, they used knowledge from the problem area and computer science to refine their understanding. However, Jeffries et al. did not report inferences of problem goals or evaluation criteria as a consequence of problem solving by understanding.

Carroll, Thomas, and Malhotra (1979) and Malhotra, Thomas, Carroll, and Miller (1980), however, found that customer-designer dialogues consisted of a sequence of cycles, each consisting of requirement elaboration, solution generation, and solution evaluation. Carroll and Rosson (1985) described design as a nonhierarchical process involving the development of tentative interim or partial solutions and involving the discovery of new goals. The deviations from top-down models in their study, however, could be attributed to the inclusion of a customer in the process or to the fact that these were dialogs. Therefore, these data do not provide definitive evidence for the design process at the individual level.

Practitioners such as Parnas and Clements (1986) argued that the design process cannot follow a top-down approach. They described the following obstacles to a top-down process: (a) requirements are usually incomplete and vague, (b) realistic projects are of such complexity that designers cannot comprehend and keep track of all the details, and (c) designers are often biased by preconceived design ideas of varying relevance. Parnas and Clements pointed out, however, that the products of design, the design documents and software, should be expressed and represented as if the software design process had been a balanced and systematic process. The following quote by Mills (1986), a proponent of top-down approaches, is

particularly informative in this regard: "The top-down [process] . . . does not claim that the thinking should be top-down. Its benefit is in the later phases of program design, after the bottom-up thinking and perhaps some trial coding has been accomplished" (p. 61).

To summarize, the design process of experts emerging from these studies is one of successive refinements in a top-down, breadth-first manner, where problem specification precedes solution development. However, these studies largely used problems that could be considered well structured in terms of design problems. Nevertheless, deviations from a top-down process have been sporadically observed in the experts in these studies (a) when the artifact presented novelty to the designer; (b) when the problem required the integration of multiple knowledge sources; and (c) when a subproblem appeared critical, very difficult, or had an immediately known solution. Moreover, software design practitioners, who deal with realistic design tasks, have expressed many reservations about the top-down approach. This concern is ironical because the top-down approach is meant to manage the very complexity that is naturally found in realistic systems.

The top-down approach is a prescriptive method whose goals are to manage complexity and produce artifacts that are easy to understand, test, verify, and modify. Nevertheless, one can expect deviations from the top-down approach even in experts (and such were reported in the reviewed studies). But are such deviations simply inadequate applications of the top-down approach due to idiosyncratic design practices or uninteresting performance breakdowns? After all, Dijkstra (1976) called structured programming a discipline, suggesting that this is not the most natural way of programming. Or, are these deviations due to an unavoidable inability to apply the top-down approach when the problem is unstructured, as in the early stages of design? How applicable are top-down approaches for problems in need of structuring? This study shows that the design process frequently deviates from a top-down approach. But, more important, it shows that these deviations are not noise or special cases resulting from bad design habits or performance breakdowns. Rather, they are a natural consequence of the ill-structuredness of problems in the early stages of design.

2. METHOD

2.1. Participants

Design protocols were initially collected from eight designers by Herb Krasner, a software engineer interested in empirical issues. He selected three protocols to be analyzed in depth on the basis of the following criteria: (a) the designers had advanced degrees and many years of professional experience,

(b) the designers were considered by their peers and managers to be very experienced and competent, and (c) each designer exhibited one of three styles of design where each style had been observed in two or three designers in the initial eight designers. The three styles were described by Herb Krasner as (a) being guided by a software design method, (b) being guided by past experience with related systems in different problem domains, and (c) being guided by a programming paradigm based on a high-level language. These three styles correspond, respectively, to Designers 1, 2, and 3. The designers were selected prior to any in-depth analysis from the first researcher. No further analyses of the protocols were made until the analysis of these three protocols was reassigned to the author.

Designer 1 had a master's degree in software engineering, was the top student of his class, and had 5 years of professional experience in designing real-time systems. Designer 2 had a doctoral degree in electrical engineering and had 10 years of professional experience with concurrent systems and communication systems. Designer 3 had temporarily suspended his doctoral training in computer science to work in an industrial setting for 3 years. He had experience with logic programming and rapid prototyping. All designers had learned structured programming and top-down design as part of their formal education or their job training. The protocols of Designers 1 and 2 were fully analyzed, the protocol of Designer 3 was analyzed to a lesser extent, and the remaining five protocols were only cursorily assessed. Although the protocols from all designers exhibited the same general features, we report detailed analyses only from Designers 1 and 2.

One issue in typical protocol studies is generalizability of the results when the number of participants is small. How representative is our sample of participants of the larger population of software designers? There is simply no reliable way to answer this question given the current maturity of the field. That is, no population data are available against which to compare our sample. There is no standard type of individual who becomes a software designer. Educational background, work experience, job setting, and skills radically differ. To deal with these issues, two active, professional, and experienced designers well respected by their colleagues were selected. These designers are representative of at least some segment of the population of actual designers.

2.2. Design Problem

The Lift Control Problem is a standard problem in software specification and software requirements research. None of the designers had solved this problem before the study. The problem statement is given in Figure 1. The goal is to design the software to control the movement of N lifts between M

Figure 1. ***The N-Lift problem statement.***

An N-lift system is to be installed in a building with M floors. The lifts and the control mechanism are supplied by a manufacturer. The internal mechanisms of these are assumed (given) in the following problem –
Design the logic to move lifts between floors in the building according to the following rules:

1. Each lift has a set of buttons, one button for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is canceled when the corresponding floor is visited (i.e., stopped at) by the lift.
2. Each floor has two buttons (except ground and top), one to request an up lift and one to request a down lift. These buttons illuminate when pressed. The buttons are canceled when a lift visits the floor and is either traveling in the desired direction or visiting the floor with no requests outstanding. In the latter case, if both floor request buttons are illuminated, only one should be canceled. The algorithm used to decide which to service first should minimize the waiting time for both requests.
3. When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests (or model a "holding" floor).
4. All requests for lifts from floors must be serviced eventually, with all floors given equal priority (Can this be proved or demonstrated?).
5. All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the direction of travel (Can this be proved or demonstrated?).
6. Each lift has an emergency button that, when pressed, causes a warning signal to be sent to the site manager. The lift is then deemed "out of service." Each lift has a mechanism to cancel its out-of-service status.

floors, given the constraints expressed in the problem statement. The problem imposes realistic constraints, such as minimizing waiting time for requests, giving all floors equal priority, and the like.

Again, there is an issue of generalizability and validity. How representative is the Lift problem of problems typically given to designers? The Lift problem certainly has the ecologically valid features typically associated with realistic design problems (Simon, 1973). The Lift problem specification is informal and therefore incomplete and ambiguous. It is also a knowledge-rich problem because it requires the integration of many sources of knowledge. In a taxonomy of software design problems, the control of a lift system for N lifts can be categorized under three types of systems: a reactive system, an embedded system, and a concurrent system. Therefore, the design of a lift system requires the integration of knowledge of scenarios of usages from end users, of servicing asynchronous input under real-time constraints, of the interaction between hardware and software, and of concurrent processes. There is also novelty because none of our designers had designed an identical system in the past, although they had worked on related real-time, concurrent, or embedded systems.

Many representative applications share design issues with the Lift problem. For example, user interfaces are reactive systems, having to respond to asynchronous user input under real-time constraints. Software used to control advanced graphics packages, CAD systems, and three-dimensional mechanical design systems, are embedded systems interacting with multiple input and output devices. The growing interest in electronically supported group work has led to the development of *groupware,* software systems that are not only reactive and embedded but also must handle concurrent events. Lewis (1990) also pointed out the ubiquity of problems that involve concurrent activities and the necessity of studying the design of concurrent systems. Hence, the Lift problem is an instance of an important and frequent type of software system.

2.3. Procedure

Thinking-aloud reports were collected by an experimenter from the designers as they designed the logic for the Lift problem. The designers exhibited no difficulty in verbalizing their thoughts, and they produced dense verbalizations with few moments of silence. The designers' natural ease of verbalization may be attributed to the fact that design often occurs in teams, and designers are comfortable expressing their thoughts aloud. When the designers fell silent, they were gently prompted by the experimenter to continue talking.

The designers were given up to 2 hr to produce a high-level design solution that was in a form and level of detail that could be handed off to a competent system programmer to implement. The participants were videotaped and supplied with paper and pencils to work their solution. The designers were free to write anything they wished on the paper provided—notes to themselves, tentative solutions, requirements, and so on. The notes and diagrams produced by the participants were regularly time stamped by the experimenter. The transcript of each participant was also time stamped, and the written notes and diagrams were included in the transcript. Because this procedure is comparable to the verbal protocol procedure used in other design studies, it permits comparison between the results of this study to those of these other studies.

In debriefing sessions following the protocol collection, the designers were asked to comment on the naturalness of the experimental situation. The designers reported that the problem was not unlike the type of design problems they had been given in the past—sketchy, incomplete, and ambiguous. They also commented that, in the early stages of design, they only used paper and pencil to jot down ideas, notes, sketches of design, and so on. Designers mentioned that they were used to designing under severe time

constraints. They commented that in the field, however, they were freer to interrupt design to seek additional information from colleagues, customers, and reference material. For example, Designer 2 frequently wanted to discuss the requirements with the experimenter, as this represented his normal mode of designer-client interactions. They also commented, on the other hand, that they felt more constrained to go by the book and follow accepted design methods and practices in the experimental session than they would in the field.

2.4. Analysis of the Verbal Protocols

Analysis Procedure

The process of protocol analysis was divided into the following major steps. First, the videotape was reviewed and the transcript of each designer was read by four researchers with different perspectives and backgrounds—a preliminary brainstorming analysis. One of the researchers was a cognitive psychologist, two were researchers with a background in artificial intelligence and an interest in software tools to support designers, and the last one had a background in software engineering and in the development of large systems. The researchers were free to note whatever they felt was significant about the design process of an individual designer. The notes were then shared and discussed at length among the researchers. This brainstorming analysis helped ensure that findings from the study were shared very rapidly with those involved in building tools. The analysis also helped ensure that design activities that might have been considered as noise or uninteresting under one theoretical framework could reveal their significance from another perspective.

Second, following this preliminary analysis, about 1 month after protocol collection, the videotape was viewed with the participant designer in a prompted review session. The participant designer was free to stop the videotape and note whatever was significant from his perspective. The researchers could also stop the videotape at any point and question the designer. These questions were particularly useful in probing for the specialized knowledge that designers had brought in during design but that was only alluded to in the verbalizations. They were also useful in uncovering the rationale underlying certain design decisions that had not been verbalized during the session.

Then, an analysis scheme was developed iteratively as additional and more detailed analyses of the protocols were performed. For each episode, a template of attribute-value pairs was filled out. The template specified the type of design activity and its knowledge domain: Lift domain scenarios, requirement understanding and elaboration, and design solution develop-

Figure 2. **Encoded design activities with their attributes and values.**

Design Activities	External Representation	Simulation	Levels of Abstraction	Inference/Adding New Requirement
Lift domain scenarios	Yes/No	Yes/No	N/A	N/A
Requirement understanding and elaboration	Yes/No	N/A	N/A	Yes/No
Solution development	Yes/No	Yes/No	High Medium Low	N/A

ment. If the design activity was development of a solution, the template specified its level of abstraction as high, medium, or low. If the design activity was about requirements, the template specified whether the designer had added or inferred a requirement or simply was understanding or reviewing the requirements. The template also specified whether the designer relied on external representations (e.g., diagrams, notes).

Finally, each designer's protocol was analyzed in depth by two people. The analyses were then compared and conflicts resolved. This comparison also led to the discovery of new categories or features of episodes.

Protocol Coding Categories

This section describes in greater detail the types of design activities and their associated knowledge domains that were encoded as episodes in the protocol. Figure 2 gives an overview of the design activities and their attribute-value pairs.

One of the designers' activities is the retrieval or simulation of scenarios in the problem domain (called the Lift domain). By problem domain we mean a subset of the real world with which a computer system is concerned, but not the design solution describing the computer system itself.[1] A lift system is concerned with lifts, floors, passengers, waiting time of passengers on the floors, safety of passengers, and so on. A Lift scenario could describe, for example, an interesting situation where there is a request to go from Floor 2 to Floor 4 and two lifts that could service it—a lift going up from Floor 1 and a lift going down from Floor 4 to Floor 1. The Lift scenarios can be accompanied by diagrams and notes to help the simulations.

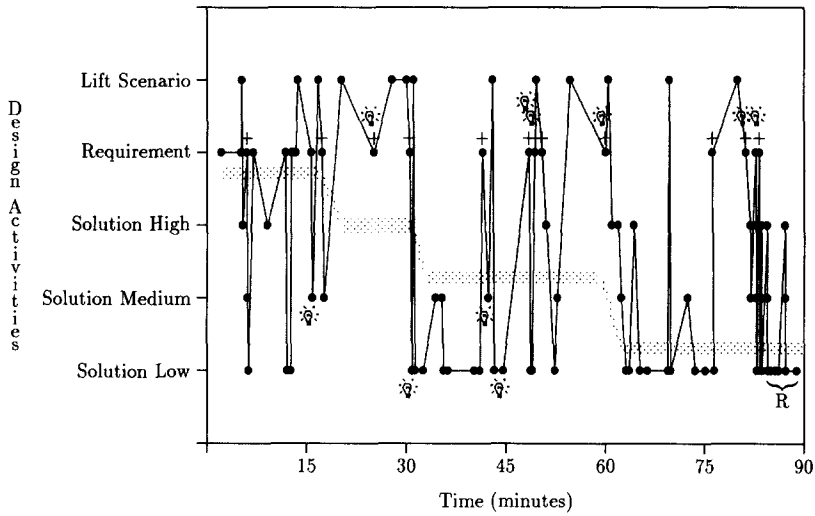
Another of the designers' activities is the understanding and elaboration of

[1] The term *application knowledge* is sometimes used to mean problem domain knowledge. However, the term is also used to refer to areas such as graphics, databases, networks, that is, applications of computer science. The term *problem domain* is preferred because it does not present this ambiguity.

the requirements in the informal specification. By elaboration of the requirements, we mean any activity whose purpose is to decrease the incompleteness and ambiguity of the informal specification of the requirements. In particular, this includes the inference of constraints that followed from the informal specification and the addition of new requirements. An inferred constraint is not explicitly given in the requirements but it can be deduced as necessary or plausible from the informal specification and one's knowledge of the problem domain. Inferred constraints include, at least, inferred relationships between objects, inferred properties of an object, inferred actions of an object, inferred objects, and inferred test cases. For example, as a result of the Lift scenario just described, a designer inferred a test case missing from the specification: A passenger makes a request to go in a particular direction, up or down, but, once inside a lift, the passenger requests the lift to go in the opposite direction. Added requirements are plausible or desirable requirements missing from the specification that cannot be deduced logically from the informal specification. For example, a designer decided that a lift system should be very safe and added to the requirements that the design should satisfy a high level of reliability.

Another design activity is the development of the design solution. This includes the representation, the addition of new partial solutions, the mental or external simulation, the evaluation, and the debugging of the design solution. During solution evaluation, a designer weighs the pros and cons of alternative solutions. For example, a designer contrasted a centralized and a distributed control solution in terms of reliability and ease of implementation. During solution simulation, the designer evaluates the internal consistency, the correctness, and the completeness of the solution with respect to the requirements, whether they be given, inferred, or added. It is important to point out that the uses of test cases from the Lift domain knowledge are considered an integral part of solution simulations and are not considered shifts to the Lift domain in the results presented in the next section. The level of abstraction of a partial solution was categorized as high, medium, or low. By level of abstraction, we mean the hierarchical partitioning of the functions accomplished and information processed by the software system found in the design solution at the end of the session. Indeed, even if a design solution is not decomposed following a top-down approach, one can establish a hierarchical partitioning in the final solution on the basis of system–subsystem relationships. The high level of abstraction concerns the control and communication schemes adopted (e.g., central vs. distributed) and how the individual lift functions are handled. The medium level of abstraction concerns how these functions are divided into subfunctions. The low level of abstraction concerns how these subfunctions are further subdivided and also includes detailed design—how functions are realized in the hardware, details about the data structures, and detailed algorithms.

Figure 3. **Shifts in design activities and levels of abstraction of Designer 1. Plus signs indicate newly inferred or added requirements. Light bulbs indicate sudden discovery of partial solutions or requirements. The region marked by *R* indicates the period of solution review.**



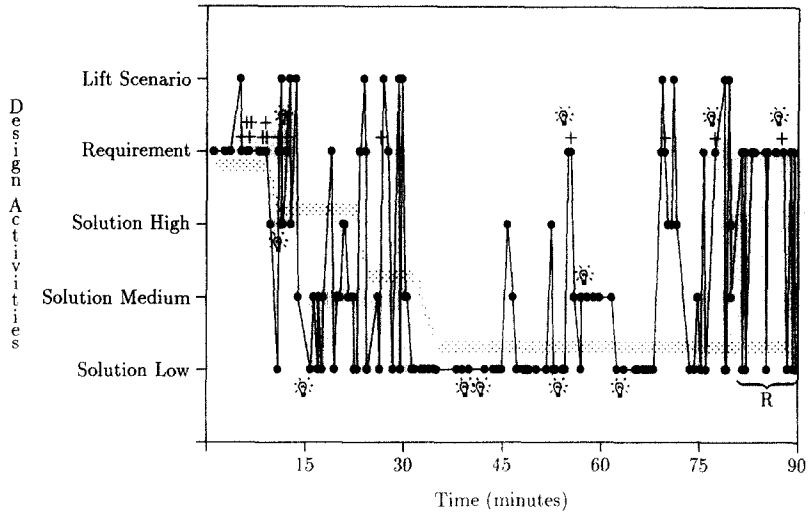
3. RESULTS AND OBSERVATIONS

Section 3.1 describes the general dynamics of the design decomposition over the complete session. It shows that the decomposition greatly deviates from a top-down process. Section 3.2 categorizes the design activities in terms of those which follow a top-down process and those which do not. Section 3.3 explores the causes of the deviations. It shows that these deviations are not uninteresting special cases but are an intrinsic consequence of the ill-structuredness of design problems.

3.1. The Dynamics of the Design Decomposition

Figures 3 and 4 are constructed from the protocol analysis in the following fashion. For each design activity or episode in the protocol, a node is drawn on the figure. On the *y* axis are given the three types of design activities: Lift domain scenarios; understanding and elaboration of the requirements; and development of the design solution at a high, medium, or low level of abstraction. Plus signs at the requirement level indicate an inference or the addition of a requirement. Light bulbs indicate sudden insights. Time since the beginning of the session is represented on the *x* axis. For readability, some of the episodes that were contiguous and very short were slightly spread out

Figure 4. **Shifts in design activities and levels of abstraction of Designer 2. Plus signs indicate newly inferred or added requirements. Light bulbs indicate sudden discovery of partial solutions or requirements. The region marked by *R* indicates the period of solution review.**



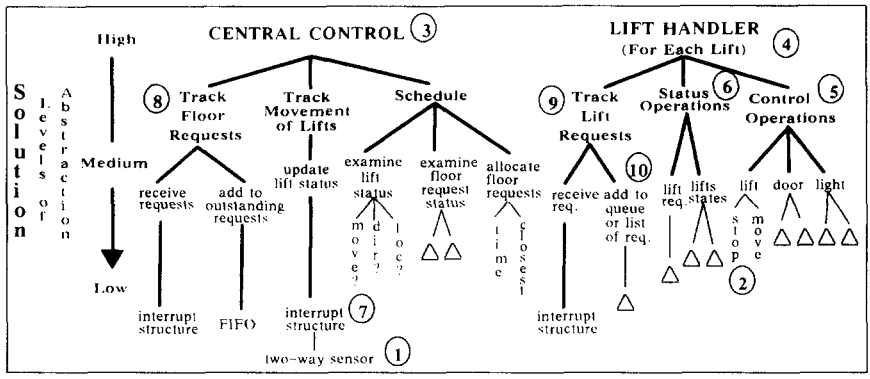
and displayed within ± 2 min of their occurrence in the session. The shifts between design activities in these three knowledge domains and levels of abstraction are displayed for the complete sessions of Designers 1 and 2.

The gray shadings in Figures 3 and 4 trace, for illustrative purposes, what the shifts might be like if the designer followed a top-down process preceded by a complete problem specification: First, the designer understands and elaborates the design goals with the help of Lift domain scenarios, and then develops the design solution at a high level, then at a medium level, and then at a low level of abstraction. The gray shadings are roughly drawn to visually maximize the fit between the actual, observed process and the prescribed process (e.g., the bar tends to follow the high frequencies of nodes at a given level and change levels when they decrease).

Figures 3 and 4 show that Designers 1 and 2 frequently departed from a top-down, breadth-first decomposition of their solutions. The designers expanded their solutions by rapidly shifting between levels of abstraction and by developing low-level partial solutions prior to a high-level decomposition. Moreover, the designers interleaved problem specification, that is, the inference of new requirements, with solution development throughout the session. In other words, designers interleaved problem structuring with solution development.

Figures 3 and 4, however, do not reveal the complete extent of deviations from a top-down approach. It is also necessary to present the order of

Figure 5. **Design solution decomposition of Designer 1 and shifts between partial solutions and levels of abstraction for a 10-min segment.**

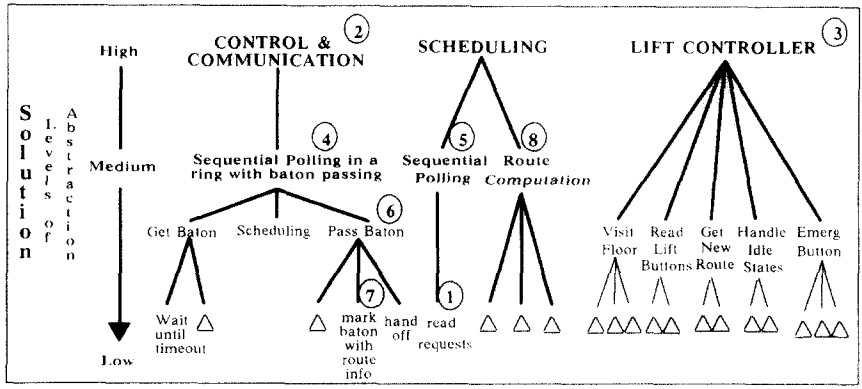


development of partial solutions in different parts of the solution decomposition. Figures 5 and 6 give, in the form of a structure chart, the functional decomposition of the final solutions for Designers 1 and 2. Owing to lack of space in the figures, the partial solutions with long labels are indicated by small triangles. The circled numbers in Figures 5 and 6 indicate the order in which partial solutions were developed for a representative 10-min segment. Specifically, Figures 5 and 6 detail the shifts between partial solutions in different subsystems.

Figures 5 and 6 show that Designers 1 and 2 deviated from a top-down, breadth-first decomposition: They shifted rapidly between partial solutions in different parts of the solution decomposition and developed partial solutions at various levels of abstraction without prior decomposition at a higher level.

Although Figures 3 and 4 show frequent departures from a top-down process, they also show some evidence that designers partly followed a top-down process. For example, Designer 1 had a cluster of problem-specification activities at the beginning of the session and a cluster of low-level solution development at the end of the session. Designer 2 had a cluster of problem-specification activities, followed by a cluster of high-level solution development, then a cluster of medium-level solution development, and finally a cluster of low-level solution development. Likewise, in Figure 5, Designer 1 developed two high-level subsystems in Events 3 and 4; then developed two medium-level subsystems in Events 5 and 6 in a top-down, breadth-first manner; then deviated and developed an unrelated low-level partial solution in Event 7; then returned to two medium-level subsystems in Events 8 and 9. As a consequence, more detailed analyses are presented in the next section that categorize and quantify the design activities in terms of those that deviate from a top-down process and those that follow such a process.

Figure 6. **Design solution decomposition of Designer 2 and shifts between partial solutions and levels of abstraction for a 10-min segment.**



3.2. Frequencies of Deviations From Top-Down Decomposition

For purpose of reporting the data, the design activities are broken down into three classes: (a) development of solution, (b) evaluation and debugging of solution, and (c) understanding and elaboration of the requirements (problem specification). The categorization of activities as balanced or unbalanced is summarized in Figure 7. The categorization rules are further described in the following paragraphs. Note that the notion of balanced and systematic has been expanded beyond the definition given by Adelson and Soloway (1984, 1985) to include a greater percentage of design activities. The justification for this expansion is clarified as the rules are described.

Because this is an analysis of the recurrent causes for deviations from a balanced process for both designers, and not an analysis of individual differences, the percentages are reported over the total number of design activities for the two designers. In fact, both designers exhibited large percentages of deviations from a balanced development—Designer 1 had 57%, and Designer 2 had 42% of unbalanced activities. Both designers had similar frequency breakdowns of these activities. A later section briefly describes interesting differences between designers. The figures report percentages over the total number of activities described in each figure. Figure 8 displays overall percentages of balanced and unbalanced activities collapsed over the two designers for a total of 256 design episodes. It shows that 52% of the design activities were balanced, whereas 47% were unbalanced. The high percentages of unbalanced design activities warrant a closer analysis of these deviations in Figures 9 and 10.

As can be seen in Figure 9, 53% of the solution development activities were

Figure 7. **Summary of categorization rules for unbalanced and balanced episodes.**

Design Activities	Unbalanced	Balanced
Solution Development	Drifting	Design schema
	Immediate recognition of solution in another part of the problem	Design method or notation
	By simulating Lift scenarios	Problem solving schema
	Immediate solution development for an inferred requirement	Design heuristics
Solution Evaluation	If solution is unbalanced	Test cases
		Systematic requirement review
		If solution is balanced
Requirements	Inferences and additions	Systematic strategy
		By simulating Lift scenarios

Figure 8. **Percentages of balanced and unbalanced design activities.**

Design Activities	Balanced	Unbalanced
Understand and elaborate the requirements	9	10
Develop solution	23	27
Simulate and debug solution	20	10
Total	52	47

unbalanced. The development of the design solution was considered unbalanced according to the following rules:

1. if the designer drifted, that is, elaborated a sequence of partial solutions, each leading to the next one, at much greater level of detail than the rest of the solution;
2. if the designer immediately recognized a solution for another part of the problem and shifted to work on this other part of the problem;
3. if the designer simulated Lift scenarios to develop a partial solution (in the waterfall model, the requirement analysis and associated use of problem domain knowledge should be completed before design starts);

Figure 9. **Percentages of balanced and unbalanced design activities for solution development.**

Design Activities	Balanced	Unbalanced
Develop solution by		
Top-down approach with specialized design schema	20	
Other methods (e.g., JSD), design notations, general problem-solving schema, problem-solving or design heuristics	8	
Drifting		18
Solution recognition in other part of problem		12
Lift domain scenarios		9
Immediately handling added requirements		7
Strategically developing at lower level		1
Meeting listed requirements	5	2
Returning to an interrupted partial solution		
Top-down approach with specialized design schema	9	
Immediately handling added requirements		2
At a higher level of abstraction than current solution (recovering from drift or other)	6	2
Total	48	53

4. if the designer immediately developed a partial solution for a new requirement he had just discovered, as opposed to take note of it and handle it later; and
5. if the designer strategically decided to violate balanced development and performed detailed explorations to get new ideas on the design solution.

Examples of each of these deviations are given in the next section with an analysis of some of their underlying cognitive mechanisms.

Solution development was considered balanced if it could be accounted for by the designer following:

1. a specialized design schema abstracted from designing related systems,
2. a design method or a design notation,
3. a general problem-solving schema, and
4. a problem-solving or design heuristic.

A design schema is a knowledge structure representing abstractions from the designs of related systems. For example, it may represent the similarities in overall design decomposition between a Lift system and a taxi cab dispatcher. A design method, by definition, dictates or suggests a sequence of activities to be performed. For example, Designer 1 partly followed a data

structure-oriented design method called Jackson System Development (Jackson, 1983). Design activities were also considered balanced when motivated by the use of design notations, such as state-transition diagrams, data-flow diagrams, or structure charts. Design activities that could be attributed to following a general problem-solving schema, such as divide and conquer, were considered balanced. For example, Designer 1 decided to divide the problem into servicing the requests made at the floor and servicing the requests made inside the lifts. Activities were also considered balanced if they could be attributed to following general problem-solving and design heuristics such as, "Consider a simpler problem" (e.g., solve the problem for one lift, then expand the solution to N lifts); "Keep the design solution as simple as possible"; "Keep the design solution parts as consistent as possible" (implement similar functions using the same algorithm throughout the design solution). A more comprehensive discussion of design schemas, methods, notations, and heuristics is given in Guindon (in press).

The next category in Figure 9, a balanced development by meeting listed requirements, refers to strategy Designer 2 exhibited: He systematically examined all parts of his solution to verify that they satisfied the requirements. In the process he developed additional parts of his solution. The unbalanced solution development refers to cases where the designers did not know how to proceed, consulted the requirements for ideas, and immediately developed a partial solution for the examined requirement.

Frequently the design development was interrupted by, for example, the inference of new requirements or the immediate recognition of a solution in another part of the problem. At other times, the development of a solution was postponed because of lack of necessary information. As a consequence, designers frequently had to return to interrupted or postponed solution development activities. Such interruptions and returns contributed to deviations from a systematic and balanced solution development and therefore are reported separately. The returns to interrupted or postponed solutions were categorized as balanced or unbalanced as a function of the status of the solutions to which they returned. For example, if a designer returned to a solution so that he could continue to follow a design method, then the return was considered balanced. On the other hand, returning to complete a solution that immediately handled a new requirement was considered unbalanced. Designers also exhibited a variety of returns to interrupted partial solutions at a higher level of abstraction. These returns were typically recoveries from a drift that had led the designer to develop partial solutions at a very low level of detail.

Figure 10 shows the solution evaluation and debugging activities categorized as balanced and unbalanced. Solution evaluations using test cases were performed systematically to ensure correctness and completeness of the solution. Therefore, such episodes were considered balanced. Moreover,

Figure 10. **Percentages of balanced and unbalanced design activities for solution evaluation and debugging.**

Design Activities	Balanced	Unbalanced
Evaluate solution by		
Simulations with test cases	30	
Comparing to requirements in final review	14	
Comparing against given/inferred requirements	4	6
Lift domain scenarios	1	6
Identify or solve bugs	18	21
Total	67	33

Designer 2 exhibited a systematic review strategy at the end of the session—He reviewed each of the requirements on the problem sheet to ensure that his solution satisfied them. The other episodes were considered balanced or unbalanced as a function of the status of the solution that was evaluated or debugged.

Turning to the requirements, 55% of these activities were related to their elaborations, and 45% were related to their understanding. Additions or inferences of new requirements were considered unbalanced because these occurred throughout the session while the design solution was developed, and thus they violated the waterfall model (see Figures 3 and 4). Therefore, the design problem goals could be modified while the design solution was developed. Of all requirements elaborations, 33% were triggered by Lift scenarios and 67% by solution developments and simulations. The understanding of the requirements was always considered balanced.

To summarize, the results presented in Figures 3 to 10 show that the designers frequently deviate from a top-down approach. These results cannot not be accounted for by a model of the design process where problem specification and understanding precedes solution development and where the design solution is elaborated at successively greater levels of detail in a top-down manner.

In fact, the design behaviors observed in this study bear great resemblance to the way in which goals were satisfied in an errand-planning task described by Hayes-Roth and Hayes-Roth (1979). They observed that their subjects mixed decisions at various levels of abstraction. For example, the subjects planned low-level (detailed) sequences of errands in the absence of or in violation of a higher level plan. In the current study, the designers could elaborate partial solutions at arbitrary levels of abstraction in the solution decomposition prior to higher level decomposition. Behaviorally, the observed design process seems best characterized as *opportunistic,* to borrow Hayes-Roth and Hayes-Roth's term. The term *opportunistic* is restricted here to describe the

observed behaviors and does not, at this point, entail a particular cognitive model.

The next section presents examples of opportunistic design behaviors and discusses the cognitive mechanisms underlying them. This results in a definition of opportunistic design presented in Section 4.2.

3.3. Causes of Opportunistic Design Decomposition

Design tasks are considered to be the domain of bounded rationality. Newell and Card (1985) argued that the explanation of bounded rationality phenomena should be found in the interplay of basic psychological mechanisms and the human's intendedly rational endeavors. Basic psychological mechanisms that seem to play a role in opportunistic design include the role of data-driven processing by experts, the associative nature of human memory and spreading activation, working memory limitations. An important intendedly rational behavior is the designer's deliberate management of time or other resources to be most effective (see Simon, 1969/1981, for a discussion of design as a resource-allocation problem).

The main causes for opportunistic solution development include immediate recognition of a partial solution in another part of the problem, immediate handling of inferred or added requirements, drifting through partial solutions, and interleaving of problem specification with solution development. Each of these is examined in detail for its genesis.

Sudden Discovery of Unbalanced Partial Solutions

Episodes classified as knowledge discovery were characterized by the sudden emergence of new knowledge, without apparent planning, which subsequently plays an important role in the solution attempt (as in Kant & Newell, 1984). The episodes in Figures 3 and 4 annotated with a light bulb are particularly striking examples. Knowledge discovery often led to the unplanned addition of new requirements or new partial solutions. In turn, knowledge discovery was frequently followed by unplanned but drastic changes in the design activities.

Figure 11 shows the main design activity transitions that led to knowledge discovery of partial solutions, with their percentages over all solution-development episodes. The left column indicates the design activity that triggered the discovery of the partial solution.

A first example illustrates the immediate recognition of a partial solution, reused from another part of the solution. It was triggered by a solution simulation. A designer had used an interrupt structure to track floor requests earlier in the session for a different subproblem. His current goal was to elaborate the control and status operations for each lift handler (corre-

Figure 11. **Main types of design activity transitions leading to discovery of partial solutions.**

Preceding Episode	→ Episode With Discovered Knowledge	Percentage of All Solution Development Activities
Simulation of partial solution	→ Immediate recognition of solution from another part of the problem	6
Given or inferred requirement	→ Recognition of low-level partial solution prior to solution decomposition	4
Lift domain scenario	→ Recognition of partial solution in another part of the problem	2

sponding to Events 5 and 6 in Figure 5). He was then immediately reminded of the interrupt structure he had used earlier in the session to track floor requests. He then shifted to a subproblem that could be solved by an interrupt structure in a different part of the problem (corresponding to Event 7 in Figure 5). The immediate recognition of the partial solution was caused by a semantic association based on an analogy between the control and status operations of each lift handler—tracking the states of each lift—and tracking floor requests. Having recognized a partial solution with very little effort, the designer elected to change his current goal and shifted to the subproblem resolved by this partial solution. Presumably, the designer found this to be a more effective course of action than making a note of the partial solution and handling it later. After all, by immediately developing the partial solution, the designer put additional constraints on the design solution with little effort and reduced the daunting size of design possibilities. Already developed partial solutions seem to be resources that can be easily retrieved and reused, enter the focus of attention through association, and modify the designer's plans.

We also observed cases where a designer was examining the external representation of the solution while performing a simulation and unexpectedly saw a bug in another part of the solution. The designer changed his goal and immediately fixed the newly discovered bug. Presumably, the designer decided that it was more effective to correct the bug immediately than to note it and handle it later. Thus, unplanned information from external representations can also enter the focus of attention and modify the designer's plans.

Other instances of opportunistic development occur when a designer immediately recognizes a low-level partial solution for a requirement prior to design decomposition. For instance, as soon as a designer realized that button pressing was asynchronous, the thought of an interrupt structure as a solution was immediately triggered. Moreover, in this case, the designer drifted

through a short chain of partial solutions—from the interrupt structure to a mechanism for detecting order of interrupts. That is, each partial solution triggered the next solution. This immediate recognition of a solution seemed stimulated by the activation of a knowledge rule derived from past experience. Presumably the designer felt that it was more effective to pursue this lead immediately than to note it and return to it later. Prior to discovering the solution decomposition, it is advantageous to pursue promising partial solutions in the hope that they will provide early insights on the solution decomposition, especially when the discovery of this promising partial solution requires little effort from the designer.

Kant and Newell (1984) reported cases where problem solving in the problem domain triggered the discovery of solution knowledge. The current study also observed such a phenomenon. For instance, a designer simulated a Lift scenario where the lift doors stayed open a fixed amount of time before closing. This immediately triggered the idea of a timer. The partial solution seemed triggered by the activation of a knowledge rule linking a particular device behavior (e.g., the fixed length of the opening of the doors) with a solution (e.g., a timer).

Newell (1969) and Nii (1986) discussed the possibility that human abilities to solve ill-structured problems, such as design, arise from the data-driven application of knowledge in the forms of empirical associations or rules derived from past experience. Moreover, expertise in many problem domains has been attributed to the development of such data-driven rules (Anderson, 1982; Larkin, 1981). The application of these data-driven rules is considered to be automatic and to impose little cognitive cost, in contrast to goal-directed behaviors (Anderson, 1983).

These are very critical observations and findings. Ill-structured problems, because of their ill-specified goals, prevent the determination of a single and stable high-level goal and of a corresponding initial hierarchical plan of actions to be executed throughout the design process. Ill-structured problems make a goal-directed, top-down process difficult. On the other hand, human expertise is associated with the application of data-driven rules. The interaction of the ill structuredness of a problem with data-driven processing by experts is likely to induce the recognition of partial solutions at various levels of abstraction prior to an overall solution decomposition. To summarize, the results presented in this section support the hypotheses of Newell and Nii linking data-driven processing and ill-structured problems. Information that becomes the focus of attention—partial solutions, problem domain scenarios, requirements, and external representations—can trigger knowledge rules. As these data-driven rules are applied, the problems become better structured. In fact, the data-driven recognition of partial solutions is advantageous. The designer increases the number of constraints on the solution and decreases the daunting size of the solution problem space at very little cognitive cost.

Figure 12. **Example of an inferred test case whose solution is immediately developed. Boldfaced capitalized material indicates the type of design activity; arrows indicate that one activity triggered the next one.**

{LIFT SCENARIO} I am going to imagine one elevator and a few scenarios. Say there is a request from Floor 2 to 4 . . .

→

{INFERRED TEST CASE} What if you press up at the floor but, once in the lift, you press a down button? . . .

→

{NEW PARTIAL SOLUTION} So there is definitively the need for a queue of lift requests for each lift, separate from the floor requests. . . . Maybe the floor requests could be handled by a completely separate system from the lift requests.

The behaviors of novices may resemble, on the surface, those of experts when solving ill-structured problems. But it is only a surface resemblance. The high frequency of data-driven application of knowledge rules distinguishes the design process of experts from those of novices. Experts have sufficiently rich knowledge so that the application of data-driven rules imposes enough structure on the problem that it can be solved. The varied and rich sources of knowledge used by designers in this study are described in Guindon (in press). Novices are expected to deviate frequently from a top-down process. Their deviations are not caused, however, by the application of data-driven knowledge rules but by an inability to structure the problem and develop the solution due to lack of relevant knowledge.

Immediate Solution Development for New Requirements

As mentioned earlier, the main triggers for requirement elaborations are the development of the solution itself and the simulation of scenarios in the problem domain. In fact, 60% of all new requirements inferred during solution development have their corresponding solution immediately developed by the designer. This induced sudden shifts to other parts of the solution decomposition.

Figure 12 shows how a Lift scenario triggered the inference of a new requirement, which is handled immediately. Note that it was not the goal of the designer to infer a new requirement or a test case when performing the Lift scenario. The designer decided to change his goal and immediately handle this new requirement by shifting to another part of the solution decomposition. Immediately handling this new requirement benefited the designer. He obtained a critical insight on the overall solution decomposition—The floor requests could be handled by an independent system from the lift requests. This is an advantageous strategy: Because the design goal

Figure 13. **Lift scenario triggering an inferred requirement triggering an unbalanced partial solution by fulfilling a previously postponed goal.**

{LIFT SCENARIO} Let us say this is the fourth floor and this is the third floor. The lift on the fourth floor is requesting Floor 1. The lift on the third floor is requesting Floor 2. Well it says in the requirements, "You must service these requests eventually with floors being serviced sequentially in the direction of travel."

→

{INFERRED REQUIREMENT} So that means the floor from which the request is originating . . . not the destination.

→

{FULFILL PREVIOUS GOAL TO DEVELOP MODEL OF SYSTEM} That insight gives me an idea that the lift requests and floor requests might have more than one piece of information. A lift request is of the form originating floor and destination floor. What about floor requests? . . . A lift request is (lift#, orig. floor#, dest. floor#). . . .

---

changes as a result of inferring new requirements, the plan to reach this goal should be modified as soon as possible to allocate resources most effectively.

Figure 13 shows an example of an inferred requirement that satisfies a solution development previously postponed. Early in the session, the designer wanted to develop a model of the state of the system. He postponed this goal due to insufficient information and moved on to satisfy another goal. As illustrated in Figure 13, much later in the design session, a Lift scenario triggered the inference of the missing information and the postponed goal was reinstated, leading to a shift to another part of the solution decomposition.

The following two examples are interesting because they show how a similar inference of a requirement was reached by two designers following quite different reasoning paths. The examples also illustrate the types of psychological mechanisms underlying these inferences.

Figure 14 illustrates that while the designer simulated his solution—low-cost links to its two nearest neighbors (lifts)—the concept of geographical separation between lifts was activated. Whether the lifts were geographically near or far from each other was not stated in the problem statement. The genesis of this requirement—the lift cages might be far from each other—can be explained by the associative nature of human memory. The concept *near* activates the concept *far,* bringing it in working memory (e.g., see Anderson, 1983). Unplanned information can enter the designer's focus of attention through spreading activation. The designer decided to assume that the lift cages were side by side. In other words, he inferred that there were no special interactions between lifts and floors. Thus, the unplanned acquisition of new requirements, triggered associatively in the designer's world knowledge, may modify the course of the solution decomposition.

Figure 15 illustrates that the adoption of a problem-solving heuristic (i.e.,

Figure 14. **Inferred constraint triggered by a solution simulation through semantic association.**

{SOLUTION SIMULATION} . . . we have a communication system where any processor was connected to its two nearest neighbors.

→

{INFERRED CONSTRAINT} Unless these elevators are geographically separated, then that is another complete independent problem, where you have some interaction on some floors but not on other floors. . . . I will add to the requirements that the elevators are side by side and that they do not service the different floors differently.

Figure 15. **Inferred constraint triggered by a solution development guided by a general problem-solving heuristic.**

{PROBLEM-SOLVING HEURISTIC} The fact that I have N lifts makes it complicated. I will start by considering the case of one lift.

→

{INFERRED CONSTRAINT} But first, I have to make sure that there is no special interaction between lifts and floors. Yes, it is okay. I just have a bank of lifts and a bank of floor requests.

"Consider a simpler problem") can lead to the same inferred requirement as described in Figure 14. The designer wanted to make sure that by considering the problem of only one lift he did not miss a critical property of the problem that would compromise his overall design decomposition. In this case, the property was a special interaction between lifts and floors. Therefore, the unplanned acquisition of new requirements, inferred as part of ensuring a prerequisite condition for a problem-solving or design heuristic, may modify the course of the solution decomposition.

To summarize, inferences and additions of new requirements occur throughout the design solution development and are triggered from many sources. Sources observed in this study include associations between related concepts, external diagrammatic representations, and prerequisites for the application of a design process strategy. But, more important, designers tend to develop immediately the partial solution corresponding to the inferred constraint, leading to a change in goal and to a shift to another part of the solution decomposition. Until a designer has discovered the design solution decomposition, it is advantageous to evaluate immediately the impact of a new inferred constraint on the solution rather than take note of it and handle it later. The inferred requirement and its corresponding partial solution were often critical in discovering the proper design solution decomposition and in reducing the space of design possibilities.

Drifting

Designers frequently developed a sequence of associated partial solutions in violation of balanced development. Whereas the partial solutions were semantically related, they each could resolve subproblems in different parts of the solution decomposition. The associations could be based on components that interacted or interfaced with each other, on components that accomplished similar or opposite functions, on components that shared data, and so on. For example, defining the data structure to store the floor requests triggered defining the data structure for the emergency button, which itself triggered defining the outputs for input interrupts. Other drifts followed a chain of data-driven solutions derived from prior designs. Designers find it advantageous to follow a train of thought temporarily, thus arriving at partial solutions at little cognitive cost. In particular, before designers have established the overall design decomposition, these partial solutions may provide them critical insights on the proper way to decompose the problem and reduce the daunting size of design possibilites.

Solution Development by Problem Domain Scenarios

The uses of Lift scenarios during solution development are interesting because they often triggered the recognition of unbalanced partial solutions or of new requirements. The first case is similar to what Jeffries et al. (1981) called problem solving by understanding: The designer is temporarily unable to develop the solution and simulate scenarios in the problem domain to get new ideas. Participants in this study seldom produced Lift scenarios for this purpose. On the other hand, novices would be expected to do the opposite. The second case is to confirm the correctness of a discovered partial solution, in terms of a plausible Lift scenario, when the problem specification does not describe the corresponding lift behavior. This is related to the observation by Kant and Newell (1984) that designers use knowledge from the problem domain to compare the results of their solution with the (sometimes implicit) goals the solution should fulfill. The third case is to confirm the relevance of an inferred requirement and to get ideas about how to handle it, because this information is not included in the problem specification.

3.4. Differences Between Designers

Early in the session, Designer 2 retrieved highly integrated knowledge corresponding to a very high-level solution decomposition for the Lift system. As evidenced in the prompted review, this knowledge was abstracted from his previous designs of taxi cab dispatcher and film controller systems. This integrated knowledge, called a *specialized design schema,* provided the designer with a solution decomposition in terms of three subsystems: one for control

between lift processors, one for communication between processors, and one for scheduling. Following a specialized design schema does not necessarily imply a single order in which to develop these subsystems because each of these subsystems will tend to be relatively independent due to modularity. But because the schema provides a plan that each of these subsystems be developed, design activities that could be accounted for by the application of a specialized design schema were considered balanced.

In fact, the retrieval of the specialized design schema seems to underlie the greatest difference between Designer 1 and 2. Designer 1 had a greater percentage of solution development activities that resembled a top-down decomposition (40%) than did Designer 2 (14%). This difference is consistent with the visual impression of a better fit to a top-down, breadth-first development by Designer 2 than by Designer 1, when comparing Figures 3 and 4. Because Designer 2 had already designed systems with high-level decompositions similar to those of the Lift system, he could, early in the session, retrieve a high-level solution decomposition, which he could then follow in a top-down manner. In terms of Simon's (1973) analysis, the knowledge the designer had about similar or related problems readily imposed structure to the problem, facilitating the application of a top-down approach. Nevertheless, frequent and varied deviations from a systematic and balanced process were observed in both designers.

4. DISCUSSION

Other studies have observed sporadic deviations from a top-down process but described the design process of their experts as balanced and systematic or as following a top-down approach preceded by problem specification (Adelson & Soloway, 1984, 1985; Jeffries et al., 1981). This study shows that the early stages of the design process are best characterized as opportunistic, interspersed with top-down decomposition. Resolutions for these differences are presented.

4.1. The Impact of Structuredness

A topic of a science of design proposed by Simon (1973) is the structure of complex artifacts and their impact on the design process. Design problems vary in level of structuredness (Simon, 1973). The problems used by Jeffries et al. (1981) and Adelson and Soloway (1985) were probably more structured than the Lift problem in the sense of presenting less novelty to the designers. For instance, Jeffries et al. (1981) described their design problem as straightforward because it only required upper undergraduate level computer science knowledge. Adelson and Soloway (1985) gave the design of a relatively

simple electronic mail system to thoroughly trained communication system specialists. However, one of their expert designers, who worked on a familiar interrupt handler but with an unfamiliar chip, exhibited frequent deviations from a balanced development when dealing with the chip. The Lift problem required advanced computer science knowledge and, although our designers had developed related systems, the Lift problem presented novelty. Supporting our argument, Designer 2, who had developed systems more similar to the Lift system than had Designer 1, exhibited a design process that matched more closely a top-down approach than did Designer 1. Thus, a first conclusion is that design problems that are simple or that present little novelty can be solved by and large in a top-down manner. The designer can rapidly retrieve or discover the proper design decomposition on the basis of a specialized design schema; the designer already knows the answer. This design decomposition into major systems and subsystems can then be used to support the rest of the solution decomposition. The designer is now in a position to apply a top-down approach in filling out solution details and in expanding the remaining parts of the solution. But prior to the discovery of the overall system decomposition, the designer does not have this knowledge and its associated memory representation to support the top-down approach. Thus, prior to discovering the overall system decomposition, the designer tends to follow promising partial solutions and immediately evaluate the impact of newly discovered requirements in the hope of discovering it.

Another feature affecting the level of structuredness of design problems is the degree of completeness of the problem specification. The problems given by Jeffries et al. (1981) and Kant and Newell (1984) were more completely specified than are most high-level design tasks, including the Lift problem. As a consequence, the designers in our study needed to interleave problem specification with solution development more frequently than did the designers of the other studies. Therefore, the design goals of this study changed during solution development. The designers frequently elected to handle immediately the newly inferred requirements. The designers considered this design-process strategy to be effective: The new requirements might provide critical insights on the solution decomposition and require modifications to the planned solution. Hence, planning the solution decomposition needed to be on line: As new requirements were discovered, the designer needed to reevaluate his current plan and modify it accordingly. Finally, due to the associative nature of human memory and to data-driven processing by experts, scenarios in the Lift domain and inferred requirements often triggered the recognition of partial solutions in arbitrary points in the solution decomposition. Simon (1973) argued that much of the effort in solving a problem is actually in structuring the problem. This study shows that the process of structuring a design problem involves inherent deviations from a top-down approach.

The Lift problem also appears to require the integration of more sources of knowledge than did the problems used in other studies. The Lift problem required the integration of knowledge from reactive, embedded, and concurrent systems. The Lift problem also required scenarios of uses from users, but so did the electronic mail system problem given by Adelson and Soloway (1985). Nevertheless, one can expect the Lift problem to involve more scenarios from the problem domain than would, for example, the book indexing or the interrupt handler problems. These scenarios often triggered the recognition of partial solutions in arbitrary points in the solution decomposition.

The Lift problem, however, is not an unusual problem. In the debriefing session, the designers mentioned that it is similar in the incompleteness of its requirements to problems they had to solve in the field. Moreover, it shares many design issues with user interfaces and other frequent applications. The Lift problem involves concurrency—an unusual feature in design problems studied so far—and both designers knew solutions to handle concurrency. Therefore, the frequent deviations from a balanced design process are not special cases due to idiosyncracies of the Lift problem, noise, uninteresting performance breakdowns, or manifestations of bad design practices. Instead they are an inherent and important aspect of the design process. They are a natural consequence of solving design problems—ill-structured problems with incomplete specification, often presenting novelty to the designer, and requiring the integration of multiple sources of knowledge.

4.2. Behaviorally, Design Decomposition Is Opportunistic

Another topic in a science of design proposed by Simon (1973) is the control of the selection and ordering of actions during design. This study shows that the early stages of the design process are best characterized as opportunistic, interspersed with top-down decomposition. In terms of its behavioral manifestations, opportunistic design is design in which interim decisions can lead to subsequent decisions at various levels of abstraction in the solution decomposition. A decision at a given level of abstraction may influence subsequent decisions at higher or lower levels of abstraction, specifying actions to be taken at different times during the process.

Opportunistic design is frequently caused by the application of data-driven rules, leading to the automatic recognition of partial solutions in various parts of the decomposition. The data for these rules originate from solution development activities, problem-domain scenarios, given or inferred requirements, and external representations. Data-driven processing, characteristic of expert behaviors, is not goal directed. However, the application of these data-driven rules helps structure the problem so that it can be solved. As a consequence, designers immediately take advantage of the discovery of

partial solutions by elaborating them sufficiently to put constraints on the space of design possibilities. Novices, lacking specialized knowledge, may be unable to structure the problem enough to solve it.

Opportunistic design is also caused by the inferences and additions of new goals that reduce the incompleteness and ambiguity of the problem specification. These inferences and additions are triggered by many sources of knowledge: other requirements, partial solutions, and problem-domain scenarios. Their genesis is also varied: through semantic associations, through external representations changing the focus of attention, as prerequisites for the application of design strategies, and so on. But, more important, designers tend immediately to develop the partial solutions for inferred or added requirements. Opportunistic design is characterized by on-line changes in high-level goals and plans as a result of inferences and additions of new requirements. In particular, designers try to make the most effective use of newly inferred requirements, or the sudden discovery of partial solutions, and modify their goals and plans accordingly.

Opportunistic planning has been modeled in computer systems with a blackboard architecture. In typical blackboard systems, the knowledge sources or specialists are at different levels of abstraction. For example, one level of specialists could deal with problems at the level of the design process—how much time to allocate to each activity or when to shift to another activity. Another level of specialists could deal with software system issues—system decomposition, control, and communication between processors. Another level could deal with scheduling issues, another level with hardware interactions, and so on. Each knowledge specialist is self-activating and can be modeled as an "if-then" rule: If there is information on the blackboard that is relevant to the specialist, by matching the "if" part of the rule, the specialist activates itself. At any point in time, one or more knowledge specialists may be contextually relevant and invoke themselves. A control process makes cyclical decisions about which of the relevant knowledge specialists is most opportune to execute, that is, which activity to perform next. The locus of control can be in a separate executive process, distributed in the knowledge sources, on the blackboard, or in a combination of the three. Opportunistic planning combines backward reasoning (inference steps are applied from the desired goal to the current state) and forward reasoning (inference steps are applied from the current state toward the goal) in what appears to be the most advantageous way.

Opportunistic planning is in fact a more general type of planning than hierarchical planning. A blackboard-based model of opportunistic planning could account, parsimoniously, for both opportunistic and systematic design behaviors observed in this study. The on-line planning capability could account for the designers' changes in high-level goals and plans as they inferred new requirements or discovered new partial solutions. The applica-

tion of forward, data-driven rules, with knowledge sources organized at various levels of abstraction, could account for the immediate recognition of solutions in arbitrary points in the design decomposition. High-level knowledge sources, such as design schema and design methods, could account for the partially systematic aspect of the design process. Additions to blackboard systems may be needed to account for other observations in this study. For instance, we have observed inferences triggered by such a mechanism as spreading activation based on semantic associations.

Opportunistic design behaviors, however, do not necessarily imply an opportunistic model of planning. In his book on the architecture of human cognition, Anderson (1983) argued that behaviors that appear to violate hierarchical planning may actually be due to simple failures of working memory. Anderson commented, "Subjects may pursue details of a current plan that is inconsistent with their higher goals, simply because they have misremembered the higher goals" (p. 130). Anderson mentioned that opportunistic behaviors may also occur through data-driven productions, which may trigger radical shifts in the current goal. Finally, he introduced the concept of intentions. Whenever a behavioral sequence under the control of a goal structure encounters insufficient information to achieve the next goal, the action is postponed and an intention is set to deal with the goal later. These intentions are components of the condition part of new productions, which act as data-driven "demons" to be executed when the missing information becomes available. Anderson acknowledged that such productions with intentions are triggered only if they are remembered by the subject. So, according to Anderson (1983), opportunistic behaviors can also be accounted for in ACT*'s with its hierarchical goal structure; flat, data-driven structure; the forgetting of higher level goals; and the execution of data-driven productions with intentions.

Anderson's points are important. Indeed, we have observed the applications of data-driven knowledge rules when designers recognized partial solutions triggered by requirements or by Lift scenarios. We also observed how inferred requirements that satisfied a postponed goal tended immediately to reinstate that goal. But these could also be accounted for by an opportunistic model of planning. Moreover, designers could pursue details of a current plan that is inconsistent with a higher level goal simply because this goal is no longer relevant due to the inference or addition of new requirements. Because design problems have ill-defined goals and evaluation criteria, one must allow for changes in goals and plans during design. Consequently, planning needs to be on line as higher level goals change.

We also observed performance breakdowns that could be attributed to working-memory limitations.[2] One performance breakdown was the failure

[2] The idea of breakdowns presented here is more restricted than the one presented in Guindon,

to integrate known and understood constraints in the design solution. Another breakdown was the difficulty in performing mental simulations of solutions or of problem-domain scenarios. For instance, designers sometimes confused or forgot the function associated with different solution parts, or, more precisely, associated with the label given to the solution part. Moreover, designers found it difficult to simulate the interactions between components of the system, the behavior of a component if it extended over many steps, or the behavior of a subsystem calling centrally embedded subsystems. To help mental simulations, designers often resorted to diagrams. However, because diagrams were a poor medium to represent changes in location and time, they were not sufficient to prevent all simulation breakdowns. However, the performance breakdowns had relatively little impact on a balanced design process. Designer 1 experienced a total of four performance breakdowns; Designer 2 experienced a total of three breakdowns. In all cases, the designers rapidly recovered from the breakdowns, within two to four episodes, to resume their design process as they wished. Hence, one cannot attribute to performance breakdowns the majority of the observed deviations from a balanced solution development.

It would be tempting to dismiss the selection of a psychological model as an irrelevant issue: The two models can make behaviorally equivalent predictions, and considerations such as parsimony and elegance are not clear cut. Unfortunately, both models can be interpreted as making different sets of claims about the features of a computational environment to support software designers. I speculate that Anderson would insist on an environment that supports a top-down design process with a hierarchical goal structure. Hayes-Roth and Hayes-Roth would insist on an environment that supports flexible and easily reorganizable goal structures and on-line planning, as unplanned relevant information may enter the focus of attention and modify the problem structure throughout the design process.

Ultimately, it might be very difficult to demonstrate empirically the validity of one psychological model against another for tasks as complex as design. Design behaviors are probably influenced by many interacting complex factors—structuredness and types of problems, degree of expertise of the designer, the amount and type of relevant specialized knowledge and heuristics the designer bring to bear, basic psychological mechanisms, the designer's goals and preferences, and so on. Finally, alternative models that are complex enough to account for design may have enough degrees of freedom to produce behaviorally equivalent predictions. Nevertheless, further descriptive and experimental studies of the design process should be performed to assess the impact of the structuredness of the design problem. They

Krasner, and Curtis (1987). It is restricted to difficulties in the execution of intended design activities.

should also investigate how designers plan and control the allocation of their time and other resources and how this interacts with basic psychological mechanisms. Such studies should also include computer models of aspects of the design process to establish the sufficiency of the proposed model.

A complementary strategy is to develop alternative design methods and computational environments that embody the competing implications from these different models. One can then evaluate which of these methods and environments actually better supports the design process. For example, one might provide a group of experts an environment that enforces a top-down approach and another group an environment that enforces an opportunistic approach. One could then compare the effectiveness of each environment and associated method in supporting the early stages of design by examining the number and types of errors, the time to produce an initial design, the quality of the design, and so on. If one method or environment were to support the design process better than the others, this finding would constitute indirect evidence for its corresponding model. However, one must realize that by doing so one is as much studying the design process as one is shaping what the process is—The design process is not a natural phenomenon but a human artifact (Simon, 1969/1981).

Problem-solving behaviors, which have been labeled opportunistic, have been observed in various areas of human activities. This observation suggests their ubiquity for a large class of problems beyond software design. Opportunistic activities have been observed in a 13-week field study of a team of programmers and designers by Visser (1987). She identified a number of causes for deviation from hierarchical plans, including economic use of available means, postponing a decision due to insufficient information, handling a solution component that is similar to the current one, and changing the decision criteria used. Unfortunately, she did not include a description of her data in the report. Ullman, Stauffer, and Dietterich (1987) in a 10-hr verbal protocol study of design in mechanical engineering, observed that expert designers progress from systematic to opportunistic behaviors as the design evolves. Schoenfeld (1985) provided some evidence for opportunistic problem solving in mathematical reasoning, whereas Flower and Hayes (1980) did so for document composition.

4.3. Implications for Training, Methods, and Environments

The results suggest that until the proper design decomposition is discovered, the design process should be opportunistic. One should immediately take advantage of any inferred information and the additional constraints it poses on the solution. Only after the proper decomposition is discovered can one apply the top-down approach. This interpretation agrees with Mills

(1986) and Fairley (1985), who argued that the benefit of the top-down approach can be obtained only after some bottom-up thinking, trial design and coding, and backtracking have been accomplished. This is a position compatible with Parnas and Clements's (1986) view of the design process that designers are expected to document the design process and artifacts as if they had been produced in a systematic fashion. Parnas and Clements argued, however, that this idealized systematic process can be achieved only in rare circumstances.

The results of this study also remind one of the techniques used by Polya (1957, 1962) in teaching mathematics. Of course, once one has discovered the solution to the mathematical riddle, one has to express proof using accepted mathematical methods and notations. But as Polya recognized, the process of discovering the solution is rife with trial and error, garden paths, partial solutions, and insights. Polya's methods encouraged the deliberate use of an opportunistic approach to mathematical problem solving.

This study concerns individuals' cognitive activities during high-level design. Although these implications are intended for supporting the early stages of software design, the observed opportunistic character of other tasks (e.g., mathematical problem solving, document composition, and mechanical design) suggests that these recommendations could apply for methods and environments for other tasks as well.

The observations of opportunistic design behaviors, the need to integrate multiple sources of knowledge, and the frequent absence of a predetermined solution path suggest the following implications for a computer environment:

1. The environment should not embody a method that locks designers into a strict order of activities. A strict order of activities may hinder the opportunistic insights critical in discovering the proper design decomposition.
2. The environment should support rapid access and shifts between tools to represent and manipulate different kinds of objects and the representations of these objects. Some of these objects are informal requirements, information about the problem domain, issues and criteria about the system and the design process, design decisions expressed in a formal or semiformal notation, and design process goal management.
3. It should support easy navigation between these objects, not imposing a predetermined order of activities, and still have the ability to support an agenda of activities by the designer.

The incompleteness and ambiguity of the problem specification and the observation of discovery of knowledge during design suggest the following implications:

1. The representation languages in the environment should support a smooth progression from requirements expressed informally, to design decisions expressed formally or semiformally, to code.
2. The environment should support easy editing and reorganization of the requirements, design issues, and design decisions as the incompleteness and ambiguity of the problem specification are reduced through the design process.
3. It should support the identification of the origin of the requirements—as explicitly given, as inferred constraints, and as added requirements.
4. It should support the representation of interim or partial design objects in varied parts of the design decomposition.

In summary, experienced system designers deviate from a strictly top-down approach in the early stages of design. This study provides evidence that opportunistic design is advantageous and manifests itself through many types of behaviors: (a) inferences of new requirements that changed the design goals, (b) the immediate development of partial solutions to these new requirements, (c) the immediate recognition of partial solutions in various parts of the solution decomposition, (d) drifting, and (e) solution insights triggered by scenarios in the Lift domain. These deviations from top-down design appear to be consequences of intrinsic features of design problems—incomplete specification of the problem, lack of a predetermined solution path, and integration of multiple sources of knowledge. Competing cognitive models based on hierarchical planning and opportunistic planning may be equally able to account for these observations, but these competing models have different implications for methods and environments that support the early stages of design. In any event, efforts to support design should reconcile themselves with the opportunistic behaviors witnessed here.

Acknowledgments. Joyce Conner performed a thorough and insightful analysis of the protocols and has contributed immensely in the production of every aspect of this article. Thanks to Herb Krasner for collecting the protocols. Jeff Conklin, David Bridgeland, Mitch Lubars, and Herb Krasner participated in the preliminary analyses of the protocols. I am grateful for excellent comments from these external reviewers: Stuart Card, Judith Olson, Peter Polson, and an anonymous reviewer. Glenn Bruns, Jeff Conklin, Bill Curtis, Jonathan Grudin, Frank Halasz, Clayton Lewis, Patrick Lincoln, and Mitch Lubars provided many useful comments on an earlier version of this article. Patrick Lincoln also provided special help in detailed reviews of the manuscript. Noreen Garrison and Nancy Gore provided excellent editing for syntax, style, clarity, and structure.

REFERENCES

Adelson, B., & Soloway, E. (1984). *A cognitive model of software design* (Tech. Rep. No. 342). New Haven, CT: Yale University, Department of Computer Science.

Adelson, B., & Soloway, E. (1985). The role of domain experience in software design. *IEEE Transactions on Software Engineering, 11,* 1351-1360.

Anderson, J. R. (1982). Acquisition of cognitive skill. *Psychological Review, 89,* 369-406.

Anderson, J. R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Carroll, J. M., & Rosson, M. B. (1985). Usability specifications as a tool in iterative development. In H. R. Hartson (Ed.), *Advances in human-computer interaction* (Vol. 1, pp. 1-28). Norwood, NJ: Ablex.

Carroll, J. M., Thomas, J. C., & Malhotra, A. (1979). Clinical-experimental analysis of design problem solving. *Design Studies, 1,* 84-92.

Dahl, O. J., Dijkstra, E. W., & Hoare, C. A. R. (1972). *Structured programming.* New York: Academic.

Dijkstra, E. W. (1976). *A discipline of programming.* Englewood Cliffs, NJ: Prentice-Hall.

Dunn, R. (1984). *Software defect removal.* New York: McGraw-Hill.

Fairley, R. E. (1985). *Software engineering concepts.* New York: McGraw-Hill.

Flower, L., & Hayes, J. R. (1980). The dynamics of composing: Making plans and juggling constraints. In G. Steinberg (Ed.), *Cognitive processes in writing* (pp. 31-51). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Guindon, R. (in press). What knowledge is exploited by experts during software system design. *International Journal of Man-Machine Studies.*

Guindon, R., & Curtis, B. (1988). Control of cognitive processes during software design: What tools would support software designers? *Proceedings of the CHI '88 Conference on Human Factors in Computing Systems,* 263-286. New York: ACM.

Guindon, R., Krasner, H., & Curtis, B. (1987). Breakdowns and processes during the early activities of software design by professionals. In G. Olson, E. Soloway, & S. Sheppard (Eds.), *Empirical studies of programmers, second workshop* (pp. 65-82). Norwood, NJ: Ablex.

Hayes-Roth, B., & Hayes-Roth, F. (1979). A cognitive model of planning. *Cognitive Science, 3,* 275-310.

Jackson, M. (1983). *System development.* Englewood Cliffs, NJ: Prentice-Hall.

Jeffries, R., Turner, A. A., Polson, P., & Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 255-283). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Kant, E., & Newell, A. (1984). Problem solving techniques for the design of algorithms. *Information Processing and Management, 28,* 97-118.

Larkin, J. H. (1981). Enriching formal knowledge: A model for learning to solve textbook problems. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition* (pp. 311-334). Hillsdale, NJ: Lawrence Erlbaum Associates, Inc.

Lewis, C. (1990). A research agenda for the nineties in human-computer interaction. *Human-Computer Interaction, 5,* 125-143.

Malhotra, A., Thomas, J. C., Carroll, J. M., & Miller, L. A. (1980). Cognitive processes in design. *International Journal of Man-Machine Studies, 12,* 119-140.

Meyer, B. (1985). On formalism in specifications. *IEEE Software, 2*(1), 6-26.

Mills, H. D. (1986). Structured programming: Retrospect and prospect. *IEEE Software, 3*(6), 58-66.

Newell, A. (1969). Heuristic programming: Ill-structured problems. In J. Aronofsky (Ed.), *Progress in operations research* (pp. 362-414). New York: Wiley.

Newell, A., & Card, S. (1985). The prospects for psychological science in human-computer interaction. *Human-Computer Interaction, 1,* 209-242.

Nii, H. P. (1986, August). Blackboard systems: Blackboard applications systems, blackboard systems from a knowledge engineering perspective. *AI Magazine,* pp. 82-106.

Parnas, D. L., & Clements, P. C. (1986). A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering, 12,* 251-257.

Polya, G. (1957). *How to solve it.* New York: Doubleday.

Polya, G. (1962). *Mathematical discovery. On understanding, learning, and teaching problem solving* (Vol. 1). New York: Wiley.

Reitman, W. R. (1965). *Cognition and thought.* New York: Wiley.

Rittel, H. (1972). On the planning crisis: Systems analysis of the first and second generations. *Beprifts Konomen, NR 8,* 390-396.

Royce, W. W. (1970). Managing the development of large software systems: Concepts and techniques. *Proceedings of the Ninth International Conference on Software Engineering,* 328-338. New York: ACM.

Sacerdoti, E. D. (1975). *A structure for plans and behavior* (Tech. Rep. No. 109). Menlo Park, CA: Stanford Research Institute.

Schoenfeld, A. H. (1985). *Mathematical problem solving.* New York: Academic.

Simon, H. A. (1973). The structure of ill structured problems. *Artificial Intelligence, 4,* 145-180.

Simon, H. A. (1981). *Sciences of the artificial* (2nd ed.). Cambridge, MA: MIT Press. (Original work published 1969)

Swartout, W., & Balzer, R. (1982). On the inevitable intertwining of specification and implementation. *Communications of the ACM, 25,* 438-440.

Ullman, D. G., Stauffer, L. A., & Dietterich, T. G. (1987, November-December). Toward expert CAD. *Computers in Mechanical Engineering,* pp. 56-70.

Visser, W. (1987, May). *Abandon d'un plan hiérachique dans une activité de conception* [Giving up a hierarchical plan in a design activity]. Cognitiva 87. Paris. (For an English version, see Tech. Rep. No. 814 from INRIA, 1988, Paris.)

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM, 14,* 221-227.

HCI Editorial Record. First manuscript received March 31, 1989. Revision received May 25, 1989. Final manuscript received September 6, 1989. Accepted by Peter Polson. — *Editor*
