Knowledge Management for Software Design

David F. Redmiles

David Hilbert, Jason Robbins, Michael Kantor, Shilpa Shukla

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425

email: redmiles@ics.uci.edu
voice: (714) 824-3823
fax: (714) 824-1715
http://www.ics.uci.edu/~redmiles/

Abstract

The poor design of interactive systems has been attributed to insufficient communication between developers and end users. Critical forms of communications include soliciting requirements and feedback on prototype systems. Our research develops a software lifecycle emphasizing feedback from end users and communication among end users, developers, and other stakeholders in a software development project. Communication implies the transferal, storage, and review of knowledge about a software project. Three software systems — Argo, EDEM, and the Knowledge Depot — illustrate one kind of support that can be provided for the management of knowledge to improve the design of interactive software. Specifically, Argo provides support for the design of systems; EDEM provides support for collecting feedback and usage data from end users; and the Knowledge Depot provides support for reviewing feedback and other information about a system.
Knowledge Management for Software Design

Introduction

For many years, the poor design of interactive systems has been attributed to insufficient communication between developers and end users (Gould, Lewis 85; Fischer 89a). The problem is compound. Usability practices from the field of human-computer interaction are not integrated into the theory and practice of software engineering. Therefore many software developers are unaware of what constitutes good communication between them, end-users, and other stakeholders. Conversely, software developers who attempt to learn and employ usability practices are often faced with high learning curves and expensive-to-implement methods (Olson, Moran 95). Our research has three objectives: 1) to develop a model of software development as a process of on-going communication; 2) to support this model through active mechanisms in software tools; and 3) improve the accessibility and acceptance of usability methods by software practitioners. In general, the objectives reflect a theory of human-centered software development.

The theory of human-centered software development is expressed as a lifecycle with three elements: design, usage, and review. The concept of knowledge management is implicit in the approach. Knowledge about what constitutes a good design is communicated to software developers. Knowledge about actual system usage is noted and recorded. Feedback from end users is sometimes solicited. System usage and other related design knowledge is discussed among developers. Three systems support these activities: Argo, EDEM, and Knowledge Depot. These systems are described below from both their theoretical contributions as well as their actual system features. An integrated scenario illustrates how the systems can work in concert to support the management of knowledge relevant to software design.

Not all details of the different systems can be presented below. Therefore, the reader is referred to the other references for details on Argo (Robbins, Hilbert, Redmiles 98), EDEM (Hilbert, Redmiles 98), and the Knowledge Depot (Kantor, Zimmermann, Redmiles 97). A Web guide concludes this paper for those interested in the most current information, on-line demonstrations, and downloading of software.

Conceptual Framework—Human-Centered Software Development

Figure 1 shows one way to illustrate a human-centered software development lifecycle. Namely, prototype systems are designed, evaluated in a usage situation, and the results of the evaluation are reviewed and incorporated back into a revised design. Of course, by this diagram, we do not mean to imply that these activities are segregated phases of a lifecycle; rather, the activities are separated out for ease of discussion and to emphasize areas where software tool support works well.
This lifecycle is human-centered in two ways. First, the emphasis on prototypes and evaluation of usage maintains a focus on the requirements of end users and other stakeholders in the project. Second, there is an underlying assumption that the software developers need cognitive support for managing the complexity of usability information and incorporating that data and other information into design. Thus, the applications being developed are engineered to human cognitive needs of the end users and the tools used to developed those applications are engineered to the human cognitive needs of the software developers.

The lifecycle illustrates three ways support for knowledge management is provided. A software design environment (Argo) provides software developers with knowledge about what constitutes a good design and supports the rapid development of prototype applications. A software substrate (EDEM) provides support for collecting knowledge about actual system usage. Knowledge about the actual usage as well as other knowledge related to the design of a system can be recorded and elaborated upon by a group memory (Knowledge Depot).

**Software Design Environments**

The approach to human-centered software development presented above emphasizes the use of software prototypes. This is well motivated by the research on participatory design (Greenbaum and Kyung 1991). Prototypes provide a focus around which software developers, end users, and other stakeholders can communicate and expose requirements. While we acknowledge the neces-
sity of non-executable prototypes (such as mock-ups or informal drawings) early in the development lifecycle, we seek to enable software developers to create executable prototypes as soon as possible. The approach we have taken is to focus on high-level design through software architecture and support that design through *domain-oriented design environments* (Fischer 1994). In particular, we have developed a design environment, called Argo, for the domain of software architecture design (Robbins, Hilbert, Redmiles 1998). In this domain, it is possible for a designer to generate a software prototype from a high-level architectural specification. Figure 2 provides a view of Argo (center) being used to design a video game (left). A “to do” checklist window appears on the right in this figure.

In general, a domain-oriented design environment is a tool that augments a human designer’s ability to design complex artifacts. Design environments must address systems-oriented issues such as design representation, transformations on those representations (e.g., code generation), and application of analysis algorithms. Furthermore, design environments exceed the capabilities of most CASE (computer-aided software engineering) tools by addressing the cognitive needs of designers. Specifically, we have examined three cognitive theories of design and problem solving: reflection-in-action, opportunistic design, and comprehension and problem solving. Figure 3 summarizes our work by showing a mapping from these cognitive theories to specific design environment features that address them.

Figure 2: The Argo Software Design Environment
The cognitive theory of reflection-in-action observes that designers of complex systems do not conceive a design fully-formed (Schoen, 1983, 1992). Instead, they must construct a partial design, evaluate, reflect on, and revise it, until they are ready to extend it further. Design environments support reflection-in-action with critics that give design feedback. Critics are agents that watch for specific conditions in the partial design as it is being constructed and present the designer with relevant feedback. Critics can be used to deliver knowledge to designers about the implications of, or alternatives to, a design decision. Examples of domain knowledge that can be delivered by critics include well-formedness of the design, hard constraints on the design, rules of thumb about what makes a good design, industry standards, organizational guidelines, and the opinions of fellow project stakeholders and domain experts. In Argo, critics are displayed in the “to do” checklist window on the right in Figure 2.

The cognitive theory of opportunistic design explains that although designers plan and describe their work in an ordered, hierarchical fashion, in actuality, they choose successive tasks based on the criteria of cognitive cost (Hayes-Roth, Hayes-Roth, 1979; Guindon, Krasner, Curtis 1987; Visser, 1990). Simply stated, designers do not follow even their own plans in order, but choose steps that are cognitively the least cost among alternatives. The cognitive cost of a task depends on the background knowledge of designers, accessibility of pertinent information, and complexity of the task. Thus, although it is customary to think of solutions to design problems in terms of a hierarchical plan since hierarchical decomposition is a common strategy to cope with complex design situations, in practice, designers have been observed to perform tasks in an opportunistic order. Design environments can allow the benefits of both an opportunistic and a prescribed design process. They should allow, and where possible augment, human designers’ abilities to choose the next design task to be performed. They can also provide information to lower the cost of following the prescribed process and avoid context switches that deviate from it.

The theory of comprehension and problem solving observes that designers must bridge a mental
gap between their model of the problem or situation and the formal model of a solution or system (Kintsch, Greeno, 1985; Fischer, 1987; Pennington 1987). Problem solving or design proceeds through successive refinements of the mapping between elements in the design situation and elements in the formal description. Multiple, overlapping design perspectives in design environments allow for improved comprehension and problem solving through the decomposition of complexity, the leveraging of the familiar to comprehend the unfamiliar, and the capture of multiple stakeholders’ interests. It is our contention that no fixed set of perspectives is appropriate for every possible design; instead perspective views should emphasize what is currently important in the project. When new issues arise in the design, it may be appropriate to use a new perspective on the design to address them.

Usage Data Collection

Usability breakdowns occur when developers’ expectations about system usage do not match users’ expectations. A usage expectation determines whether a given interaction (e.g., a sequence of mouse clicks) is expected or unexpected. For example, a developer may hold the usage expectation that users will fill in forms from top to bottom with minor variation, while a user may hold the expectation that independent sections may be filled out in any order.

Figure 4: Differing Expectations between Developers and Users

Figure 4 shows a conceptual picture of expectations held by two groups of stakeholders (developers and users) and expectations encoded in the system being used. Each lowercase “e” represents a tacit expectation held in the mind of a person or in the code of a program. Developers get their expectations from their knowledge of the requirements, past experience in developing systems, domain knowledge, and past experience in using applications themselves. Users get their expectations from domain knowledge, past experience using applications, and interactions with the system at hand. The software system embodies implicit assumptions about usage that are encoded in screen layout, key assignments, program structure, and user interface libraries. Each upper case “E” represents an explicit expectation. Several usability methods seek to make implicit expectations of developers and users explicit, e.g., cognitive walkthroughs (Wharton et al. 1994), participatory design (Greenbaum, Kyung 1991), and use cases (Rumbaugh, Booch, Jacobson 1997).
Once a mismatch between users’ and developers’ expectations is detected it can be corrected in one of two ways. Developers can change their expectations about usage to better match users' expectations, thus refining the system requirements and eventually making a more usable system. For example, features that were expected to be used rarely, but are used often in practice should be made easier to access. Alternatively, users can adjust their expectations to better match those of developers, thus learning how to use the existing system more effectively.

Detecting a breakdown or difference in expectations is useful, but aligning expectations requires knowledge of the other expectations and the specific differences. For developers to learn about users’ expectations they need specific details of actual usage, including context, history, timing, and intent. For users to learn of developers’ expectations they need clear documentation of the intended system operation and rationale to be presented to them at the time of breakdown. In either case, dialog between users and developers can help clarify and expose expectations.

**Information Review and Project Awareness**

In the CSCW literature, systems labeled as group awareness tools are generally designed to provide relatively instantaneous awareness of a group of individuals. This research explores a type of awareness tool that does not fit within this implicit classification of awareness. The approach instead focuses on providing individuals with awareness of multiple groups (rather than multiple individuals), over days or weeks (rather than fractions of seconds to minutes). The types of informational benefits derived from such a system vary greatly from the benefits of a typical group awareness tool. Table 1 illustrates the types of questions answered by different classes of awareness systems. The two dimensions are the frequency with which users receive information and the unit of observation.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Unit of Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Individual</td>
<td>Group</td>
</tr>
<tr>
<td>Seconds to Minutes</td>
<td>What is a person’s location and current activity? (example tools are Portholes, Piazza)</td>
</tr>
<tr>
<td>Days to Months</td>
<td>What is a person trying to accomplish this week? What are a person’s plans for this week? What problems is a person working on solving? (example tools are various calendar applications and distribution lists).</td>
</tr>
</tbody>
</table>
A tool whose unit of observation is an individual, helps users maintain awareness of a set of individuals; example tools are Portholes (Dourish & Bly, 1992; Mantei, Baecker & Sellen, 1991; Lee, Girgensohn, Schlueter 1997) and Piazza (Isaacs, Tang & Morris, 1996). A tool whose unit of observation is a group helps users maintain awareness of a set of groups; example tools are Video Windows (Abel, 1990) and wOrlds (Boggy & Kaplan, 1995). Information that users need to be aware of over days rather than over minutes tends to be much more conceptual and task oriented than physical. Instead of providing awareness through graphical or audio means of some physical fact (the user is in room X, sitting, talking on the phone and browsing the web), we instead have to provide information that is much more abstract (a person plans to work on the following tasks, and the tasks have this set of priorities, and the following problems are delaying them) which type of information is best represented textually, as is done in calendars. Whereas the unit of information in a typical awareness system is an individual, the unit of awareness in a system aimed at groups over days is the task or set of tasks of the group. Awareness of the task then is what makes this type of awareness so important.

We have focused on an approach being developed to fill the Group/Days-to-Months quadrant of the diagram. We refer to tools that fit the this quadrant as project awareness tools. We have developed one such tool, called Knowledge Depot, which we illustrate below.

Project Awareness, as with other forms of awareness, is about retaining an awareness of some unit of observation (in this case the various tasks or groups within a project) in order to help people to better coordinate their work. Simple examples of this are two software development groups within a project: groups A and B, and group A depends upon a component produced by group B. Workers in group A need to remain aware of any changes B makes in the interface to their component, so that they can account for those changes in their own work. Better yet would be to enable group A to be aware of the possibility of change if group B starts discussing plans to change their interface. Awareness between groups allows them to coordinate schedules and designs. A network of awareness between each group and all of the diverse kinds of groups that can affect it can help an entire project to coordinate its work.

The capabilities needed to provide this kind of awareness can often be found or added to a group memory. Group Memories are used to store and organize project related information. Project awareness can be achieved by keeping people aware of what kinds of information has been put in the group memory. As the information is added, the people who are affected by that information are informed that it has been added, allowing them to remain aware of any changes in design or schedule that could affect them. This is the approach used by Knowledge Depot.

**System Implementation Progress**

The human-centered software development lifecycle presented above is being explored through three systems. The Argo Software Design Environment implements the design theory component illustrated in Figure 1. The EDEM (Expectation-Driven Event Monitoring) system substrate implements the observation of usage component. The Knowledge Depot implements the review component. The preceding sections introduced our theoretical understanding of a human-centered software development software lifecycle and each of the component activities. The following sections illustrate how the components could work together through a hypothesized scenario and then
explain more about each system’s details. As noted earlier, additional details can be found elsewhere (Robbins, Hilbert, Redmiles 1998; Hilbert, Redmiles 1998; Kantor, Zimmermann, Redmiles 1997).

System Integration — A Scenario of Design, Usage, and Review

We are working on integrating our tools for design, observation, and review (project awareness). Our objective is to provide integrated support of the human-centered software development life cycle described earlier and shown in Figure 1. We present our integration effort by means of a scenario in which a developer uses Argo, EDEM, and Knowledge Depot in developing a simple Travel Expense Form application written in Java, monitor its usage, and review the data from the monitoring.

The interface shown in Figure 5 represents applications used to report travel expenses for reimbursement purposes. The layout of the interface is based on certain tacit, cognitive assumptions about the tasks and the users of the form. For example, the grouping and order of fields reflects developers’ expectations about how users mentally organize the requested data and the sequence in which it is most convenient or efficient to provide it.

Figure 5: Travel Voucher Application
In order to test some of these assumptions, a developer uses the EDEM Agent Editor to specify an Expectation Agent that will trigger whenever a user begins editing the “Address Section” of the Travel Expense Form, see Figure 6. The developer uses the Agent Editor to express interest in detecting when the user begins to edit the “ZIP” text field. The developer then adds this simple event specification to an Expectation Agent that will trigger whenever a user begins editing the “Address,” “City,” “State,” or “ZIP” fields in the Travel Expense Form.

The agent specification consists of: (1) a name, (2) an activate pattern (an event pattern which, if satisfied, causes the agent to be activated), (3) a cancel pattern (an event pattern which, if satisfied, causes the agent to cancel evaluation of its activate pattern, (4) timing constraints, (5) a condition to be checked once the activate pattern has been satisfied, (6) an action to be performed if the agent is activated while the condition is TRUE, (7) and a flag indicating whether the agent should continue checking after successfully triggering once.

Figure 7 presents a display of the state of the event monitors when a user has edited the “Traveler” field and then shifted input focus to the “Address” field. This display is primarily for our development and debugging. Note that the LOST_EDIT event is not generated for the “Traveler” field until the user actually begins editing another field in the form. This is an example of how EDEM provides events at a higher level of abstraction than the window system events generated by Java. In this fashion, an agent centered on detecting when editing has been completed in a particular field does not need to be concerned with monitoring every other field in the interface.
Figure 7: Monitoring Events

Figure 8: Organizing Information for Review in Knowledge Depot
The agent defined in this example generates a single event (“Address Started”) as soon as any of the fields in the “Address Section” has been edited. This agent can then be used in conjunction with other agents that detect when other sections have been started and/or finished. The developer can thus compare the actual order that users complete the sections and fields in the Travel Expense Form with the order that was originally expected.

Feedback from agents must be reviewed and analyzed to inform future design decisions. To this end, we have integrated EDEM with the Knowledge Depot which collects and organizes Expectation Agent reports for later review. Figure 8 shows several categories of data collected from Expectation Agents monitoring the Travel Expense form. These categories can then be used to link back into the appropriate (of affected) architectural elements in the Argo design of the system (Figure 9), thus providing an indication of where changes to the system will need to occur.

Figure 9: Architecture of the Travel Voucher Application in Argo

![Architecture of the Travel Voucher Application in Argo](image)

Figure 10: Process Model for Constructing the Architecture of an Application

![Process Model for Constructing the Architecture of an Application](image)
A software developer can use the Argo design environment to specify the components of the Travel Voucher Application. In future work, we seek to make the feedback from agents link directly to these design components.

**The Argo Software Design Environment**

Argo is our software architecture design environment illustrated in Figures 1, 9, and 10. It is based on the cognitive theories of reflection-in-action, opportunistic design, and comprehension and problem solving (see the earlier section, “Software Design Environments” on page 3). Argo offers basic CASE tool features for entering an architectural model and applying code generating transformations. Furthermore, Argo contains critics that automatically provide design feedback, a user interface for browsing design feedback, a process model to guide the architect and help control critics, and multiple coordinated design perspectives.

Our current Argo implementation supports critics that run in a background thread of control to continuously analyze a software architecture as it is being manipulated. Critics are made up of an analysis predicate, a design feedback item, and various attributes used to determine if the critic is timely and relevant. Criticism control mechanisms are objects that define Argo’s policies on when to apply individual critics to a given part of the architecture. For example, one criticism control mechanism selects critics that are relevant to stated design goals, while another selects critics that are timely (relevant to design decisions that the architect is currently considering). Argo associates critics with specific types of design materials (i.e., elements of the architect model) rather than storing all critics in a central knowledge-base. This allows us to keep Argo’s kernel simple and flexible, and will soon allow us to load models and their associated critics over the Internet as needed.

Argo presents design feedback to the architect through a “to do list” metaphor. Items on the “to do list” indicate an issue that the architect must consider before the design is finalized. Critics unobtrusively add and remove items as design problems arise and are resolved. When the architect selects an item an explanation of the issue and possible resolutions is displayed and the “offending” elements of the architecture are highlighted to help build the mental context needed to resolve the issue. Furthermore, the architect can send email to the expert who authored the critic to open a dialog and draw the architect into the organizational context needed to resolve some issues. In future work we will address recording these design issues and resolutions as a form of design rationale.

Perspectives are implemented as predicates that select a part of the architecture model to be displayed in a given context. For example, one perspective may show software components and their communication relationships, while another perspective may show the relationship between components and operating system resources.

The Argo process model consists of a set of design materials for representing design tasks and dependencies between them. Since the process modeling domain is built on top of the Argo kernel (just as is the architecture modeling domain), we can apply Argo’s own facilities to editing, visualizing, and critiquing the design process. Each step in the design process indicates a set of design decisions that the architect will consider during that step. As the architect progresses through the
process model, process status information is used to infer which critics are timely.

Critics, criticism control mechanism, and perspectives are each implemented as Java boolean functions. This allows great freedom in expressing the analyses to be performed and permits us to use standard Java development tools. In the future, we hope to integrate an expert system or other technologies that focus on the kinds of predicates that are most often needed. Support for end user customizability is also planned.

**EDEM Software for Usage Data Collection**

We have developed prototype software for detecting and resolving mismatches in usage expectations by allowing developers to specify their expectations in terms of user interface events. These expectations are encoded in the form of expectation agents that continually monitor usage of the application and perform various actions when encapsulated expectations are violated.

In order to support this functionality, expectation agents need access to events occurring in an application’s user interface. However, in order for expectations to be expressed at an appropriate level of abstraction, they need access to higher level events than are produced by the window system. As a result, expectation agents are implemented on top of an expectation-driven event monitoring substrate (EDEM) that provides a multi-level event model. Aspects of the EDEM software are illustrated in Figures 6 and 7.

Other researchers have proposed event monitoring as a means for collecting usability data, however, existing approaches suffer from one or more of the following limitations: (1) low-level semantics—events are captured and analyzed at the window system level, or just slightly above; (2) decontextualization—analysis is done post-hoc on raw event traces, potentially relevant contextual cues are lost; (3) “one-way” communication—data flows from users to developers who must then infer meaning, no “dialogue” is established to facilitate mutual understanding; (4) batch orientation—hypothesis formation and analysis is performed after large amounts of (potentially irrelevant) data have been collected, no means for hypotheses to be analyzed and action taken while users are engaged; and (5) privacy issues—arbitrary events are collected without any explicit constraints on the purposes of collection, no way to provide users with discretionary control over what information is collected and what information is kept private.

The EDEM software addresses these issues and goes beyond existing approaches in supporting user involvement in development. Our system provides: (1) a multi-level event model—allowing agents to compare usability expectations against actual usage at reasonable levels of abstraction; (2) contextualization—taking action and collecting information while users are engaged in using the application; (3) two-way communication—helping initiate dialog between users and developers when breakdowns occur; and finally, (4) specializable monitoring and analysis—promoting a shift from batch-oriented data collection and analysis to hypotheses-guided collection and analysis.

**The Knowledge Depot for Information Review and Project Awareness**

In our scenario, groups need a mechanism for communicating about usability and other data,
design decisions, and general progress. The Knowledge Depot, an enhanced version of GIMMe (Zimmermann, Lindstaedt, Girgensohn, 1997), is a group memory that captures email conversations, email generated automatically by software (such as EDEM) and documents submitted over the web. It organizes this information, allowing users to browse through the information to rediscover (or learn for the first time) why different decisions were made, what problems were encountered as a result of those decisions, and allowing the user to regain some of the context in which those decisions were made. The Knowledge Depot is illustrated in Figure 8.

The system organizes its knowledge around a set of topics defined over time by all users of the system. The topics frame is shown in Figure 8 as a hierarchical list of discussion items. A topic is four things:

1. A phrase describing a concept, task or activity representing aspects of the group’s work;
2. A place where people go to find information;
3. A definition of the type of information the system looks for to determine whether something belongs in the topic; and
4. A destination that people will aim their messages at in order to have the message stored correctly for later retrieval.

A topic then might have a name like “Form Completion Sequence.” This says that any message that has “Form Completion Sequence” in the subject line will be captured in this topic, and people browsing for messages will know to look for email discussing the Form Completion Sequence within this topic. People having an email conversation about the Form Completion Sequence then need only put “Form Completion Sequence” in the subject, and CC the Knowledge Depot system for the information to be captured and put in an appropriate place. Furthermore, the hierarchical organization of topics allows a message that has “Form Completion Sequence Violation” in the subject to enter the “Form Completion Sequence” topic, and then enter the “Violation” subtopic. For more details on the capabilities of Knowledge Depot’s group memory component see (Kantor, Zimmermann, Redmiles, 1997)

Knowledge Depot moved from being a group memory to a Project Awareness tool when we added a feature allowing users to select a category and indicate that they wanted to be mailed summaries of all new information to arrive in that category. This feature is called the subscription feature. At its simplest, this approach can be used to allow members of a group to have an awareness of what information has been put into the group memory so that if in the future they need the information, they can say “didn’t I see that go into the group memory? I’ll go look there!” (Berlin, Jeffries, O’Day, Paepcke, Wharton, 1993). On a larger scale this feature allows different groups that must coordinate their work to remain aware of relevant information about each other. By allowing groups to subscribe to key pieces of information generated by other groups, they can remain aware of the aspects of changes in the work produced by those other groups that can affect them. When the system is set up to allow groups within a project to subscribe to one another’s categories, the tool becomes a mechanism to enhance project awareness.

The Knowledge Depot has many similarities to a newsgroup, if the newsgroup is used as a place where people Cc mail so that it can be made available to non-recipients of the message, and so that it can be stored for a time. The Knowledge Depot though is not constrained a single news-
group with a linear list of messages, nor is it equivalent to a large number of newsgroups where users have to determine which group (or groups) are most appropriate. The depot uses a flexible set of topics where any user can at any time add a new topic, and based on the definition of the topic, all existing messages will be checked to see if they belong in the topic (in addition to any other topics that the message is already in). Potentially, the entire topic structure could be removed, and replaced with a new topic structure, with messages automatically reorganizing themselves to account for the new hierarchy. While a newsgroup acts as an oversized shared email box, Knowledge Depot acts as a shared database. The database is used to organize all of a groups discussions and documents, and provides standard types of database queries and reorganizations.

**Future Work**

Our future plans for these systems include more tightly integrating the systems together, and user testing in industry. Rockwell, Lockheed-Martin and Bell Atlantic respectively will test Argo, EDEM and Knowledge Depot.

Plans for Argo include increased critic analysis and visualization support in the software design environment, and investigating the application of the design environment architecture to new domains such as requirements representations (CoRE method) and newer architectural specification languages (UML-Unified Modeling Language).

Plans for EDEM include improving the interface for the developers defining usage expectations and an examination of the expectation agent model to determine whether it can be generalized beyond usability testing.

A redesign of Knowledge Depot has been aimed at processing personal mailboxes, and acting as a personal mail tool to help evaluate the techniques used by the Depot in a setting where those actions will have immediate consequences for the user. This redesign will make Knowledge Depot a more familiar part of their work environment, and an easier place to turn to for answers.

**Conclusion**

The design of software systems is characterized as a problem of communication among stakeholders. Inherent to communication about software is the issue of the management of knowledge. There is knowledge about what constitutes a good design. There is knowledge about actual usage of prototype applications. There is knowledge about reviewing and improving the design based on actual usage and other insights. Communication and knowledge management are incorporated into a lifecycle model of human-centered software development based on three elements: design, usage, and review. These elements correspond to the above three kinds of knowledge communicated in design. Three systems are proposed to support the lifecycle: the Argo software architecture design environment for supporting design, the EDEN event monitoring software substrate for collecting information about system usage, and the Knowledge Depot project awareness tool for supporting the collection and review of designs.
**Web Site Guide**

Project information and selected publications are accessible from the PI’s home page:

http://www.ics.uci.edu/~redmiles/

A direct link to the research funded under this grant is the following:

http://www.ics.uci.edu/pub/eden/

**Acknowledgments**

The authors would like to thank Gerhard Fischer, Mike Atwood, Bea Zimmermann, Andreas Gergensohn, Frank Shipman, Kumiyo Nakakoji, and Stephanie Lindstaedt who worked on precursors to this work and who continue to provide insight and support.

This work is supported in part by funding from the National Science Foundation under grant CCR-9624846 and by Rockwell and the University of California under MICRO research project #97-148.

**References**


