Agent-Based Support for Communication between Developers and Users in Software Design

Andreas Girgensohn

NYNEX Science and Technology
White Plains, NY
E-mail: andreasg@nynexst.com

David F. Redmiles, Frank M. Shipman, III

Department of Computer Science
University of Colorado at Boulder
E-mail: {redmiles, shipman}@cs.colorado.edu

Abstract

Research in knowledge-based software engineering has led to advances in the ability to specify and automatically generate software. Advances in the support of upstream activities have focussed on assisting software developers. We examine the possibility of extending computer-based support in the software development process to allow end users to participate, providing feedback directly to developers. The approach uses the notion of "agents" developed in artificial intelligence research and concepts of participatory design. Namely, agents monitor end users working with prototype systems and report mismatches between developers' expectations and a system's actual usage. At the same time, the agents provide end users with an opportunity to communicate with developers, either synchronously or asynchronously. The use of agents is based on actual software development experiences.

1: Introduction

Research in knowledge-based software engineering has led to advances in the ability to specify and automatically generate software. This research has, in recent years, led to advances in supporting upstream software processes, including requirements analysis [18]. However, the tools that exist are intended to assist software developers only; end users are one-step removed from the systems being developed for them. We examine the possibility of extending computer-based support in the software development process to allow end users to participate, providing feedback directly to developers.

Our research effort shares many goals with research in computer-supported cooperative work (CSCW), but it also extends it by integrating it with concepts of participatory design [4][7]. Namely, end users are also "stakeholders" in a software product and need to be brought into the development process. We build on an evolutionary model of software development with the goal of improving the expected usability of a system by active, computer-based support for feedback from users of prototype systems.

Representative users would interact with prototype systems while software agents monitor their interactions and, under certain conditions, enable synchronous and asynchronous communication with the system developers or other stakeholders. This agent-based interaction complements traditional participatory design in which users are tightly integrated into the development process with face-to-face meetings. The agent-based approach enables the usability of a prototype to be observed unobtrusively while users perform their normal tasks. It also reduces the travel requirements between users' and developers' sites in companies that are geographically distributed. Furthermore, agents can monitor a greater number of users as a prototype becomes more robust or even after its distribution as a product.

This approach is based on real development experiences and current research into agent-based software environments. Namely, a system called Bridget was developed by the Intelligent Interfaces Group at NYNEX with the explicit goal of producing a more usable system for customer sales representatives who create new accounts for small businesses [1]. The development of Bridget illustrated a process in which participatory design was carried out manually. Simultaneously, an environment called HOS was developed in the Human-Computer Communication group at the University of Colorado to support active delivery of design decisions recorded in a hypermedia environment used by collaborating designers [17]. In our present work, we extend the active delivery mechanisms of HOS to provide support for the participatory design process followed in the development of Bridget. The discussion that follows explores the details of the participatory design of Bridget and the possibilities for their computer-based support.

2: Engineering a useful and usable system for the domain of service provisioning

The development of Bridget, a new system for tele-

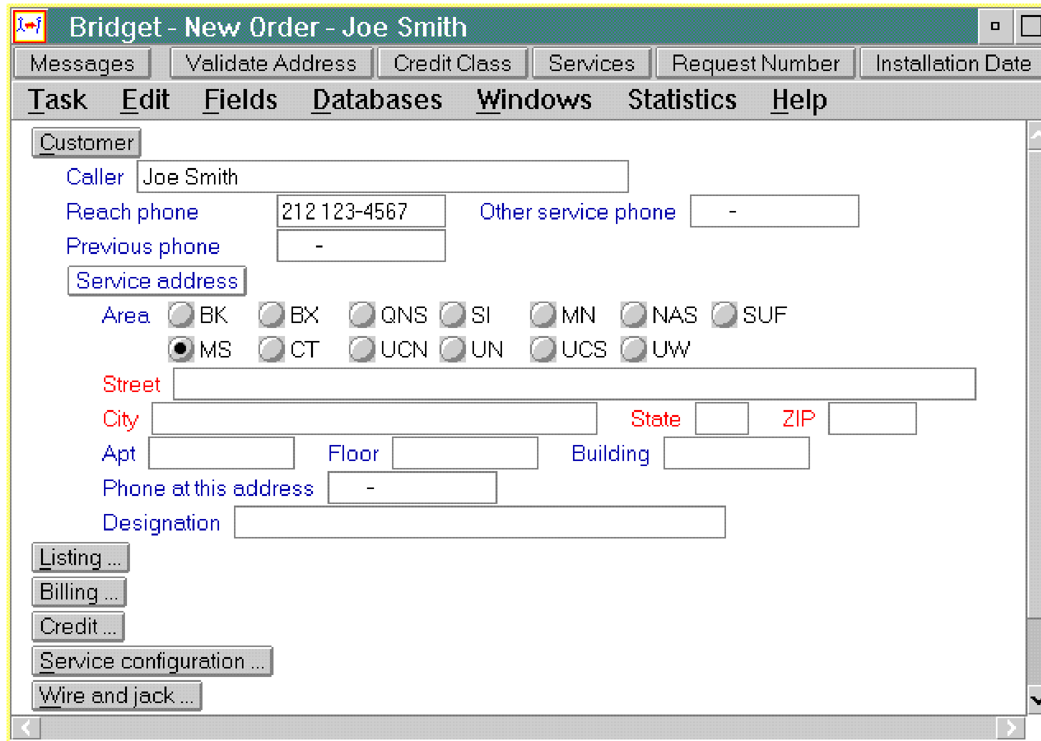

Figure 1: Bridget

Details of each section are hidden under the shaded titles in the form-based interface. Selecting a title reveals detailed fields that must be completed by the customer representative. In this screen, the first section has been expanded, revealing fields about the customer's service address.

phone service provisioning, provides an example of participatory software design. Many groups of stakeholders participated: users, managers, quality assurance staff, and billing and collections staff. Through this experience, the potential role and benefit of agents became clear.

2.1: The problem domain of service provisioning

Service provisioning is the design of a telephone service to meet a customer's requirements. The interaction between customer and service representative takes place over a telephone. During the conversation with the customer, the representative enters information into a variety of databases. The representative's main problem is accessing and updating mainframe databases. This problem is common to any operation that maintains a large database of customer and service information.

Currently service representatives attend three months of training before interacting with a customer. Much of this training time is spent learning how to work with the databases. Since there is little or no support for the representatives to manage database communications and navigate the service information space, the representatives are often too preoccupied to determine fully the needs of the customers and the services available to them.

2.2: The Bridget system

The goal in the development of Bridget was to provide the representatives with a uniform, window-based interface that hides the intrinsic complexities of the mainframe databases and provides support for customer service configuration (see Figure 1). Bridget consists of a single, simple interface through which representatives can access all the databases and submit orders to the appropriate departments (e.g., installation, billing). By decreasing the complexity of the interface and freeing representatives from complex database transactions, training time can be reduced. Additionally, context-sensitive help reminds representatives of information learned in training.

Bridget's user interface is organized as a dynamic form in which only the needed fields are shown. For example, the fields for deposit information are not shown if the customer's credit rating is good enough. The form is scrollable and divided into several sections that can be opened and closed by clicking the section titles.

2.3: User participation in developing Bridget

The views of several different groups of stakeholders had to be included in the development of Bridget. The intended users (customer representatives), their managers, quality assurance and billing people all contributed their knowledge of systems and tasks. These participants were the source of domain knowledge. Observations of the users in their work environment and discussions of various tasks took place. Also, once or twice a week, some participants entered service orders in Bridget. They noticed bugs and problems with the support for certain tasks and made suggestions for how to improve the system. Our goal is to facilitate this process with the support of agents.

We intentionally formulated the initial requirements for Bridget in a vague way. Rather than considering the requirements as being well-understood and inflexible, the requirements provided an initial direction for system development. Interviews were conducted with domain experts, observation of service representatives were made as they performed their task, and an analysis was performed on the existing documentation and tools. Prototypes demonstrated how a system might enable the service representative to conceptualize their task in terms of services and customer requirements rather than in terms of databases and prescribed methods and procedures.

User participation in the design of Bridget led to features that might not have been introduced otherwise. A good example for this is Bridget's built-in checklist. Observation of users of earlier prototypes of Bridget showed that they often overlooked filling some of the required fields. Users did not notice the problem until after Bridget refused to submit the service order to the mainframe database and they had to go back to all the missing fields. In response to the observed user problems, a built-in checklist was added to Bridget. The checklist uses color to indicate the parts of the form requiring work. Such an embedded checklist draws the user's attention without requiring any extra screen real estate.

2.4: The software development process

The Bridget experience involves a model of software development with the following elements or characteristics: domain orientation, user-centered task analysis, evolving prototypes, user participation, and constant communication between users, developers, and other stakeholders. The goals of usefulness and usability for Bridget were defined in terms of users' tasks (e.g., "enable representatives to spend more time servicing the customer"). These, in turn, could not be understood independently of the more general domain. Members of the design team not only spoke with potential end users but took some of the same training. Evolution played a key factor as there was not one, but many prototypes and trials with the end users.

The evolution of a system from this perspective occurs as a feedback mechanism by responding to discrepancies between a system's actual and desired states. Crucial to an understanding of the need for evolving prototypes is that the "desired state" cannot be clearly articulated. Irrespective of training or task analyses, many design concepts remain tacit [16]. The presence of an actual prototype forces design decisions and their full implications to become explicit [5][19].

The evolutionary model of software development we have espoused is not completely novel. Boehm [2] suggested a variation on the waterfall model that emphasized iteration. Henderson-Sellers [8] increases the granularity of iteration in a "fountain" model. However, a major difference between the model we espouse and that of others, is that we consider participation of end users to be an essential factor. The types of software systems these other models address are often customer driven, but the customer is only a manager of the end user or another person projecting end user needs at a distance. Iteration in other approaches is a means for software developers to converge on specifications rather than a means to evolve a prototype according to a growing understanding of the task at hand. No method to date has considered the potential of agents in software development.

3: Supporting user participation with agents

The key to the evolutionary model of software development is continuous communication between developers, end users, and other stakeholders. This ideal is problematic since access to users is often difficult and the overhead in developer time is large. To enable this process to occur when it may otherwise not be feasible or convenient we have begun to look at actively supporting communication via agents.

3.1: Expectation agents

In creating a prototype, developers have certain expectations of how users should perform tasks with a system. If these expectations do not match actual use, i.e. a "breakdown" in the developers' expectations has occurred [19]. The users might be performing tasks in a suboptimal manner or the developers' expectations might be based on false assumptions. Either the users need to be informed or the system needs to be redesigned, but most importantly, the discrepancy needs to be recognized and resolved early in the development process.

Expectation agents can help in observing "actual use" in contrast to "expected use." Such agents know about a

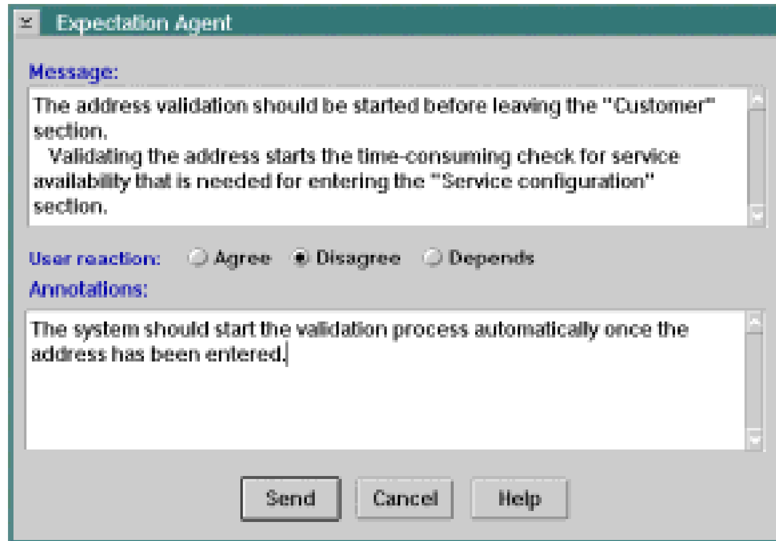

Figure 2: Message of an expectation agent

family of tasks the system is intended to support and the sequence of user interactions the developers envisioned for performing these tasks. In the case of a mismatch, agents may perform the following actions: (1) notify developers of the discrepancy; (2) provide users with an explanation based on developers' rationale, (3) solicit a response to or comment about the expectation. What action is appropriate depends on the type of discrepancy and whether the situation allows for immediate communication between users and developers.

In sum, this type of agent initiates a dialog between users and developers in which the developers communicate their *intent* to the users and the users have the opportunity to respond. Additional follow-up discussions can take place outside of the agent-based software development environment.

This model of expectation agents parallels methods used in the development of Bridget. In particular, developers had users redo their actual tasks using the newest prototype. The developers observed and asked for explanations of unexpected behavior after the task was completed. The users also used Bridget unsupervised and compiled a list of suggestions for changes.

3.2: Extending existing field methods

The use of agents makes it possible to involve more users in parallel, to exchange information more efficiently, and to provide a better documentation of the design decisions than a manual process of data collection such as followed in the development of Bridget. However, our flavor of agent-supported participatory design is not intended and cannot be a complete replacement for traditional participatory design. Direct interactions between developers and users are generally richer and are obviously essential before an initial prototype is created. We see agents as a way of making these interactions more productive by enabling them to be more convenient, more frequent, and more precisely directed with respect to requirements specification.

Furthermore, existing methods of task analysis provide developers with the basis for deriving expectations based on users' tasks and representing these expectations in such a way that similarities and discrepancies may be identified. Existing methods include cognitive walkthroughs [12], predicate representations [15], and task action grammars [9].

3.3: Facilitating communication

The notion of agents introduced with this work is that of a mechanism to facilitate communication. The communication enables a mutual understanding to evolve between developers and end users. The emphasis is on accurately matching a system to the needs of its intended users and less on enforcement of requirements that, in actuality, may be inappropriate for a problem situation. In this role, agents provide several advantages.

In the development of Bridget, participatory design was very helpful. However, this process required a lot of travelling both by the developers going to the users' offices and by the other participants coming to the developers' site. The geographical distribution of users and developers is increasingly a problem with national and multinational companies. Agents can alleviate this situation to a degree by supporting *distance communication*.

Distance communication is the notion that developers would not always have to travel to user sites to share design decisions. Video link up's to console monitors as well as for person-to-person communication would be facilitated by agents. A shared workspace between developers and users can be used to explore new ideas for the application. The combination of the shared workspace with video can be used to make the participants aware of each others focus of attention [10].

Using agents to initiate person-to-person communication creates the potential for agents to provide the user with an explicit or implicit choice as to whether a discussion occurs. It also provides the potential for the user to choose between synchronous and asynchronous modes of communication. Despite the richness of face-to-face communication, asynchronous communication mechanisms (e.g. electronic mail, design rationale) have advantages for discussions where comments and arguments are carefully crafted. Also, people seem more open to conveying or receiving negative criticism from a computer than a human.

4: Examples of agents in Bridget

Some examples are given below in order to illustrate how expectation agents would be used. The context for the examples is that system developers have already implemented a prototype version of Bridget (see Figure 1). The prototype is the result of establishing an initial set of requirements through discussions with end users (service representatives) and their managers, as well as an analysis of observing existing tasks the end users perform. Agents monitor the trial use of this prototype and support communication between developers and end users with the goal of refining the prototype and the requirements it implements.

As mentioned earlier, the primary task the Bridget end users perform is to establish a new service account for a customer. Over the phone, the service representative solicits information from the customer. The process requires accessing several legacy databases. The first of these databases is used to validate the new customer's address. In addition to validation, the query to this database returns information about services available at that address. This information can constrain the information fields in the service configuration section. In developing the initial requirements and prototype, the developers made an assumption that the customer representatives would initiate the address validation query by clicking on the "Validate Address" button (at the top of the screen in Figure 1) when they finished filling out that section and before proceeding to the next section. Doing so gives time for the query to be evaluated before the additional configuration information is needed.

The above assumption is expressed by the developers in an agent either through the agent editor or created by demonstration (see Section 5.2). If the customer representative opened another section (e.g., "Listing") before starting address validation, a discrepancy between expected and actual use would occur; the agent would be triggered. As a result, the representative would be presented with rationale and the possibility to respond, shown in Figure 2.

Another use for an expectation agent in Bridget is to determine any changes to the order of the sections in the form. While customer representatives may fill out sections in any order, they can be more efficient if the sections are placed in the normal usage order. An agent can keep statistics about the order followed by many representatives. Such usage patterns will be reported back to developers. Patterns found can then be used to engage users in discussions about whether a changed order would be better or whether some fields should be put in different sections. The goal of such changes being a more natural flow, minimizing the jumping back and forth between sections.

A user-initiated mechanism, such as a "suggestion" button, can be used to support the volunteering of comments and suggestions by Bridget users. Beyond traditional electronic mail, the suggestion button could combine the user's suggestion together with the user's current context, such as recent actions, to be sent to the developers. This information can improve the developer's understanding of the user's situation and suggestion.

5: A software substrate implementing agent-based support

The preceding section points to a number of ways in which agents can aid the participatory process that occurred during the creation of Bridget. Our initial investigation of agents is based on the Hyper-Object Substrate (HOS) [20]. HOS provides a domain-independent framework, combining characteristics of hypermedia and knowledge engineering systems.

5.1: Agents in the Hyper-Object Substrate

Agents in HOS use the information represented in the form of attributes and relations to determine when to take some action. They search for objects with certain attributes within the system and perform operations based on what, if any, objects they locate.

Agents consist of a trigger, a query, and an action. This representation, detailed in Figure 3, is similar to the representation of agents in OVAL [14]. The trigger specifies when the agent evaluates its query. Any objects that are returned by the query are passed to the action. If no objects

Formal representation for agents:

```
Agent = {<trigger> <query> <action>}
<trigger> = {immediate | every action |
  when requested | when displayed |
  particular user event}
<query> = {object has property | relation
  between objects exists | user events
  match/don't match pattern | usage
  statistics in/out of range}
<action> = {present message | add
  bookmark | collect/display objects |
  select/highlight objects | establish
  A/V communication link | record work
  context}
```

How agents are activated:

```
WHEN <trigger>
IF <query> returns information
THEN do <action> with information
  returned
```

Figure 3: Representation for agents

Options currently available in HOS are shown in normal type, extensions for supporting user/developer communication are shown in italics.

are returned, the action is not performed.

The existing set of triggers, queries, and actions for agents is being extended in order to provide support for participatory design. This includes a representation to allow queries which look for cases of unexpected user behavior and actions to create real-time textual, audio, or video links.

Triggers: Controlling the Activity. The first component of an agent object is the trigger. The trigger defines when an agent is active, i.e. when it will evaluate its query. As such the trigger defines the agent's control characteristics—whether the agent will interrupt the user or not. HOS includes the following options for triggering: (1) check every action of the user, (2) check only when requested by the user, (3) check when the agent is displayed, and (4) check immediately after the definition of the agent.

The addition of a trigger for specific user events (i.e. opening form, pressing button, exiting field) will enable better control of when agents may interrupt users. This control is necessary since agents, like the developers observing Bridget use, should not interrupt users in the middle of partially completed tasks.

Queries: Looking at the Current Situation. The second component of a HOS agent is a query. The query defines the information that must be located before the agent will execute its action. HOS currently allows queries for objects with some combination of attributes and relations. For example, a query could search for objects which have the attribute "Assigned to" with a value of "David" and the attribute "Completion date" with value "April 20".

Additional queries important for representing expectation agents are matching user events to expected events and comparing usage statistics to expected usage. The agent whose output is shown in Figure 2 would use event patterns to compare actual use (user opens "Listing" section) to expected use (validate address before opening another section).

Actions: Advertisement and Collection. The final component of a HOS agent is an action. The trigger and query together determine whether an action will be taken. The action defines the support service that the agent will provide. In HOS, the options for actions are (1) present a message to the user, (2) create a hypertext bookmark, (3) collect found objects in the current view, and (4) select/highlight found objects.

An action creating an audio/video link is needed for agents supporting distance communication. Besides facilitating communication between developers and users, this action would implement agents which connect users to peers, "power users", or managers as well. The action to save the work context requires a representation of the current state and the last several events to provide developers with information needed to interpret users' comments and concerns.

5.2: Interface for the creation of agents

Interfaces for creating agents are important because of the potential for the benefits provided by agents to be overshadowed by the costs of creating and maintaining them. To address this concern we have begun looking at both specialized editors and the use of programming by demonstration for manipulating agents.

In the development of editors for defining agents, methods and techniques used for achieving end-user modifiability [6] and incremental formalization [20] play a prominent role. Developers can currently create or modify agents in the "agent editor" shown in Figure 4. The agent editor provides popup menus containing a set of triggers and actions. After selecting a particular trigger or action, the developer is asked to specify extra information needed for that trigger or action. For example, when a developer selects the action to "present a message" or "add a suggestion to the bookmark list" the system asks what message should be displayed or what view should be added to the bookmark window.

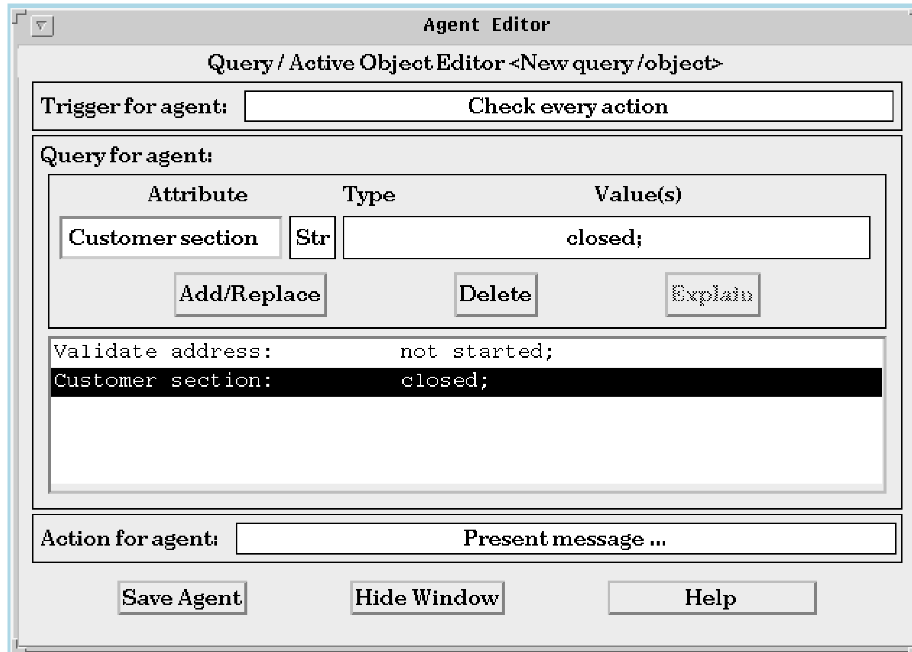

Figure 4: The agent editor in HOS

The top part of the interface is where the user chooses a trigger from a popup menu of choices. In the middle the user defines the query and in the lower part the action to be performed if objects are returned by the query. The agent being edited will check to see if the user closes the customer section before address validation is started. The result of the agent's action is shown in Figure 2.

The query definition area within the agent editor currently limits queries to searching for objects with a particular set of attributes. The need to express developer expectations and our experience with the use of HOS agents [17][20] has indicated the need for greater expressiveness within query definition. But greater expressiveness generally comes at a reduction in ease of use.

To reduce the burden of defining expectation agents a method similar to programming by demonstration [3] may be useful. Developers would demonstrate how a certain task should be performed. These demonstrations are recorded and analyzed for patterns which are turned into agents. Rules identify which parts of an interaction are important so that the generation of agents for less important interactions can be avoided. After the agents are generated, the developers can fine-tune them if necessary, e.g., by introducing variables or ranges of values. After the agents are generated, they should be tested with a few users, i.e., the user feedback mechanism would be turned off to see whether the number of interruptions of the users is reasonable. An advantage of this approach to agent creation is that multiple agents can be defined from the same demonstration(s). Also, the implicit expectations of the developer may be captured through their creation of demonstrations.

Managing and adapting agents is also a concern. In particular, experiences with HOS point to the necessity of developing better interfaces for agent management. The small knowledge-based systems developed within the class projects had only on the order of tens of these agents. Larger projects, with longer lifetimes and many stakeholders creating agents, can expect many more. This leads to the issue of how a system can provide users with control of these agents without requiring too much extra effort. We have begun investigating methods for manipulating groups of agents at once.

Another important aspect is the adaptation of agents to changing prototypes of the system. The agent mechanism needs to be able to identify which agents are affected by a change and ask the developers whether these agents should be discarded or changed.

Trade-offs between the expressiveness and ease-of-creation of agents are guaranteed. By providing a variety of methods for defining agents we hope to allow developers to make decisions concerning agent creation within the context of their work.

6: Conclusions

Complex relationships exist among people, tools, and tasks. One of the crucial aspects of these relationships is that they all change over time in a co-adaptive manner: people adapt their practices and problem-solving approach to the affordances of a tool, and people adapt their tools to better support their practices and problems [4][13]. Our approach supports co-adaptation by supporting communication of *design intent* among software developers, from software developers to end users, and from end users back to developers. Alan Kay notes that "many are just discovering that user interface design is not a sandwich spread [11];" our approach seeks to improve the usefulness and usability of a software system by focusing on the process of systems development and not just on the end result.

The process of developing Bridget has been very helpful in identifying the potential benefits that agents supporting participatory design can have. Agents can communicate the designers' intent to the users and establish communication links. Combining these capabilities in a software development environment will help in the production of more user-centered software.

Acknowledgments

We thank Hiroshi Ishii for his many helpful comments about this formative work. We also thank Mike Atwood and Gerhard Fischer for supporting this research. This work is part of an on-going project also involving Beatrix Zimmermann, Alison Lee, Althea Turner, Bart Burns, and Jonathan Ostwald. The effort at C.U. is funded by NYNEX and by the Advanced Research Projects Agency.

References

[1] Atwood, M.E., Burns, B., Girgensohn, A., Zimmermann, B. Dynamic Forms: Intelligent Interfaces to Support Customer Interactions, Technical Memorandum TM 93-0048, NYNEX Science & Technology, White Plains, NY, Dec. 1993.

[2] Boehm, B.W. A Spiral Model of Software Development and Enhancement. *IEEE Computer* 21, 5 (May 1988), pp. 61-72.

[3] Cypher A. (ed.), *Watch What I Do: Programming by Demonstration*. The MIT Press, Cambridge, MA, 1993.

[4] Ehn, P. *Work-Oriented Design of Computer Artifacts*. Almquist & Wiksell International, Stockholm, Sweden, 1988.

[5] Fischer, G., Lemke, A.C., McCall, R., Morch, A. Making Argumentation Serve Design. *Human Computer Interaction* 6, 3-4 (1991), pp. 393-419.

[6] Girgensohn, A. End-User Modifiability in Knowledge-Based Design Environments. Ph.D. Thesis, Department of Computer Science, University of Colorado, June 1992.

[7] Greenbaum, J., Kyng, M. *Design at Work: Cooperative Design of Computer Systems*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1991.

[8] Henderson-Sellers, B. The O-O-O methodology for the object-oriented life cycle. *ACM SIGSOFT Software Engineering Notes* 18, 4 (Oct. 1993), pp. 54-60.

[9] Hoppe, H.U. Task-Oriented Parsing: A Diagnostic Method to be Used by Adaptive Systems. In *Proceedings of CHI '88*, ACM, New York, May 1988, pp. 241-247.

[10] Ishii, H., Kobayashi, M., Grudin, J. Integration of Inter-Personal Space and Shared Workspace: ClearBoard Design and Experiments. In *Proceedings of CSCW '92,* ACM, New York, 1992, pp. 33-42.

[11] Kay, A. C. In *The Art of Human-Computer Interface Design*, B. Laurel, Ed. Addison-Wesley Publishing Company, Reading, MA, 1990, pp. 191-207.

[12] Lewis, C., Polson, P., Wharton, C., Rieman, J. Testing a Walkthrough Methodology for Theory-Based Design of Walk-Up-and-Use Interfaces, In *Proceedings of CHI '90*, ACM, New York, 1990, pp. 235-242.

[13] Mackay, W. E. In *CHI'92 Basic Research Symposium*, 1992.

[14] Malone, T.W., Lai, K.Y., Fry, C. Experiments with Oval: A Radically Tailorable Tool for Cooperative Work. In *Proceedings of CSCW '92,* ACM, New York, 1992, pp. 289-297.

[15] Mannes, S., Kintsch, W. Routine Computing Tasks: Planning as Understanding, *Cognitive Science* 3, 15 (1991), pp. 305-342.

[16] Polanyi, M. *The Tacit Dimension*, Doubleday, Garden City, NY, 1966.

[17] Reeves, B.N., Shipman, F.M. Supporting Communication between Designers with Artifact-Centered Evolving Information Spaces. In *Proceedings of CSCW '92,* ACM, New York, 1992, pp. 394-401.

[18] Reubenstein, H.B., Waters, R.C. The Requirements Apprentice: Automated Assistance for Requirements Acquisition, *IEEE Transactions on Software Engineering* 17, 3 (March 1991), pp. 226-240.

[19] Schön, D. *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, New York, 1983.

[20] Shipman, F.M., McCall, R. Supporting Knowledge-Base Evolution with Incremental Formalization. In *Proceedings of CHI '94,* ACM, New York, 1994, pp. 285-291.