EDEM: Intelligent Agents for Collecting Usage Data and Increasing User Involvement in Development

David M. Hilbert **Jason E. Robbins** **David F. Redmiles**

Dept. of Information and Computer Science
University of California, Irvine
Irvine, California, 92697-3425 USA
+1-714-824-3100
{dhilbert,jrobbins,redmiles}@ics.uci.edu

ABSTRACT
Expectation-Driven Event Monitoring (EDEM) provides developers with a platform for creating software agents to collect usage data and increase user involvement in the development of interactive systems. EDEM collects information that is currently lost regarding actual usage of applications to promote improved usability and a more empirically grounded design process.

Keywords
Event monitoring, intelligent agents, expectation agents, usability engineering, software engineering

INTRODUCTION

Software evolution is largely driven by issues arising during use. Once a system has been deployed, problems invariably arise that developers did not anticipate. When the environment in which a system is deployed and/or its users behave in unexpected ways, usability and performance may be affected. This can, in some cases, lead to lost data and productivity and even threats to safety and security.

Involving end users in the development of interactive systems increases the likelihood those systems will be useful and usable [1][5][6]. However, user involvement should not end once initial requirements, design, and prototypes have been completed. If developers could continue to watch end users interacting with systems once they have been deployed, they could learn a lot about users' tasks and work environments that would suggest improvements in system design, on-line help, documentation, and training. Unfortunately, this is not feasible in general.

Expectation-Driven Event Monitoring (EDEM) allows developers to create automated software agents to monitor applications as they are being used and report data back to developers when environmental features or user behavior violate developers' expectations.

Currently, most usage data is lost. Information that *does* get back to developers is often communicated through slow, *ad hoc*, and manual channels. EDEM helps automate the task of collecting usage information, allowing developers to base design decisions on empirical data.

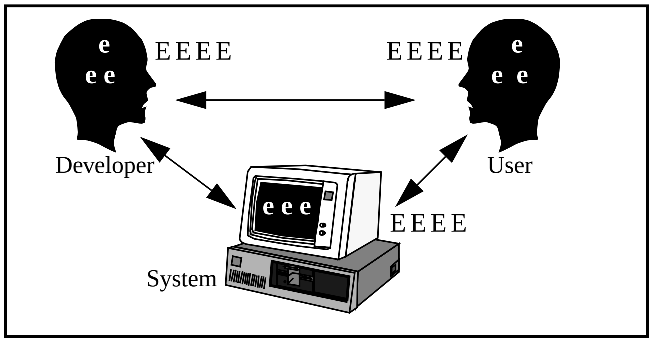


Figure 1. Usability expectations in the development process.

APPROACH

Usability breakdowns occur when developers' expectations about system usage do not match users' expectations. Several benefits can be realized when these mismatches are detected and resolved [3].

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 only minor variation, while a user may hold the expectation that independent sections may be filled out in any order.

Figure 1 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 uppercase "E" represents an explicit expectation. Several usability methods seek to make implicit expectations of developers and users explicit (e.g., cognitive walk-throughs, participatory design, and use cases). Expectations embedded in the system can be made explicit though representations that we discuss below. Many expectations will remain implicit despite these methods. We can treat such expectations as unknowns and attempt to detect mismatches

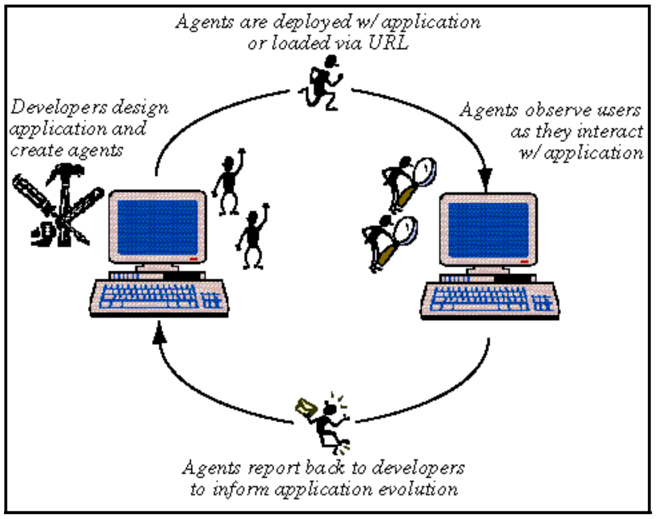

Agents are deployed w/ application or loaded via URL
Developers design application and create agents
Agents observe users as they interact w/ application
Agents report back to developers to inform application evolution

Figure 2. The EDEM development cycle.

by comparing observed usage against expectations that *have* been made explicit.

Once a mismatch between users' and developers' expectations is detected it can be resolved 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. Learning that they are not expected to type full URL's in Netscape Navigator™ can lead users to omit characters such as "http://".

Detecting breakdowns or difference in expectations is useful, but aligning expectations requires knowledge of the other party's expectations and 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.

IMPLEMENTATION

We support detecting and resolving mismatches in expectations (as described above) by allowing developers to encapsulate usability expectations in the form of intelligent agents that monitor graphical user interface events. These *expectation agents*, or EA's, continually monitor usage of the application and perform various actions when encapsulated expectations are violated (Figure 2).

Other authors have proposed event monitoring as a means for collecting usability data, however, existing approaches usually 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], (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. (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.

EDEM addresses these issues and goes beyond existing approaches in supporting user involvement in development. Specifically, EDEM provides the following benefits: (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.

Multi-Level Event Model

EDEM is based on a multi-level event model to allow event monitoring to be raised to the level of expectations (Figure 3). At the lowest level are *physical events*, such as pressing keys with one's fingers or moving the mouse with one's hand. *Input device events*, such as key and mouse interrupts, are generated by hardware in response to physical events. *Window system events* associate input device events with windows and widgets on the screen.

Window system events are the lowest level events that EA's can monitor. Events at this level include button presses, list and menu selections, focus events in input fields, and window movements and resizing. An EA could, for example, perform input field validation when the "Submit" button is pressed on a web-based input form.

Application level events are generated by the expectation-driven event monitoring substrate based on computations involving window system events and global window system

Goal/Problem-Related
(e.g. ordering new software)

Domain/Task-Related
(e.g. providing billing information)

Abstract Interaction Level
(e.g. providing values in input fields)

Application Level
(e.g. shifts in editing attention)

Window System Events
(e.g. shifts in input focus, key events)

Input Device Events
(e.g. hardware generated key or mouse interrupts)

Physical Events
(e.g. fingers pressing keys or hand moving mouse)

Figure 3. A multi-level model of user events.

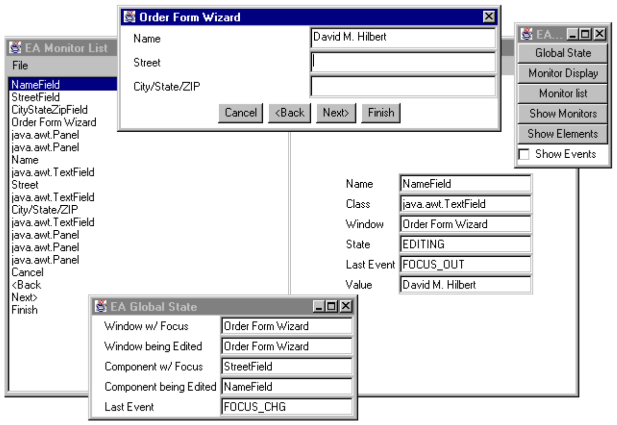
state. Events at this level are intended to indicate changes in the application interface that correlate with salient shifts in the users' state of mind.

Consider a user editing a field at the top of a form, then pressing tab repeatedly to edit a field at the bottom of the form. In terms of window system events, input focus shifted several times between the first and last fields. In terms of application level events, the user's editing attention shifted directly from the top field to the bottom field. Until the user starts actually editing another application component, the monitoring substrate assumes the user's editing attention has not shifted.

Application level events are associated with application components, groups of components, and application windows. To infer that the user has shifted editing attention away from a given component, group of components, or window, the monitoring substrate must look for editing events in components outside of that component, group of components, or window. The event-monitoring substrate performs the computations required to generate application-level events so that agents can detect such events on particular application components without monitoring all components on the screen. Not only does this simplify individual EA's, but it also factors code that would be common across agents and avoids redundant computation.

Abstract interaction events occur when recurring, idiomatic patterns of user interface events (from the window system and/or application levels) are recognized. For example, an abstract interaction event may be generated when a user provides a value by manipulating an application component. If that component were an input field, this would mean the field had been edited, was no longer being edited, and now contains data. The patterns of lower level events that indicate an abstract interaction event such as VALUE_PROVIDED will differ from one type of application component to another, and from one application domain to another. This is why detection of abstract interaction events is not performed directly by the event monitoring substrate. Abstract interaction event EA's can be defined to generate these events when the lower level events indicating them have occurred, so that other interested (or higher level) agents can use them in their own event monitoring.

Domain/task-related and *goal/problem-related events* are at the highest levels. Unlike previous levels, these events indicate progress in the user's tasks and goals. Detecting these events is straightforward when interfaces provide explicit support for structuring tasks or indicating goals. For example, Wizards in Microsoft Word™ lead users through a sequence of steps in a predefined task. EA's can easily recognize the user's progress by recognizing simple lower level events, such as button presses on the "Next" button. In other cases, task and goal related events might be detected by EA's working individually or in groups to recognize combinations of lower level events. For example, the goal of placing an order includes the task of providing customer contact information. An expectation agent could recognize the task-related event CONTACT_INFO_PROVIDED by recognizing the VALUE_PROVIDED abstract interaction events generated for every field in the contact information section of the form. In order to allow EA's to understand task



Name	David M. Hilbert
Street	
City/State/ZIP	

NameField
StreetField
CityStateZipField
Order Form Wizard
java.awt.Panel
java.awt.Panel
Name
java.awt.TextField
Street
java.awt.TextField
City/State/ZIP
java.awt.TextField
java.awt.Panel
java.awt.Panel
java.awt.Panel
Cancel
<Back
Next>
Finish

Name	NameField
Class	java.awt.TextField
Window	Order Form Wizard
State	EDITING
Last Event	FOCUS_OUT
Value	David M. Hilbert

Window w/ Focus	Order Form Wizard
Window being Edited	Order Form Wizard
Component w/ Focus	StreetField
Component being Edited	NameField
Last Event	FOCUS_CHG

Figure 4. Monitoring of a simple order form wizard.

and goal related events within the context of a broader process, the event monitoring substrate could be integrated with process and workflow modeling tools.

Our multi-level event model provides a hierarchy of abstraction not provided by traditional event monitoring approaches. Figure 4 shows a simple wizard for filling out an order form that has been connected to the event monitoring substrate. Notice that while input focus has shifted from the name field to the street field, the event monitoring substrate is indicating that the user's editing attention has not yet (been confirmed to have) shifted.

Contextualization

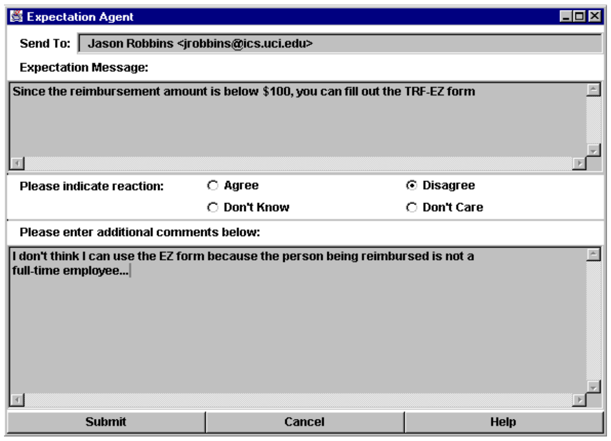
Currently, users and developers bear full responsibly for recognizing when breakdowns occur, determining the reasons for the breakdown, and deciding how to recover. Because EA's operate within the context of use, they can assist users and developers in making these determinations.

When a breakdown occurs, EA's can provide developers with important contextual information such as system state and event history. They may also collect information from users regarding the reasoning and incidents leading up to breakdowns, *while* that information is still fresh in users' minds. When breakdowns are due to errors in the code, EA's can help provide developers with much richer contextual information for bug reporting purposes than has typically been possible. EA's can help make external bug reports as useful as internally generated bug reports.

Another benefit of EA's is that they can operate in real usage contexts since they don't noticeably affect user interface operation. Also, since monitoring is unobtrusive, EA's are less likely than direct observation methods to distract users or influence their behavior.

Two-Way Communication

When breakdowns occur, it may not be enough to simply provide context. Dialogue between users and developers may need to be established in order to evolve mutual understanding. When an EA detects a breakdown, it can prompt the user to communicate with developers (Figure 5). The same facilities can also be used to volunteer comments when EA's fail to detect breakdowns experienced by the user. Communication can be synchronous or asynchronous, via voice, video, or electronic mail.



Send To: Jason Robbins <jrobbins@ics.uci.edu>

Expectation Message:

Since the reimbursement amount is below $100, you can fill out the TRF-EZ form

Please indicate reaction: Agree [✓ Disagree] Don't Know Don't Care

Please enter additional comments below:

I don't think I can use the EZ form because the person being reimbursed is not a full-time employee...

Figure 5. An expectation agent message dialog.

Once communication has been initiated, ongoing dialogue between developers and users may continue outside the scope of the agent-based system. The communication policy appropriate will depend on the development situation. For instance, a direct video link might work well in small-scale, in-house development situations, while asynchronous policies might be preferable in Web-based product development. When users greatly outnumber developers, information gathered from EA's will need to be filtered through information management mechanisms before being presented to developers. Mediator roles [4] may need to be established to manage communication between users and developers.

Specializable Monitoring and Analysis

Expectation-driven event monitoring represents a shift from traditional batch-oriented approaches to a more proactive, hypothesis-guided approach. Instead of forming and analyzing hypotheses after large amounts of (potentially irrelevant) data have been collected, data collection can be tuned based on a-priori hypotheses (or expectations) that are analyzed *while* users are engaged. Our approach is hypothesis-guided in that only data, and results of analyses, that are relevant to specific hypotheses (expectations) are reported.[1]

Specializablity makes monitoring tractable on a larger scale than is possible with traditional approaches. It is scalable in terms of the number of users that can be monitored because it allows analysis to be computed on the client-side. This means that computation can be distributed among potentially thousands of processors, and only relevant data, or results of analyses, need to be reported to developers. Thus, usability information can be captured on a scale that is statistically significant, observations can be categorized a-priori as well as a-posteriori, and factor analysis is facilitated.

Specializable monitoring and analysis can thus contribute to an empirically guided development process. Effort can be focused on those changes that will benefit the greatest number of users, or resolve the greatest number of non-trivial breakdowns. Also, the impact of proposed changes can be analyzed in terms of how well they "agree" with existing user expectations. For instance, before making a change, a developer could deploy an agent to the current user base to look for user expectations that would be violated by introducing the change.

Since EA's can be dynamically added or removed, investment in EA's can be made incrementally. There is no need to delay deployment of a product until all EA's are in place. Even a single EA can yield some useful feedback.

1. EA's can support traditional event monitoring in addition to the expectation-based approach we advocate here.

SUMMARY AND CONCLUSIONS

In summary, we propose a conceptual model of how usage expectations might be treated explicitly in the software development process and describe a system that can help automate the task of detecting and resolving breakdowns. EDEM helps developers collect usage data in real usage contexts, thus promoting an empirically guided development process in which developers can base design changes on actual usage information.

ACKNOWLEDGMENTS

The authors would like to thank A. Girgensohn, F. Shipman, A. Lee, and A. Turner who worked on precursors to this work and continue to provide insight and support. This work is financially supported by the National Science Foundation, grant number CCR-9624846, and by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government. Approved for Public Release - Distribution Unlimited.

REFERENCES

1. Baecker, R. M., Grudin, J., Buxton, W. A. S., Greenberg S., eds. (1995) *Readings in Human-Computer Interaction: Toward the Year 2000*. Morgan Kaufmann Publishers, Inc. San Francisco, CA, USA.
2. Chen, J. (1990) Providing Intrinsic Support for User Interface Monitoring. In *Human-Computer Interaction - INTERACT '90*.
3. Girgensohn, A., Redmiles, D. F., and Shipman, F. M. III. (1994) Agent-Based Support for Communication between Developers and Users in Software Design. In *Proceedings of the Knowledge-Based Software Engineering Conference '94*. Monterey, CA, USA.
4. Grudin, J. (1991) Interactive Systems: Bridging the Gaps between Developers and Users. *IEEE Computer.* April, 1991.
5. Lewis, C. and Rieman, J. Getting to Know Users and their Tasks. In Baecker, R. M., Grudin, J., Buxton, W. A. S., Greenberg S., eds. (1995) *Readings in Human-Computer Interaction: Toward the Year 2000*. Morgan Kaufmann Publishers, Inc. San Francisco, CA, USA.
6. Nielsen, J. (1993) *Usability Engineering*. Academic Press, Inc., Cambridge, MA, 1993.