IST PROGRAMME

Action Line: IST-2002-8.1.2

End-User Development

Empowering people to flexibly employ advanced information and communication technology

Contract Number IST-2001-37470

D3.2 Proceedings Workshop on End-User Development held in conjunction with the ACM CHI 2003 Conference

Editors:

Henry Lieberman, Fabio Paternò, Alex Reepenning, Volker Wulf

Summary

This document contains the proceedings of the workshop on End-user Development held in conjunction with the ACM CHI conference (Fort Lauderdale, 6 April 2003).

October 2003

Workshop on

END USER DEVELOPMENT

Fort Lauderdale
Sunday April 6, 2003

Henry Lieberman
MIT Media Lab

Fabio Paternò
ISTI-CNR

Alexander Repenning
University of Colorado

Volker Wulf
FRAUNHOFER-FIT

Table of Contents

Cooperative Development & Realization of Situation Dependent Mobile Services ………………………. 1
Michael Amberg and Jens Wehrmann

Personal Wizards: collaborative end-user programming …………………………………………………... 7
Lawrence Bergman, Tessa Lau, Vittorio Castelli and Daniel Oblinger

Software Engineering for End-User Programmers ………………………………………………………… 12
Margaret Burnett, Gregg Rothermel and Curtis Cook

End-User Programmers Need Improved Development Support ……………………………………….... 16
David A. Carr

Software Shaping Workshops: Environments to Support End-User Development ……………………….. 19
M.F.Costabile, A. Piccinno, D. Fogli and P. Mussio

Supporting End User Programming of Context-Aware Applications ……………………………………... 23
Anind K. Dey and Tim Sohn

Some Generic Mechanisms for Increasing the Usability of EUD Environments ………………………….. 27
Simone Diniz Junqueira Barbosa, Philippe Palanque and Rémi Bastide

End user Development by Tailoring Blurring the border between Use and Development………………… 33
Yvonne Dittrich, Lars Lundberg and Olle Lindeberg

User-Programming of Net-Centric Embedded Control Software ………………………………………….. 39
Jens H. Jahnke, Marc d'Entremont, Mike Lavender and Andrew McNair

The EUD-Net’s Roadmap to End-User Development ……………………………………………………... 43
Markus Klann

Objects for Users End-User Development of a Cooperative Information Model ………………………….. 47
Barbara Kleinen

Toward a Culture of End-User Programming Understanding Communication
about Extending Applications …………………………………………………………………………… 51
Cecília Kremer Vieira da Cunha and Clarisse Sieckenius de Souza

Feasibility Studies for Programming in Natural Language ………………………………………………… 55
Henry Lieberman and Hugo Liu

Using Domain Models for Data Characterization in PBE …………………………………………………. 56
José A. Macías and Pablo Castells

End User Programming for Web Users ……………………………………………………………………. 61
Robert C. Miller

Studying Development and Debugging to Help Create a Better Programming Environment …………….. 65
Brad Myers and Andrew Ko

Towards a Research Agenda in End User Development …………………………………………………... 69
Fabio Paternò

Prototyping Interactivity before Programming …………………………………………………………….. 73
John Sören Pettersson

Fostering End-User Participation with the Oregon Groupware Development Process ……………………. 76
Till Schuemmer and Robert Slagter

Design Principles and Claims for End-User Development ………………………………………………… 80
Alistair Sutcliffe

The Pragmatic Web: Customizable Web Applications ……………………………………………………. 84
Alexander Repenning

Supporting End-User Development of Componet-Based Application
by Checking Sematic Integrity ………………………………………………………………………... 88
Markus Won

Cooperative Development & Realization of Situation Dependent Mobile Services

Michael Amberg, Jens Wehrmann

Friedrich-Alexander-University of Erlangen-Nuremberg, Department of Economics,

Chair for Business Information Systems III, Lange Gasse 20, 90403 Nuremberg, Germany,

email: amberg@wiso.uni-erlangen.de, jens.wehrmann@wiso.uni-erlangen.de

1 Motivation

As the experiences with mobile services are showing, service concepts known from the stationary internet cannot be transferred into the mobile environment. Instead, only those mobile services tend to be successful that take the specific features of the user's context into account and apply this information to generate an added value for the mobile customer. Services that automatically adapt to the context are termed ***Situation Dependent Services*** (SDS). Initial examples for SDS are mobile *Location Based Services* (LBS) or personalized internet services. By now, LBS are based on a low level of situation dependency and use mostly simple filtering techniques with database lookups.
A popular example of a **service platform** is i-mode that has been developed in Japan and was recently launched in Europe. In i-mode only basic situation dependent services are supported by now. One possibility to enrich and simplify the usage of a mobile service, is the adaptation to the individual end-user's needs. From the service provider's point of view there are several aspects that make the realization of situation dependent mobile services complex:

- Service providers are mainly context providers, not network providers.
- Service providers are not allowed to store data about the user's behavior. (only with an end-user's agreement of confirmation or anonymization of information)
- Service providers have no access to sophisticated information about the end-user's situation, which is important for an efficient adaptation of mobile services.
- Currently, there are no standards for the sharing and the accounting of situation specific information in mobile radio networks, that allow to transfer differentiated information about the user anonymously.

Important **research questions** for the development and realization of situation dependent mobile services are:

- **Understanding the End-User's Situation:** How can an end-user's situation be defined? Which information about the end-user's situation are relevant for situation dependent mobile services? How can situations be classified, described, enriched and transferred?
- **The Development and Usage Process:** How could the lifecycle of a *Situation Dependent Mobile Service* be described? What opportunities are connected in this lifecycle? For Whom?
- **Analysis of the End-User's Acceptance:** What about the end-user's acceptance of *Situation Dependent Mobile Services*? Which properties are important for the end-user? How can the user be protected from becoming a transparent individual? How can this be communicated to the end-user?
- **Designing Cooperative Development:** In which way should the interaction, the information flow and the cooperation for service distribution work. Who plays which role? How does the accounting work?

- **Realization of *Situation Dependent Mobile Services*:** What technical conditions have to be considered? What kind of architecture is needed? Which cooperation partner has to accomplish which technological task?

The balanced combination of these elements in a methodical framework is regarded to be fundamentally important for the development of successful *Situation Dependent Mobile Services*.

2 Understanding the End-User's Situation

A situation concept should classify the mobile situation context and make the customer's situation context suitable for the cooperative providing of situation dependent services. A situation can be distinguished into the measurable aspects of a user's situation according to three dimensions: ***Time***, ***Place*** and ***Person***. These dimensions correlate with the primary situation determinants that are presently transmittable in mobile networks. *Time* and *Place* are the common and most obvious dimensions that are easy to measure. The *Person* summarises all measurable aspects of a person. It includes the identity and demographic information as well as information about the specific behaviour. Depending on the scope of application this basic classification can be extended (Amberg, Wehrmann, 2002).
The proposed situation concept (Amberg et al. 2002,1) is based on the idea that the adaptation of a mobile service according to the customer's situation context provides a real benefit and an improved user experience. A mobile service that is able to access the context is much more able to solve a problem efficiently and to provide a certain added value compared to a service without this information.
The situation concept includes a three-step process to determine the user's situation for a mobile service:

- **Determination:** In a first step, the elementary situation information (called situation determinants here) are measured. For the *identification* of a mobile customer in mobile GSM networks the *Mobile Subscriber International Subscriber Directory Number* (MSISDN) can be used. To calculate the *position* of the mobile terminal, there are network or terminal based solutions. By merging this information with the world time, the end-user's *local time* can be calculated.
- **Interpretation:** On the basis of the situation determinants and by consulting additional data sources, detailed information about the user's situation is derived.
- **Description:** The derived knowledge about the user's situation is then coded in a suitable mark-up language.

3 The Development and Usage Process

The development and usage process describes the main process steps for providing situation dependent mobile services. The usage cycle (Amberg et al. 2002,1) differentiates the following three basic (not disjunctive) categories of services: *Individualised Services* are any kind of user initiated services. They are adapted to the individual customer's needs. *Proactive Services* are automatically generated services which are triggered by special events. *Evolutionary Services* are services which are updated and enhanced successively by continuous analysis and evaluation. The main processes of an usage cycle are:

- **Detection of the Situation Determinants:** The *Mobile Network Operator* detects the situation determinants. Objects of the detection are position, time and user identity.

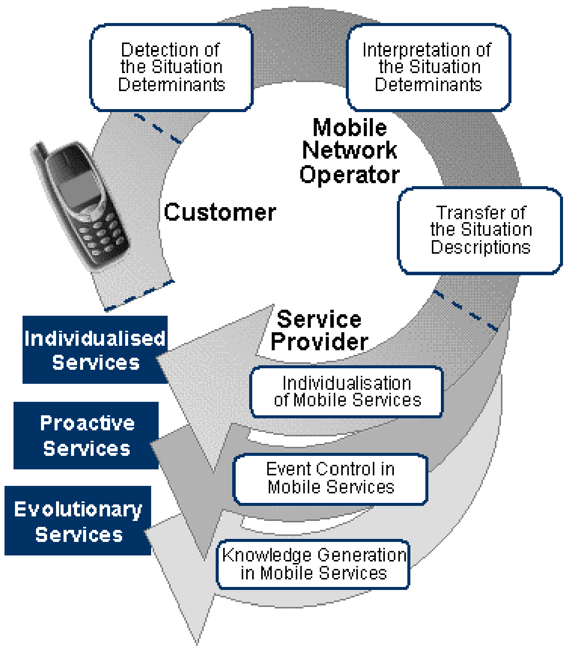

Detection of the Situation Determinants
Interpretation of the Situation Determinants
Mobile Network Operator
Customer
Transfer of the Situation Descriptions
Service Provider
Individualised Services
Individualisation of Mobile Services
Proactive Services
Event Control in Mobile Services
Evolutionary Services
Knowledge Generation in Mobile Services

Figure 1 - Usage Cyle for *Situation Dependent Mobile Services*

- **Interpretation of the Situation Determinants:** The *Mobile Network Operator* enriches the information by consulting additional information sources.
- **Transfer of the Situation Descriptions:** The *Mobile Network Operator* encodes the situation description and transfers it to the service provider. To ensure the privacy, personal information is removed.
- **Individualisation of Mobile Services:** The service provider uses the situation description for the individualisation of user initiated services (pull services). The individualisation of mobile services is a tool for customer orientation and the manageability of services.
- **Event Control in Mobile Services:** The service provider can define situation based rules. The *Mobile Network Operator* compares these rules with the situations. If a rule matches a situation, a proactive service will be generated (push service). A great potential of mobile services is the ubiquitous addressability of customers which is founded in the close interconnection of customer and personal mobile device. This allows services to get activated or initiated by a particular circumstance and enables active notification services. Regarding the legal aspects and Godin's permission marketing concept (1999), a complete new dimension of services for customers and service providers is conceivable.
- **Knowledge Generation in Mobile Services:** Knowledge generation for mobile services makes a long-term analysis, evaluation and extension of services possible. A service provider may use the historical data about customer transactions and the respective user's situation as valuable sources for an evaluation of his mobile services. Thus he can conclude the demographic properties, the regional allocation or many other attributes that help to enhance or upgrade a service. Additional tools may further help the service provider to better understand the intentions, purposes and the special needs of users in special situations. An evaluation of services by the customer may help to identify wrong adaptations. Depending on the success and the influencing factors, a service can be stopped or advanced in an evolutionary style.

4 Analysis of the End-User's Acceptance

Service providers can use an acceptance model to understand the reasons for the user's acceptance of existing mobile services ex post or to adapt the requirements for the service development. The characteristics of this acceptance model should specifically focus on situation dependent mobile services and the usability as a permanent controlling instrument for the iteractive adaptation of services to the user's requirements. The structure of the suggested acceptance model (Amberg et al. 2003,1) is based on the principal idea of the ***Balanced Scorecard*** (BSC) (Kaplan, Norton, 1996). Accordingly, the acceptance model uses a balanced set of individually measurable acceptance criterions for the analysis and the evaluation of the end-user's acceptance. The balanced consideration of the criterions that are used for measuring the user's acceptance, leads to a more sophisticated evaluation.
We use the complementary categories benefits/costs and service/general conditions to structure the acceptance model. The distinction between service specific acceptance and general acceptance factors (general conditions of services) is derived from the model of Herrmann (1999). There are four dimensions, that can be distinguished: **P*erceived Usefulness***, ***Perceived Ease of Use***, ***Mobility*** and ***Costs***. The first two dimensions *Perceived Usefulness* and *Perceived Ease of Use* are taken from the ***Technology Acceptance Model*** TAM (Davis, 1989). The *Perceived Usefulness* is an additional incentive to use a service. In opposition to this, the *Ease Of Use* is an effort, which is an obstruction for the *Usage* of a service. Both dimensions describe the service specific influencing factors of the acceptance of a service. The acceptance model extends the TAM approach with two additional dimensions: *Mobility* and *Costs*. To regard the influencing factors in more detail, a further refinement of the dimensions is recommended. According to Kollmann (1998), the subdivision in ***First Use*** and (regular) ***Usage*** is reasonable. The *First Use* is a kind of barrier for the *Usage* of a service. Both are necessary for a balanced consideration of the user's acceptance of mobile services. In addition to this, it is possible to

subdivide these eight criterions further on (e.g. in emotional/rational or qualitative/quantitative) (Amberg et al. 2003,2).

5 Designing Cooperative Development

An interaction model describes the service and information relationships between the involved participants. From a conceptual perspective of providing situation dependent mobile services, three or four market participants can be differentiated. Information products are offered by the **service provider**, procured by the ***Mobile Network Operator*** (MNO) and paid by the **customer**. For physical products a **logistic provider** is involved for the physical transportation between service provider and customer.
In the scope of the proposed interaction model (Amberg et al. 2002,2), the *Mobile Network Operator* takes a major role as an **intermediate** between service provider, customer and if necessary logistic provider. From the customer's view he is the contact for all customer specific concerns. He ensures the access to the mobile network, manages the personal settings and profiles (e.g. privacy protection), receives and processes the user requests, transmits the information products and is responsible for billing. From the service provider's view he provides a widespread service platform, which enables him to offer any service to the customer. The resulting central role of the *Mobile Network Operator* is obvious. Consequently, aspects like protection of privacy or data security have to meet high demands. Considering the security aspects, the *Mobile Network Operator* has to establish himself as a trustable party, commonly termed as ***Trusted Third Party*** (TTP). The authors consider emotional barriers to be very important. Concepts to ensure and guarantee trustability are an important field of research.
The *Mobile Network Operator* is the only involved party, which has the infrastructure to measure the situation determinants. This is an essential reason for being the only one who can handle the interpretation and description of situations efficiently. The strict borders of data protection and legal regulations (Enzmann, et al. 2000) on the one hand and the sensibility of customers regarding their personal data on the other hand determine that the *Mobile Network Operator* should only transfer anonymous situation descriptions. Most information products provided over the platform of the *Mobile Network Operator* do not depend significantly on the user's identity. An implementable concept for ensuring the privacy is using alias or session-IDs instead of a personal ID.

6 Realization of Situation Dependent Mobile Services

A convenient system architecture focuses on the implementation of the cooperation platform. The situation description that is conveyed from the *Mobile Network Operator* to the service provider normally contains a reference to the identity of the user. The type of reference depends on the degree of intensity that characterises the relationship between the mobile customer and the service provider (Amberg et al. 2002,2). The customer must have the choice to select the type of reference that he wants to transmit to the service provider:

- **Anonymity** (e.g. Session-ID): The service provider only gets a weak reference that points to the current data session of the customer. The customer-ID can not be resolved by the service provider.
- **Pseudonymity** (e.g. X-ID or Nickname): The service provider receives a pseudonym for the user that remains the same over all data sessions. Therefore, the service provider can recognise a mobile customer without knowing his identity.
- **Identity** (e.g. MSISDN): The service provider gets access to the technical address of the mobile terminal that enables him to resolve the customer's identity.

7 Outlook

One future task is the refinement of the existing facets, that were shortly introduced in this paper and the search for further aspects, that significantly affect *Situation Dependent Mobile Services*. For the further design of these services the development of the mobile commerce market is very important. According to the imminent global rollout of 3rd generation UMTS networks and later 4th generation applications, the identification and understanding of key success factors will play an important role.

Literature

Amberg, M.; Figge, S.; **Wehrmann, J.** (2002,1): Compass – Ein Kooperationsmodell für situationsabhängige mobile Dienste. In: Hampe, J. F.; Schwabe, G. (Hrsg.): Proceedings zur Teilkonferenz Mobile and Collaborative Business der Multikonferenz Wirtschaftsinformatik (MKWI 2002), p. 31-50, Nürnberg, Germany.

Amberg, M.; Figge, S.; **Wehrmann, J.** (2002,2): A Cooperation Model for Personalized and Situation Dependent Services in Mobile Networks: Proceedings of the International Workshop Conceptual Modeling Approaches to Mobile Information System Development at the 21st International Conference on Conceptual Modeling Proceedings (ER 2002), p. 13-24, Tampere, Finland.

Amberg, M.; Hirschmeier, M.; **Wehrmann, J.** (2003,1): Ein Modell zur Akzeptanzanalyse für die Entwicklung situationsabhängiger mobiler Dienste im Compass Ansatz: Proceedings of the 3rd Workshop on Mobile Commerce (MC3), Augsburg, Germany. (Accepted)

Amberg, M.; Hirschmeier, M.; **Wehrmann, J.** (2003,2): The Compass Acceptance Model for the Analysis and Evaluation of Mobile Information Systems: International Journal for Mobile Communications (IJMC), Finland. (Submitted)

Amberg, M.; Okujava, S.; **Wehrmann, J.** (2003,3): Ein Komponenten-Framework für die situationsabhängige Adaption Web-Service-basierter Standardsoftware: Proceedings of the 5th Workshop Komponentenorientierter Betrieblicher Anwendungssysteme (WKBA5), Augsburg, Germany. (Accepted)

Amberg, M.; **Wehrmann, J.** (2002): A Framework for the Classification of Situation Dependent Services: Proceedings of the Eighth Americas Conference on Information Systems (AMCIS 2002), p. 1838-1843, Dallas, USA.

Davis, F. D. (1989): Perceived Usefulness, Perceived Ease of Use, and User’s acceptance of Information Technology, MIS Quarterly, 13:3 (8/1989), p. 319-341.

Enzmann, M; Pagnia, H.; Grimm, R. (2000): Das Teledienstedatenschutzgesetz und seine Umsetzung in der Praxis. In: Koenig, W. (Hrsg.) Wirtschaftsinformatik 42, (5/2000), p. 402-412. Wiesbaden, Germany.

Godin, S.(1999): Permission Marketing: München, Germany.

Herrmann, T. (1999): Perspektiven der Medienwirtschaft. Kompetenz – Akzeptanz – Geschäftsfelder. In: Szyperski, N. (Hrsg.): Perspektiven der Medienwirtschaft. Köln, Germany.

Kaplan, R.S.; Norton, D.P. (1996): The Balanced Scorecard – Translating Strategy into Action. Boston, USA.

Kollmann, T. (1998): Akzeptanz innovativer Nutzungsgüter und -systeme : Konsequenzen für die Einführung von Telekommunikations- und Multimediasystemen. Wiesbaden, Germany.Biographical Notes

Michael Amberg, Jens Wehrmann

Friedrich-Alexander-University of Erlangen-Nuremberg, Department of Economics,

Chair for Business Information Systems III, Lange Gasse 20, 90403 Nuremberg, Germany,

email: amberg@wiso.uni-erlangen.de, jens.wehrmann@wiso.uni-erlangen.de

Michael Amberg

Professor Dr. Michael Amberg, born 1961, is the chair holder of the Business Information Systems III department at the University of Erlangen-Nuremberg. Previously he headed the chair for Business Information Systems at the technical university of Aachen (RWTH). The focus of his work is the development and management of information technology (IT Management). The research interests are the development and management of complex software and hardware components and their integration into industrial processes. Further research topics cover multimedia, mobile information systems as well as embedded intelligent systems.

Jens Wehrmann

Jens Wehrmann, born 1975, studied electrical engineering and business economics at the technical university of Aachen (RWTH). He is working at the chair for Business Information Systems III since the foundation in 2001. He is the manager of a research project to build an adaptive component based standard software. His main research topics are situation depended applications, development of adaptive software, mobile application management, mobile commerce and mobile architectures.

Personal Wizards: collaborative end-user programming

Lawrence Bergman, Tessa Lau, Vittorio Castelli, and Daniel Oblinger
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
+1 914 784 7946
bergmanl@us.ibm.com

INTRODUCTION

Users of computing systems follow procedures to accomplish their goals. In some cases, where procedures are dictated by an organization's business process, users must follow a prescribed sequence of steps to accomplish tasks such as requesting travel reimbursement, procuring a new workstation, or managing payroll for their employees. Systems management administrators follow a different set of best practice procedures for tasks such as configuring and optimizing an organization's email systems, and diagnosing and repairing network problems. End users accumulate their own personal collections of procedures for accomplishing their own goals, such as monitoring stock portfolios or collecting information for a presentation.

Despite the ubiquity of these procedures, however, current systems provide little support for documenting or capturing procedural knowledge. Printed manuals are expensive to produce, and are often inadequate: it is hard to find what you want, and difficult to use what you find. Infrequent users of a procedure are often forced to make notes on paper in order to remind themselves of the right sequence of steps to complete a task. Yet no matter how often a user completes a task, the system always requires the user to perform the same rote actions over and over again. For widespread business processes, the local IT department may write software to automate these procedures, but these programs are typically brittle and costly to upgrade when the underlying process changes.

The vision of the Personal Wizards project is to dynamically capture corporate and personal procedures through cross-application programming by demonstration [1,3]. The Personal Wizards system observes experts' keystrokes and mouse actions as they perform a procedure on the desktop. Experts may annotate the procedure at key points, associating appropriate text with certain steps in the procedure. The system then produces a Personal Wizard that can guide a new user through a similar task, presenting the right information at the right time, and automating particularly repetitious steps in the procedure.

In designing the system, we are guided by the following desiderata:

- Lowering the barriers to authoring procedural knowledge
- Learning from multiple experts
- Collaborative exploration of all possible paths through a procedure
- Creation of robust procedures with branches and failure recovery
- Human-understandable procedures

Collaborative authoring is a central concept in our work. Our vision is that many procedures will be created by gathering information from a variety of experts. This has several advantages:

- The procedure captures execution under a variety of system configurations and working environments.
- The procedure captures variation in the ways in which a task can be performed.
- The collaborative process facilitates identification of operations that are relevant to a procedure, and those that are peculiar to a user (such as frequent breaks to read e-mail), without a need for manual labeling.

This paper presents our work on the Personal Wizards project and situates it within the context of end-user programming. We begin by outlining a number of concrete scenarios where Personal Wizards could be applied.

USAGE SCENARIOS

In this section we present a set of scenarios intended to illustrate some of the different ways in which the Personal Wizards (PW) system might be used. Although one of the goals of Personal Wizards (and end-user programming in general) is to blur or remove the distinctions between authors and consumers of procedures, we note that some users of this system will be "expert users". In fact, some users will be explicitly charged with developing procedures for use by others. For this reason, in the following set of scenarios, we will use the term "expert" and "novice" to distinguish these roles, pointing out where the two roles may be interchangeable or merge.

Scenario 1: Technical support

In this scenario, we consider development and deployment of procedures for troubleshooting failures in a technical support environment. Desk-side support personnel will be responsible for authoring of procedures, for example a procedure that troubleshoots installation of a network card. The support staff may train the PW system by deploying a PW client on the desktop of a caller and "driving" it remotely. During the troubleshooting session, or at a later time, the support person may add annotations to the procedure.

Once trained on a number of service calls, troubleshooting procedures can be deployed electronically by support staff, or made available automatically through websites. This allows end-users to invoke the PW procedure in a novice role, gaining access to an electronic troubleshooting assistant. The Wizard will prompt for input as needed, and guide the novice through the troubleshooting procedure. The novice can choose to execute in a step-wise fashion, or to allow the procedure to execute on its own, only pausing when user-input is required.

We note that there will always be cases where the Wizard will run into previously unencountered configurations. In such cases, PW will notify the novice and suggest consultation with user support. If the novice is able to complete the procedure on their own, they will be encouraged to submit their execution trace back to the helpdesk, allowing the troubleshooting procedure to evolve. In this case, the novice dynamically assumes the role of expert.

Scenario 2: Software Debugging

Debugging a piece of software often involves elaborate procedures that include: supplying the application with the required inputs, examining and capturing key state variables, and exercising control over the procedure – stepping in and out of functions, setting and removing breakpoints, etc. Often the same debugging procedure is employed multiple times with only small variations in each execution. We envision PW assisting with such repetitive debugging tasks. In this case, the programmer assumes the roles of both expert and novice.

Scenario 3: Desktop procedures

Use of computers in both home and business environments involves a wide variety of repetitive tasks. These include tasks such filing expense accounts, ordering supplies, and reorganizing address books. In these scenarios, the same user may play both expert and novice, first recording a procedure then employing it again in the future to automate or replay a similar task.

Scenario 5: Live Tutorials

Tutorials are traditionally document-based walkthroughs of the set of steps required to accomplish a particular goal. Recently, video-style tutorials, which show the sequence, often by highlighting the controls of the application itself, have become popular. These tutorials are typically hand-scripted.

We envision the PW system being used to provide “live” tutorials – showing the novice how a procedure is executed by guiding him through the actual performance of the task. PW enables rapid authoring of such tutorials; the author simply demonstrates the procedure, supplying annotations as desired. Furthermore, the tutorial readily adapts to changes in the underlying procedure, since new demonstrations are far easier to perform than manually adjusting a script.

USER INTERFACE CHALLENGES

Several key issues must be addressed in the design of the PW user interface. These include supporting mixed-initiative control during training and execution of procedures, providing support for annotation during authoring, providing cues to the novice that indicate the outcomes and implications of various pathways through a PW procedure, and providing means for debugging a procedure when it fails.

Control

The PW interface must provide for flexible changes of role – at times the user may be authoring a procedure in the role of expert, at other times using it as a novice. We want to support the user in changing roles with as little overhead as possible, allowing them to alternate between roles as teacher and student. By observing when a novice changes parameters, or deviates from the default procedure, and adjusting the procedure accordingly, PW supports a continuous authoring process, in which the procedure evolves over time as it is used in new situations. Furthermore, we will provide mechanisms for the user to specify whether updates are local, or can be contributed to procedures in a wider community.

Annotation

One way in which expert knowledge is easily communicated from the expert to the novice is through annotations. In the PW system, the expert provides annotation associated with individual execution steps or sets of steps as liberally as she deems appropriate. On playback, the novice is presented with annotations, which may indicate not only the required actions or inputs, but also can provide information on the sub-procedure being performed as well as the rationale for proceeding along particular paths within a procedure.

Visualization

The ability for PW to communicate to the novice possible actions that the procedure might take is critical. This is the problem of procedure visualization. We anticipate exploring a variety of procedure visualization alternatives.

Two main strategies for visualizing procedures make use of spatial and of temporal sequencing of information. Spatial arrangements include collapsible hierarchies and comic strips (see [1] for an example of the latter). We will explore extensions to each. Providing for expert manipulation of hierarchical procedure/sub-procedure structures may be an effective way for an expert to impart knowledge of procedure structure, and for the novice to examine that information. Comic strips have the shortcoming of showing a linear path through a procedure. We will extend this metaphor to show branch points and permit the novice to interactively “scroll” through alternatives at any branch point.

Temporal displays show a sequence of actions in time. Procedure visualization can begin by highlighting the control(s) to be activated at the current step. The user will be able to examine the consequences of particular actions by selecting the associated highlighted UI element in "examine mode" and then "stepping through" the procedure. This gives the novice the capability of "looking ahead" in the procedure without actually executing it.

Debugging

Another important consideration in designing the PW interface is providing methods for recovery and repair when the procedure "goes wrong". One possible approach is to provide an "undo" for any operation. The novice is much more likely to let PW "have its head" if she knows that any incorrect actions can be easily undone.

RESEARCH CHALLENGES

This section outlines the technical research challenges we have identified in learning end-user programs by demonstration. Our general view is to approach programming by demonstration as a machine learning problem: acquiring generalized programs based on traces of those programs' execution behavior. The general problem can be described in terms of several sub-problems.

Given a trace, the first problem is to *segment* the trace in order to identify procedure and sub-procedure boundaries. Traces must then be *generalized* in order to determine the user's intent in performing each of the concrete user interface actions in the trace. Given several traces, the next challenge is to simultaneously *align* portions of the traces such that subsequences of similar functionality are paired together. Underlying the alignment and generalization process is a specific *procedure representation* that captures the meaning of the procedure. Finally, a *retrieval* process is needed for a user to index and locate a procedure in the knowledge base that will assist in a particular task. The following subsections briefly describe each of these research challenges in turn.

Segmentation

The first research challenge is to segment the traces into procedures and sub-procedures. In previous programming by demonstration systems [5], the user manually indicates the start and end of each demonstration. However, this may prove to be too much of a burden for some users, who may not realize that they have begun executing a repetitive task until partway into the procedure. In addition, more complex procedures are logically broken down into sub-tasks, some of which may be common across multiple procedures. For example, a procedure for diagnosing email problems may include a sub-procedure for checking whether the workstation is able to connect to the network. Manually indicating the boundaries of each of these subtasks is certainly going to require too much user effort. Thus one research goal in our work is to consider automated approaches to the segmentation problem.

Generalization

Generalization is the process of inferring a user's intent from a concrete trace. For instance, if the user clicks the mouse button, she may be following a link in a web browser, launching an application, or invoking a button. In a subsequent demonstration of the same task, the mouse click may occur at different coordinates, or the user may use a keyboard equivalent to perform the same function. Generalization of the two different actions (which both have the same underlying intent) identifies the similarity between the two actions. A generalized procedure is less sensitive to the exact configuration and layout of a user's machine, and recognizes different possible ways to accomplish the same goal.

Our approach to generalization is based on version space algebra [2], a framework for efficiently enumerating the space of possible generalizations for concrete actions, and maintaining the set of consistent generalizations given one or more examples of the target action.

Alignment

Our goal is to learn robust procedures from traces generated by different experts or under different conditions. In these cases, traces may contain steps in different order, or traces in which a whole sequence of steps is missing (perhaps because those steps are not applicable on a particular system). The alignment problem is to recognize and align together subsequences of similar functionality across multiple traces. In our work, the similarity metric is based on generalization; two actions are similar if they share a common generalization. Our approach is based on an extension to hidden Markov models [4], which provide a mechanism for considering all possible alignments and iteratively selecting the locally best alignment.

Procedure representation

One challenge in programming by demonstration is to identify a sufficiently expressive yet tractable representation of the procedure that will support the user interface we wish to display to the user. Our work relies on a representation of procedures as collections of executable actions. Each action is modeled as a function that maps from the state of the system to a new state in which some action has been performed. These actions are joined together into a procedure using a probabilistic finite state machine representation. Each procedure execution is a path through this graph, and the choices at each node in the graph represent decisions made based on the information visible on the user's screen at that time. For example, a procedure that specifies different actions to perform depending on whether a previous step failed or succeeded will examine the visible state (including the exit condition of the previous step) in order to decide which steps to follow next.

Retrieval

Learned procedures are only useful insofar as the user can retrieve them again when they are needed. Our approach to solving this research challenge involves the construction of an indexed knowledge base of procedures. Procedures can be indexed based on criteria such as the applications or application screens used within each procedure, their length, the time at which they were created or used (imagine procedures for calculating income tax, which tend to occur in early April), and keywords extracted from the commands and user interface components involved in the procedure.

Procedural dissemination can also be either implicit or explicit. A novice can simply have PW running on their machine at all times. At any point, the user can ask for assistance and PW can examine the recently recorded set of actions as the basis of a query, and retrieve procedures that begin with similar steps. Alternately, PW can signal the user that a procedure is available for a particular task based on matching novice actions to the procedure library (of course, great care must be taken to keep "suggestions" from being obtrusive or annoying).

Explicit dissemination could be either based on either a pull or a push model. A novice would pull a procedure from a repository by formulating a request, perhaps in terms of a set of goal keywords. Push-based dissemination would include desk-side support emailing a procedure to a novice to accomplish a particular task, such as a mandatory upgrade.

EVALUATION STRATEGIES

A key question in end-user programming is how to evaluate a system designed to assist end-users in their tasks. We propose several metrics, each designed to evaluate a different aspect of the system.

Learning efficiency: how much training does the system require to learn a procedure that can accurately predict the steps required to complete the task in a new situation?

Effort savings: how much effort does the system save a user in a given task situation?

Usability: what kind of training is required to author procedures in the system? Can regular end-users author and consume procedure nuggets?

CONCLUSION

We have presented the Personal Wizards project, an end-user programming system that acquires procedural knowledge by observing experts perform tasks directly in a user interface. We have outlined scenarios for which end-user programming would be useful, described a user interface for end user programming, characterized the research problems involved in creating such a system, and proposed strategies for evaluating the result.

REFERENCES

1. David Kurlander and Steven Feiner. A History-Based Macro by Example System. *In Proceedings of UIST'92,* pp. 99--106, 1992.

2. Tessa Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pp. 527-534, June 2000.

3. H. Lieberman, ed. Your Wish is My Command: Giving Users the Power to Instruct their Software. Morgan Kaufmann, 2001.

4. Lawrence R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257-285, February 1989.

5. Steven A. Wolfman, Tessa Lau, Pedro Domingos, and Daniel S. Weld. Collaborative Interfaces for Learning Tasks: SMARTedit Talks Back. In *Proceedings of the 2001 Conference on Intelligent User Interfaces*, 20

Biographical Notes

Lawrence Bergman has been a Research Staff Member at IBM T.J. Watson Research Center since 1994. His interests are in tools and environments for design, query, and application development, with a particular emphasis on making computer systems more accessible both to experts and novices. Currently he is involved in creating the Personal Wizards system, a desktop programming-by-demonstration environment. From 2000-2002, he worked on developing environments and techniques for model-based authoring of applications for deployment to multiple mobile platforms. From 1994-2000, he worked on developing a novel interactive query interface to multimedia databases, used for applications in forestry, petroleum exploration, space science, and epidemiology. He received his PhD in 1993 from the University of North Carolina at Chapel Hill, where he developed a visualization environment demonstrated to provide flexible movement between user and programmer roles. He is co-editor of the book "Image Databases", and has been a member of the program committee for several image retrieval and user-interface conferences.

Tessa Lau is a Research Staff Member at IBM's T.J. Watson Research Center. Her interests are in improving the usability and functionality of end-user interfaces through the application of innovative AI techniques. She completed her PhD at the University of Washington in 2001, where she developed a machine learning framework for programming by demonstration, and produced the SMARTedit system that automated repetitive text-editing tasks by example. Since joining IBM Research she has played an active role in defining and building the Personal Wizards system, drawing on prior experience in the field to further the vision of bringing end-user programming to the desktop

Software Engineering for End-User Programmers

Margaret Burnett, Gregg Rothermel, and Curtis Cook
Department of Computer Science
Oregon State University
Corvallis, OR 97331
(541)737-3273
{burnett, grother, cook}@cs.orst.edu

INTRODUCTION

There has been considerable work in empowering end users to be able to write their own programs, and as a result, end users are indeed doing so. In fact, the number of end-user programmers is expected to reach 55 million by 2005 in the U.S. alone [2], writing programs using such devices as special-purpose scripting languages, multimedia and web authoring languages, and spreadsheets. Unfortunately, evidence from the spreadsheet paradigm, the most widely used of the end-user programming languages, abounds that end-user programmers are extremely prone to errors [15]. This problem is serious, because although some end users' programs are simply explorations and scratch pad calculations, others can be quite important to their personal or business livelihood, such as for calculating income taxes, e-commerce web pages, and financial forecasting.

We would like to help reduce the error rate in the end-user programs that are important to the user. Although classical software engineering methodologies are not a panacea, there are several that are known to help reduce programming errors, and it would be useful to incorporate some of those successes in end-user programming. Toward this end, we have been working on a vision we call *end-user software engineering*, a holistic approach to the facets of software development in which end users engage. Its goal is to bring some of the gains from the software engineering community to end-user programming environments, *without* requiring training or even interest in traditional software engineering techniques.

RESEARCH UPON WHICH END-USER SOFTWARE ENGINEERING BUILDS

Our research into end-user software engineering draws from previous research in three areas: HCI, programming languages, and software engineering.

Programming is a collection of problem-solving activities, and our goal is to help end users in these activities. Hence, we draw heavily on HCI research about human problem-solving needs. The HCI research with the greatest influence on our work so far has been Blackwell's theory of Attention Investment [1], Green et al.'s work on Cognitive Dimensions [7], Pane et al.'s empirical work [14], and psychologists' findings about how curiosity relates to problem-solving behavior [12]. Other strong influences have come from the extensive work on end-user and visual programming languages (e.g., [11, 13, 18]), and the software engineering research community's work regarding testing, assertions, and fault localization (e.g., [8, 10, 16, 19]).

COMPONENTS OF END-USER SOFTWARE ENGINEERING

End-user software engineering is a highly integrated and incremental concept of software engineering support for end users. Hence, its components are not individual tools, each with a button that can be separately invoked, but rather a blend of knowledge sources that come together seamlessly. A continually evolving prototype of the end-user software engineering concept exists for Forms/3 [5], a member of the spreadsheet paradigm. The components we have so far blended into Forms/3 are briefly summarized in this section.

WYSIWYT Testing

One of the components is the "What You See Is What You Test" (WYSIWYT) methodology for testing [20, 21, 3]. WYSIWYT allows users to incrementally edit, test and debug their formulas as their programs evolve, visually calling users' attention to untested cells by painting their cell borders red (light gray in this paper). Tested cells are painted blue (black), and partial testedness is depicted in purples (grays) along the continuum from red to blue (light gray to black). For example, in Figure 1, cell WDay_Hrs is not tested. Whenever the user notices that a cell's value is correct, he or she checks it off in the checkbox in its corner, increasing the testedness. Empirical work has shown that the WYSIWYT methodology is helpful to users [9] but, even with additional visual devices such as colored arrows between formula subexpressions to indicate the relationships remaining to be covered, after doing a certain amount of testing, users sometimes find it difficult to think of suitable test values that will cover the as-yet-untested relationships. At this point, they can invoke a "Help Me Test" feature.

Help Me Test

The "Help Me Test" (HMT) feature [6] suggests test values for user-selected cells or user-selected dataflow arrows. HMT then tries to find inputs that will lead to coverage of an untested portion of the user's selections, or of any cell in the program if the user does not have any cells or arrows selected. HMT cannot always find a suitable test case, but in performance experiments it has succeeded in less than 4 seconds approximately 90% of the time [6]. There is also a Stop button available, if HMT is deemed as taking too long.

HMT's efforts to find suitable test values are somewhat transparent to the user—that is, they can see the values it is considering spinning by. The transparency of its behavior turns out to contribute to the understandability of both HMT and assertions.

Assertions

There is an assertion feature [4, 22] in the environment. (Our system terms these "guards" when communicating with users, so named because they guard the correctness of the cells.) Assertions protect cells from "bad" values, i.e., from values that disagree with the assertion(s). Whenever a user enters an assertion (a *user-entered assertion*) it is propagated through formulas creating *computer-generated assertions* on downstream cells. The user can use tabs (not shown) to pop up the assertions, as has been done on all cells in Figure 1. The stick figure icons on cells Monday, Tuesday, ... identify the user-entered assertions. The computer icon, on cell WDay_Hrs, identifies a computer-generated assertion, which the system generated by propagating the assertions from Monday, Tuesday, …, through WDay_Hrs's formula. A cell with both a computer-generated and user-entered assertion is in a conflict state (has an *assertion conflict*) if the two assertions do not match exactly. The system communicates an assertion conflict by circling the conflicting assertions in red. In Figure 1 the conflict on WDay_Hrs is due to an error in the formula (there is an extra Tuesday). Since the cell's value in WDay_Hrs is inconsistent with the assertions on that cell (termed a *value violation*), the value is also circled.

Users have two concrete syntaxes for entering assertions onto a cell: one textual, and one primarily graphical. Examples of the textual syntax are in Figure 1. Or-assertions are represented with comma separators on the same line (not shown), while and-assertions are represented as multiple assertions stacked up on the same cell, as with cell WDay_Hrs. (And-assertions must always agree.) It is possible to omit either endpoint from a range, allowing for relationships such as <, <=, and so on. Further information on the relationships supported, the assertions' relative power, and the graphical syntax can be found in [4]. Also reported in [4] is an empirical study, which resulted in participants using provided assertions being significantly more effective at debugging than were participants without access to assertions.

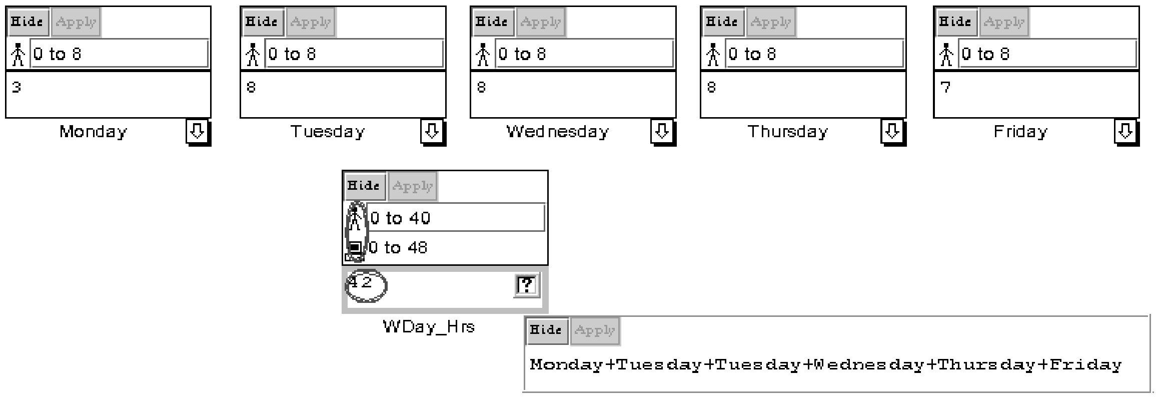
HMT Assertions

To entice users to consider entering assertions, we use a strategy based on findings about the psychology of curiosity [12]. We term our strategy the *Surprise-Explain-Reward* strategy [22]. The first step of our strategy is to generate a meaningful surprise for the user. That is, the system needs to violate the user's assumptions about their program. We have devised a pseudo-assertion for this purpose, termed an *HMT assertion* because it is produced by HMT. An HMT assertion is a guess at a possible assertion for a particular cell.

HMT assertions exist to surprise and thereby to create curiosity. Consider Figure 1 and Figure 2, which are part of a weekly payroll program. The user may expect values for Monday to range from 0 to 8, and rightly so, because employees cannot be credited with fewer than 0 or more than 8 hours per day. Since HMT was not aware of this, it attempted inputs less than zero. Thus, the HMT assertion for Monday probably violates the user's assumptions about the correct values for Monday.

Once an HMT assertion has been generated, it behaves as any assertion does. Not only does it propagate, but if a value arrives that violates it, the value is circled in red.

It is important to note that, although our strategy rests on surprise, it does not attempt to rearrange the user's work priorities by requiring users to do anything about the surprises. No dialog boxes are presented and there are no modes. HMT assertions are a passive feedback system; they



Monday+Tuesday+Tuesday+Wednesday+Thursday+Friday

Figure 1: The Forms/3 environment. Cell formulas can be displayed via the tab at the lower right hand side of the cell, as has been done in WDay_Hrs. Additionally, cells with non-constant formulas have borders colored depicting "testedness" and a check box, as shown with a "?" in it, in the right hand corner of the WDay_Hrs cell.

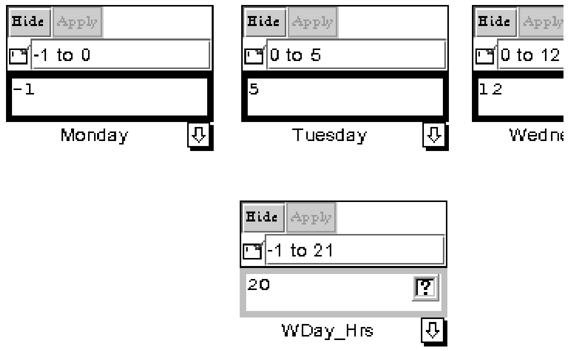


Figure 2: HMT has guessed assertions for the input cells (top row). (Since HMT changed the values of the input cells, they are highlighted with a thicker border.) The guesses then propagated through WDay_Hrs's formula to create an HMT assertion for that cell as well.

try to win user attention but do not require it. If users choose to follow up, they can mouse over the assertions to receive an explanation, which explicitly mentions the rewards for pursuing assertions. More information about the surprises, explanations, and rewards will be presented at CHI'03 [22].

Fault Localization

Given the explicit, visualization-based support for WYSIWYT testing, it is natural to consider leveraging it to help users with fault localization once one of their tests reveals a failure. But what is the best way to proceed? We began with a particular approach [17], but have since devised two other approaches for reasoning about where the faulty cells might lie. We are currently conducting a variety of empirical work to see which is the most effective, given the testing end-user programmers actually do.

CONCLUDING REMARKS

Most researchers working on end-user programming are working on exactly that—programming. Our view is that giving end-user programmers ways to easily create their own programs is important, but is not enough. We believe that, like their counterparts in the world of professional programming, end-user programmers need support for other aspects of the software lifecycle. Supporting software development activities beyond the programming stage—without requiring end users to invest in software engineering training—is the essence of the end-user software engineering vision.

ACKNOWLEDGMENTS

This work was supported in part by NSF under ITR-0082265.

REFERENCES

1. Blackwell, A. and Green, T. R. G. Investment of attention as an analytic approach to cognitive dimensions. In T. Green, R. Abdullah & P. Brna (Eds.) *Collected Papers Wkshp. Psych. of Programming Interest Grp.*, (1999), 24-35.
2. Boehm, B., Abts, C., A. Brown, W., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, J., and Steece, B. *Software Cost Estimation with COCOMO II.* Prentice Hall PTR, Upper Saddle River, NJ, 2000.
3. Burnett, M., Sheretov, A., Ren, B., and Rothermel, G. Testing homogeneous spreadsheet grids with the 'What You See Is What You Test' methodology, *IEEE Trans. Software Engineering*, (May 2002), 576-594.
4. Burnett, M., Cook, C., Pendse, O., Rothermel, G., and Summet, J., and Wallace, C. End-user software engineering with assertions in the spreadsheet paradigm, *Int'l. Conf. Software Engineering* (Portland, OR, May 2003), to appear.
5. Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J., Yang, S. Forms/3: a first-order visual language to explore the boundaries of the spreadsheet paradigm. *J. Functional Programming 11*, 2 (March 2001) 155-206.
6. Fisher, M., Cao, M., Rothermel, G., Cook, C., Burnett, M. Automated test generation for spreadsheets, *Int. Conf. Software Engineering*, (Orlando FL, May 2002), IEEE 141-151.
7. Green, T. R. G. and Petre, M. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. Visual Languages and Computing 7*, 2 (June 1996), 131-174.
8. Jones, J.A., Harrold, M.J., and Stasko, J. Visualization of test information to assist fault localization. *Int. Conf. Software Engineering*, (May 2002), 467-477.
9. Krishna, V., Cook, C., Keller, D., Cantrell, J., Wallace, C., Burnett, M., and Rothermel, G. Incorporating incremental validation and impact analysis into spreadsheet maintenance: an empirical study, in *Proc. Software Maintenance*, (Florence Italy, Nov. 2001), IEEE, 72-81.
10. Laski, J. and Korel, B. A data flow oriented program testing strategy. *IEEE Trans. Soft. Eng. 9*, 3 (May 1993), 347-354.
11. Lieberman, H. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, San Francisco, 2001.
12. Lowenstein, G. The psychology of curiosity. *Psychological Bulletin 116*, 1 (1994), 75-98.
13. Nardi, B. *A Small Matter of Programming: Perspectives on End-User Computing*, MIT Press, Cambridge, MA, 1993.
14. Pane, J., Myers, B., and Miller, L. Using HCI Techniques to Design a More Useable Programming System, in *Proceedings Human-Centric Computing 2002* (Arlington VA, September 2002), 198-206.
15. Panko, R. What we know about spreadsheet errors. *J. End User Computing*, (Spring 1998).
16. Rapps, S., and Weyuker, E.J. Selected software test data using data flow information. *IEEE Trans. Soft.Eng. 11*, 4 (Apr. 1985), 367-375.
17. Reichwein, J., Rothermel, G., and Burnett, M. Slicing spreadsheets: An integrated methodology for spreadsheet testing and debugging. *2nd Conf. Domain Specific Languages,* (Oct. 1999), 25-38.
18. Repenning, A. and Ioannidou, A. Behavior processors: layers between end-users and Java virtual machines, *1997 IEEE Symposium on Visual Languages*, (Capri, Italy, September 1997), 402-409.
19. Rosenblum, D. A practical approach to programming with assertions. *IEEE Trans. Soft. Eng. 21*, 1 (Jan. 1995), 19-31.
20. Rothermel, G., Li, L., DuPuis, C. and Burnett, M. What

you see is what you test: a methodology for testing form-based visual programs, in *Proc. ICSE ‘98*, (Kyoto Japan, Apr. 1998), IEEE, 198-207.

21. Rothermel, G., Burnett, M., Li, L., DuPuis, C., and Sheretov, A. A methodology for testing spreadsheets, *ACM Trans. Software Engineering and Methodology 10*, 1 (Jan. 2001), 110-147.
22. Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M., and Rothermel, G., Harnessing curiosity to increase correctness in end-user programming, In *Proc. CHI ’03*, (Ft. Lauderdale, FL, April 2003), ACM Press, to appear.

End-User Programmers Need Improved Development Support

David A. Carr
Institutionen för Systemteknik
Luleå Tekniska Universitet
SE-971 87 Luleå, Sweden
+46 920 49 19 65
david@sm.luth.se

ABSTRACT

This paper argues that end-user programming systems need to support "professional" programming-language functions such as: program understanding, reuse, and automated error checking. We illustrate how these functions can be incorporated into an end-user programming system by describing the design of ReMIND+, an end-user programming system for modeling and optimizing industrial processes.

Keywords
End-user programming, visual programming

INTRODUCTION

End-user programming systems have been in common use since VisiCalc appeared. Research on end-user and novice programming has concentrated on individual problem solving and trying to understand the programming process. For example, Pane, et. al. [5] studied how non-programmers (primarily children) formulate solutions to programming problems when they had no programming language. The idea was to compare "natural" solution formulations to those used in current programming languages. Large differences between current language construction and novice descriptions were found, particularly in the areas of conditional clauses and looping constructs. This led Pane to suggest that end-user language designers should abandon current programming languages as a model. They should base their languages on novice perceptions of how looping and conditionals work in order to reduce the cognitive distance between the novice's mental model of the solution and its expression in a programming language.

Another trend has been the use of visual programming languages. Advocates of visual programming such as Shu [6] claim that it is the solution to the end-user programming problem. However, our own work [2] suggests that finding representations closer to the problem domain is more important. Another aspect that seems to have significant effects is how well the language supports human cognition. Indeed, the Cognitive Dimensions Framework [1] has been developed to assist visual language designers in evaluating how well their system supports human cognition.

Current commercial tools seem to assume that end-users construct programs individually. However, studies of actual users show otherwise. For example, spreadsheet users work in groups [3]. The group as a whole takes responsibility for debugging and revision. Group members also share spreadsheets, fragments, and techniques. Spreadsheets live over long periods of time. They are updated to reflect new practices. In many ways, end-user programs are no different than "professionally" developed programs. We believe that end-users would benefit from more careful attention to features that aid professional program developers such as program understanding, reuse, and automated error checking.

For example, spreadsheets use a model where the program elements (formulas and macros) are hidden by the data display. This model is the strength of the paradigm. However, it makes it extremely difficult for an end-user to understand the computation that is actually performed. In the domain of multimedia presenters, systems such as Macromedia Flash™ [7] divide the program into small fragments that are attached to objects and timeline instants. This makes construction of the movie (program) simpler, but at the same time it makes it nearly impossible to gain an overview of what the movie does and how it does it.

We are not arguing that the basic programming method and model of either system should change, but rather that extra functions are necessary to support the development of end-user programs as long-lived tools for business.

The rest of this paper discusses the support that we feel is necessary for end-user programming and describes reMIND+, a proposed design of an end-user programming system. reMIND+ will replace an existing application used by industrial engineers to model and optimize manufacturing processes.

SOFTWARE DEVELOPMENT SUPPORT

During the last 40 years programming languages and tools have evolved to support the professional software developer. We have learned that much of software development is maintenance. Maintenance is usually preformed long after the original coding by someone unfamiliar with the system. It has been clear for years that programming languages and methods must support reading to gain an understanding of the program. Software engineers have developed methods of design and documentation (e.g., UML) that provide an overview of a system and facilitate understanding how different components work together. Language constructs such as objects and functions also help program understanding by hiding details until they are necessary. End-user programming systems usually facilitate ease of programming by embedding small code fragments within an application's structure. They encourage incremental construction and leave documenting the "bigger picture" to the user. However, end-users by definition lack the training to do this. We feel that end-user systems need to be designed so that constructing the bigger picture is natural or that it can be automatically generated by the system.

Because system development is expensive, elaborate strategies for reusing fragments have evolved. These include object-oriented programming, object and function libraries, and message-based architectures. End-user systems generally lack these facilities. However, we do not feel that end-users will want the complication that most of these techniques bring. We believe that user extendable libraries would be useful and believe that systems should support simple ways to extract parts of an end-user system to be saved for reuse in a library.

A final area where professional programming languages are designed to reduce errors is in automatic consistency checking such as type systems. End-user programming systems usually avoid type schemes; however, type checking provides real benefit in reducing programming errors. We believe that where users provide type information, it should be used to check program reasonableness. For example, spreadsheet users often implicitly type spreadsheet cells by declaring an output format. A spreadsheet could easily check for conflicting types in a formula. Adding a percentage to a currency value is probably an error. Warning the end-user could well save debugging time or prevent the use of an incorrect result.

reMIND+, AN EXAMPLE

reMIND+ is a redesign of the reMIND system to convert it into an end-user programming system. The reMIND system is designed for industrial engineers who wish to model and optimize use of resources in industrial process. reMIND is itself a revision of an earlier system called MIND (Method for analysis of INDustrial energy systems). MIND [4] uses a model of resource flows into processes that use or transform the resources. The processes produce new resources that can be finished products or used in other processes. MIND used a text language to describe the flows and processes. The engineer also provided a set of linear constraints on the process model. These were then transformed into input for a commercial linear-system solver, CPLEX, and CPLEX provided an optimal solution for the system of linear equations and inequalities. A problem with MIND was that the text language made it difficult to trace errors with connecting flows and processes. So, the system was revised to add a graphical flow diagram editor and reMIND was born. (See Figure 1.) However, there are certain limits to reMIND that still make it difficult to use.

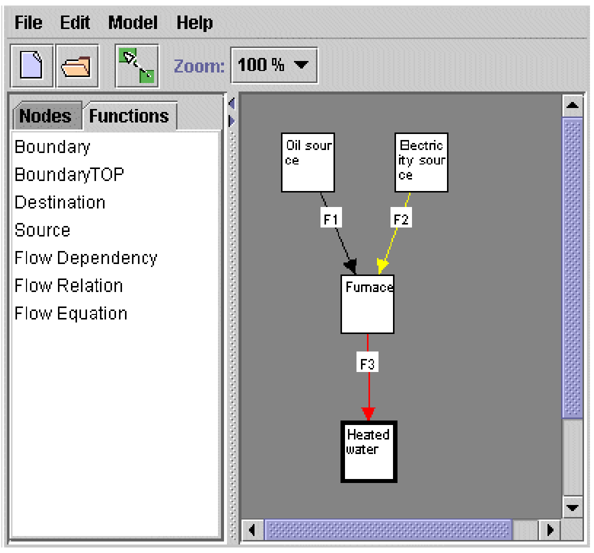


Figure 1: A reMIND flow diagram.

- Foremost is the design of the nodes. Nodes are of specific types and are defined by the Java code of the reMIND application. If a current node type does not fit user requirements, then a programmer must be hired to insert a new node type into the program.
- Another problem is that reMIND has a flat, monolithic structure. There is no easy way to reuse parts of models. Furthermore, a model's graphical representation rapidly exceeds the screen size, making it difficult to gain an overview.
- Finally, reMIND does not support any type checking on the flows. This can lead to mistakes were a flow representing energy is connected to an input representing a raw material such as iron ore.

The Design of reMIND+

In order to correct the above problems, we are redesigning reMIND and converting it to an end-user programming system. reMIND+ will retain the flow diagram of reMIND because the users find it much easier to use than MIND's text language. However, we will introduce the concept of a "submodel". A submodel will be a collection of processes (nodes) and flows (links between nodes) that will permit users to construct abstractions consisting of partial models.

We do not believe that end-users will want to consciously plan submodels. So, reMIND+ will allow them to extract them from existing models by selecting nodes. The system will then extract the selected nodes plus their incoming and outgoing flows. The flows will be traced back to their source and those with a common source merged. The entire submodel will then become a super node, which can be manipulated as a single entity. Submodels will also be added to a library for further reuse. Finally, reMIND+ will support semantic zooming on nodes in the flow diagram.

Thus, a submodel can be viewed as just a node or expanded to reveal its details.

The current multiple node types will be replaced with a single node. The end-user will define the incoming and outgoing flows when creating a node. The system will then produce a structured form similar to a spreadsheet with the incoming and outgoing flows already in cells. The form will provide cells for defining intermediate calculations and for specifying the transformation of incoming flows to outgoing flows. In this way, we hope to avoid the need to hire outside programmers each time a new node type is required. In addition, simple nodes can also be placed in a library.

MIND and its descendents model industrial processes as resource transformations and flows in a node-link diagram. Therefore, it is quite easy to connect the wrong resource (flow) into a transformation (node). In order to detect this, we will experiment with specifying resource types. We envision a "resource class" based system. For example, a transformation may require an energy resource. The model may provide for electricity, natural gas, or oil as possible energy resources. Any of these would be acceptable and could be converted to energy by the system. The system will also support a generic flow, which will be a member of all resource classes so that users may avoid the complexities of resource types.

Support for Program Understanding in reMIND+

reMIND+ is based on a hierarchical flow diagram with a semantically zooming browser. The end-user will be able to hide details and view the model at the subsystem level. This will provide an overview of the model. The end-user can also reveal the details of a submodel in order to understand it more fully. We also plan to introduce the ability to more descriptively name flows and specify their resource class.

Support for Reuse in reMIND+

The chief shortcoming of the reMIND system is that all models are flat and effectively monolithic. This means that copying an entire model and modifying it is the only practical means of reuse. reMIND+ introduces the concept of a submodel both as a abstraction and as a unit for reuse and sharing. It will support libraries as a repository for submodels. Finally, individual nodes may be described and deposited in a library permitting reuse of common, simple transformations.

Automated Error Checking

Error checking in reMIND was limited to verifying that each input to a transformation was connected and that each output from a transformation was connected. reMIND+ will add the ability to check resource type match as well. In order to reduce the burden for the end-user, we also anticipate the need to automatically convert units such as liters of oil to joules.

CONCLUSION

While much progress has been made in the design of end-user programming systems we feel that improvements need to be made in support for long-term system development. Some of these improvements can be borrowed from professional software development and adapted to end-user systems.

REFERENCES

1 Green, T. R. G. and Petre, M. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7, 2 (June 1996) 131-174.

2 Moström, J. E. and D. Carr, Programming Paradigms and Program Comprehension by Novices, *Proceedings of the 10th Annual Workshop of the Psychology of Programmers Interest Group* (PPIG'98), Milton Keynes, UK. Jan. 1998, 117-127.

3 Nardi, B. A. and Miller, J. R. Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration*, Morgan Kaufmann., 1993, 260-271.

4 Nilsson, K. Cost-effective Industrial Energy Systems. Linköping University, Dissertation #315, 1993, ISBN 91-7871-156-8, ISSN 0345-7524.

5 Pane, J. F., Ratanamahatana, C. A., and Myers, B. A. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, 54, 2 (Feb. 2001) 237–264.

6 Shu, N. C. Visual Programming Languages: A perspective and a dimensional analysis. *Visual Programming Environments: Paradigms and System*s, E. P. Glinert, Ed., IEEE Computer Society Press, 1990, 41-58.

7 Macromedia. Macromedia Flash MX, http://www. macromedia.com/software/flash/, visited 2003-01-14.

Software Shaping Workshops: Environments to Support End-User Development

M.F. Costabile, A. Piccinno
Dipartimento di Informatica
Università di Bari
70125 Bari, Italy
+39 080 544 3300
{costabile, piccinno}@di.uniba.it

D. Fogli, P. Mussio
Dipartimento di Elettronica per l'Automazione
Università di Brescia
25123 Brescia, Italy
+39 030 3715450
{fogli, mussio}@ing.unibs.it

ABSTRACT

In the Information Society, end-users keep increasing very fast in number, as well as in their demand with respect to the activities they would like to perform with computer environments, without being obliged to become computer specialists. There is a strong request of providing end-users with powerful and flexible environments, tailorable to the culture, skills and needs of very diverse end-user population. In this paper, we discuss a framework for End-User Development (EUD) and present our current work to design environments that support the activities of domain-expert users, with the objective of easing the way these users work with computers. Such environments are called workshops in analogy to artisan workshops since they provide users with the tools, organized on a bench, that are necessary to accomplish their specific activities by properly "shaping" software artifacts.

Keywords
End-User Development, Domain-Expert Users, Software Environments.

INTRODUCTION AND MOTIVATION

The development of computer systems that provide accessibility and high quality of interaction to their end-users is the big challenge we face in the information society. Following the definition of Cypher, end-user is a person who uses a computer application as part of daily life or daily work, but is not interested in computers per se [13]. In accordance with [8], we recognize that most end-users are experts in a specific domain, not necessarily experts in computer science, who use computer environments to perform their daily tasks. Our work primarily addresses the development of software environments and tools, which supports such domain-expert users.

In [12] we have analyzed the needs of domain-expert users; it appears that they are very demanding with respects to the software they use, they are willing to carry out activities of End-User Development (EUD), meaning by EUD the possibility of modification and even creation of software artifacts, in order to tailor the software to the users' real needs.

Several phenomena contribute to the current difficulty of user-system interaction. Some of these are described in the following:

- *Communicational gap between designers and users* [17][2]. This phenomenon is related with the variety and complexity of the knowledge involved in interactive system design, which pose a serious problem of knowledge elicitation and sharing. The communicational gap arises from the fact that designers and users have different cultural backgrounds, and, as a consequence, detain distinct types of knowledge and follow different approaches and reasoning strategies to modeling, performing and documenting the tasks to be carried out in a given application domain. Because of the communicational gap, the interactive system usually reflects the culture, skill and articulatory abilities of the designer. Users find often hurdles in mapping the interactive tools into their specific culture, skill and articulatory abilities. Users may be unable to follow their own solving strategies during the interaction process.
- *User diversity*. As highlighted in [9], hurdles arise in designing interactive systems because of user diversity even within a same population. Such diversity depends not only on user skill, culture, knowledge, but also on specific abilities (physical and/or cognitive), tasks and context of activity. As a consequence, specialized user dialects stem from user diversity [11], rising from the existence of users sub-communities which develop peculiar abilities, knowledge and notations, e.g. for the execution of specialized subtasks. If, during system design, this phenomenon is not taken into account, some users may be forced to adopt specific dialects related with the domain but different from their own and possibly not fully understandable, making difficult the interaction process.
- *Co-evolution of systems and users* [10] [2]. It is well known that "using the system changes the users, and as they change they will use the system in new ways" [19].

These new uses of the system make the environment evolve, and force to adapt the system to the evolved user and environment. This phenomenon is called co-evolution of system, environment and users [7]. Designers are traditionally in charge of managing the evolution of the system. This activity is made difficult by the communicational gap.

- *Grain*. Every tool is often suited to specific strategies in achieving a given task. Users are induced by the tool to follow strategies that are apparently easily executable, but that may be non optimal. This is called "grain" in [14], i.e. the tendency to push the users towards certain behaviors. Interactive systems tend to impose their grain to users resolution strategies, a grain often not amenable to user reasoning, and possibly even misleading for them [14].

As defined in [21], the requirement of universal access implies accessibility, usability, and acceptability of Information Society Technologies by anyone, anywhere, anytime, thus enabling equitable access and active participation of potentially all citizens in existing and emerging computer-mediated human activities. Our view of universal design does not imply that a single user interface is suitable for all users. Instead, as designers we put effort in proposing solutions tailored to the needs of different user populations. It is important to achieve a right balance between adaptability and adaptivity. Adaptability calls for a system flexibility that allows users to perform modifications performed by users that may go from simple parameterizations to more complex EUD activities [12]. On the other hand, adaptivity calls for a system capable of monitoring users' behavior and other contextual properties, like the current task or situation, and use different approaches to automatically adapt itself, for the benefits of users. Such concepts were addressed at a recent Workshop of EUD-Net [15].

Because of their different cultural backgrounds, designers and users may adopt different approaches to abstraction, since, for instance they may have different notions about the details that can be abstracted away. Moreover, users reason heuristically rather than algorithmically, using examples and analogies rather than deductive abstract tools, documenting activities, prescriptions, and results through their own developed notations. These notations are not defined according to computer science formalisms but they are concrete and situated in the specific context, in that they are based on icons, symbols and words that resemble and schematise the tools and the entities which are to be operated in the working environment. Such notations emerge from users' practical experiences in their specific domain of activity [17][14]. They highlight those kinds of information users consider important for achieving their tasks, even at the expense of obscuring other kinds [20], and facilitate the heuristic problem solving strategies, adopted in the specific user community.

A system acceptable by its users should have a gentle slop of complexity: this means it avoids big steps in complexity and keeps a reasonable trade-off between ease-of-use and expressiveness. Systems might offer for example different levels of complexities, going from simply setting parameters, to integrating existing components, up to extending the system by programming new components [15]. To feel comfortable, users should work at any time with a system suitable to their specific needs, knowledge, and task to perform. To keep the system easy to learn and easy to work with, only a limited number of functionalities should be available at a certain time to the users, those that they really need and are able to understand and use. The system should then evolve with the users, thus offering them new functionalities only when needed.

The problem of managing user culture and co-evolutive design is growing in importance because the WWW technologies allow users of different cultures to share data and knowledge, and to collaborate in real time to perform common tasks. We are currently refining a design methodology that faces the challenges posed by the four phenomena presented above [18], [11]. It is described in the next section.

SOFTWARE SHAPING WORKSHOPS

The aim of the design methodology we are developing is to design multimedia and multimodal environments that support the activities of domain-expert users, with the objective of easing the way these users program and interact with computers. The design methodology is collaborative in that, by recognizing that users are experts in their domain of activity, it requires that representatives of the users collaborate to the development of the system as domain experts, in a team with HCI experts and software experts. Moreover, the team of designers, including domain experts, is in charge of driving the co-evolution of the system.

Recognizing users as domain experts means recognizing the importance of their notations and dialects as reasoning and communication tools. Moreover, with the aim of increasing the closeness between programming and problems worlds [16], our design methodology adopts users' notations as core for the development of the language used for user-system interaction [6]. Adopting users' notation also supports the team of designers in identifying the grain problems and in defining their solutions.

In scientific and technological communities, such as mechanical engineers, geologists, physicians, experts often work in a team to perform a common task. The team might be composed by members of different sub-communities, each sub-community with different expertise. Members of a sub-community should need an appropriate computer environment, suitable to them.

The developed environments appear to their users as workshops, providing them with the tools, organized on a bench, that are necessary to accomplish their specific activities. Users work in analogy to artisans, who carry out their work using their real or virtual tools, as it occurs in blacksmith or joiner workshops. For this reason, the computer environments developed with this methodology are called Software Shaping Workshops (SSWs) [11].

SSWs allow users to develop software artifacts without the burden of using a traditional programming language, but using high level visual languages tailored to users' needs. Moreover, users get the feeling of simply manipulating the objects of interest in a way similar to what they might do in the real world. Indeed, they are creating an electronic document through which they can perform some computation, without writing any textual program code.

The SSW methodology is aimed at generating virtual environments, the workshops, in which each user sub-community interacts using a computerized dialect of their traditional languages and virtual tools, which recall the real tools with which users are familiar. In other words, the SSW approach provides each sub-community with a personalized workshop, called *application workshop*. Using an application workshop, experts of a sub-community can work out data from a common knowledge base and produce new knowledge, which can be added to the common knowledge base. All the data available for the community are accessible by each expert using the specialist notation of its sub-community.

The application workshops are designed by a design team composed by various experts, who participate to the design using workshops tailored to them. These workshops are called *system workshops* and are characterized by the fact that they are used to generate or update other workshops. In other words, using a system workshop, the design team defines notations and tools, which are added to the common knowledge base and exploited in the generated workshops. This approach leads to a workshop hierarchy that tries to bridge the communicational gap between designers and domain expert users, since all cooperate in developing computer systems customized to the needs of the users communities without requiring them to become skilled programmers [5].

The system workshop at the top of the hierarchy is the one used by the software engineers to lead the team in developing the other workshops. Each system workshop is exploited to incrementally translate concepts and tools expressed in computer oriented languages into tools expressed in notations that resemble the traditional user notations and therefore understandable and manageable by users. More precisely, at each level of the hierarchy but the bottom level, experts use a system workshop to create a child workshop tailored to a more specialized user.

The hierarchy organization depends on the working organization of the user community to which the hierarchy is dedicated: each hierarchy is therefore organized into a number of levels. The top level (software engineering level) and the bottom level (application level) are always present in a hierarchy. The number of intermediate levels is variable according to the different working organization of the user community to which the hierarchy is dedicated.

The SSW approach is aimed at overcoming the communicational gap between designers and users by a ‘gentle slope’ approach to the design complexity [15]. In fact, the team of designers performs its activity by: a) developing several specialized system workshops tailored to the needs of each designer in the team; and b) using the system workshops to develop the application workshops through in incremental prototypes [11][9]. In summary, the design and implementation of application workshops is incremental and based on the contextual, progressive gain of insight on the user problems, emerging from the activity of checking, revising and updating the application workshops performed by each member of the designer team.

Recognizing the diversity of users calls for the ability to represent a meaning of a concept with different materializations, in accordance with local cultures and the used layouts, sounds, colors, times and to associate to a same materialization a different meaning according, for example, to the context of interaction. The SSW methodology aims at developing application workshops which are tailored to the culture, skill and articulatory abilities of specific user communities. To reach this goal, it becomes important to decouple the pictorial representation of data from their computational representation [4]. In this way, the system is able to represent data according to the user needs, by taking into account user diversity. Several prototypes were developed in this line, in medical and mechanical engineering [18], exploiting C and C++. The XML technologies, which are founded on the same concept of separating materialization of a document from its content, are being extensively exploited.

The workshops in the SSW hierarchy are implemented as XML documents and a software tool has been developed allowing to create, manage and interact with such documents, whose content is distributed in the Web [9]. Finally, XML is also the technological basis to build the tools to generate the SSW hierarchy: each XML document can be steered by its users to self-transform into a new XML document representing a new workshop. On the whole, SSW hierarchy is generated from the system workshop of the software engineers by a co-evolutive process determined by the activities of the experts of the design team.

In a SSW, domain-expert users are allowed to perform various activities of EUD, e.g. those called in [12] modeling from the data and extended annotation. To give some examples, in [1], the system derives patterns of interaction from monitoring user-system interaction in order to allow system co-evolution based on the observation of user behavior; in the system presented in [11] the experts using the system workshop to develop, write comments next to data in order to remember to obtain what they did, how they obtained their results; they can also associate a new functionality with the annotated data, in order to make available such data to other users who work in a different SSW.

CONCLUSIONS

Some studies report that by 2005, there will be in USA 55 millions of end-users vs 2.75 millions of professionals developer [3]. Most end-users are asking for environments in which they can make some ad hoc programming activity related to their tasks and adapt the environments to their emerging new needs. Moreover, several phenomena contribute to the current difficulty of user-system interaction, such as the communicational gap often existing between designers and systems, the user diversity, the co-evolution of systems and users, and the grain imposed by software tools. The methodology discussed in this paper, by taking into account the four mentioned phenomena, is a step toward the development of powerful and flexible environments, with the objective of easing the way end-users interact with computer systems to perform their daily work.

REFERENCES

1. Arondi, S., Baroni, P., Fogli, D., Mussio, P. Supporting co-evolution of users and systems by the recognition of Interaction Patterns. *Proceedings of the International Conference on Advanced Visual Interfaces (AVI 2002)*, Trento (I), May 2002, 177-189.
2. Baroni, P., Fogli, D., Mussio, P. An agent-based architecture to support knowledge management in interactive system life-cycle, Accepted for presentation at WOA 2002" Dagli Oggetti agli Agenti - Dall'informazione alla Conoscenza", Milan (I), 2002.
3. Boehm, B. W., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Reifer, D.J. and Steece, B. *Software Cost Estimation with COCOMO II*, Prentice Hall PTR, Upper Saddle River, NJ, 2000.
4. Bottoni, P., Costabile, M.F., Levialdi, S., Mussio, P. Defining Visual Languages for Interactive Computing. *IEEE Trans. SMC, Part A*, 27 (6), November 1997.
5. Bottoni, P., Costabile, M.F., Levialdi, S., Mussio, P.: From User Notations to Accessible Interfaces through Visual Languages. In: Stephanidis, C. (ed*.): Universal Access In HCI: Toward an Information Society for All*, Vol. 3. Lawrence Erlbaum Associates, Mahawah, New Jersey, London, 252-256, 2001.
6. Bottoni, P., Costabile, M.F., Mussio, P. Specification and Dialogue Control of Visual Interaction through Visual Rewriting Systems, *ACM Trans. on Programming Languages and Systems TOPLAS*, Vol. 21, No. 6, 1077-1136, 1999.
7. Bourguin, G., Derycke, A., Tarby, J.C. Beyond the Interface: Co-evolution inside Interactive Systems - A Proposal Founded on Activity Theory, *Proc. IHM-HCI* 2001.
8. Brancheau, J.C., Brown, C.V. The Management of End-User Computing: Status and Directions. *ACM Computing Surveys* 25(4), 1993.
9. Carrara, P., Fogli, D., Fresta, G., Mussio, P. Toward overcoming culture, skill and situation hurdles in human-computer interaction. *Int. Journal Universal Access in the Information Society*, 1(4), 288-304, 2002.
10. Carroll, J.M., Rosson, M.B., Deliberated Evolution: Stalking the View Matcher in design space. *Human-Computer Interaction* 6 (3 and 4), 281-318, 1992.
11. Costabile, M.F., Fogli, D., Fresta, G., Mussio, P., Piccinno, A. Computer Environments for Improving End-User Accessibility. *Proc. of 7th ERCIM Workshop "User Interfaces For All"*, Paris, 187-198. 2002.
12. Costabile, M.F., Fogli, D., Letondal, C., Mussio, P., Piccinno, A. Domain-Expert Users and their Needs of Software Development, invited paper at Special Session on EUD, UAHCI Conference, Crete, June 2003.
13. Cypher, A. *Watch What I Do: Programming by Demonstration*. The MIT Press, Cambridge, 1993.
14. Dix, A., Finlay, J., Abowd, G., Beale, R. *Human Computer Interaction*, Prentice Hall, London, 1998.
15. EUD-Net Thematic Network, Network of Excellence on End-User Development, http://giove.cnuce.cnr.it/eud-net.htm.
16. Green, T.R.G, Petre, M.: Usability Analysis of Visual Programming Environments. *Journal of Visual Language and Computing* 7(2), 131-174, 1996.
17. Majhew, D.J. *Principles and Guideline in Software User Interface Design*, Prentice Hall, 1992.
18. Mussio P, Finadri M, Gentini P, Colombo F. A bootstrap technique to visual interface design and development. *The Visual Computer* 8(2), 75-93, 1992.
19. Nielsen, J. *Usability Engineering*, Academic Press, San Diego, 1993.
20. Petre, M., Green, T.R.G. Learning to Read Graphics: Some Evidence that 'Seeing' an Information Display is an Acquired Skill. *Journal of Visual Languages and Computing*, 4(1), 55-70, 1993.
21. Stephanidis, C. (Ed.), Salvendy, G., Akoumianakis, D., Arnold, A., Bevan, N., Dardailler, D., Emiliani, P.L., Iakovidis, I., Jenkins, P., Karshmer, A., Korn, P., Marcus, A., Murphy, H., Oppermann, C., Stary, C., Tamura, H., Tscheligi, M., Ueda, H., Weber, G., & Ziegler, J. Toward an Information Society for All: HCI challenges and R&D recommendations, *International Journal of Human-Computer Interaction*, 11 (1), 1-28, 1999.

Supporting End User Programming of Context-Aware Applications

Anind K. Dey and Tim Sohn
anind@intel-research.net, tsohn@cs.berkeley.edu

The emergence of context-aware applications, those that take into account their context of use, has shown the ability for rich interaction with the surrounding environment. However, although some of these applications have been developed, the proliferation of context-aware applications is inhibited by the lack of programming support to rapidly develop them. Currently, to develop a context-aware application, developers are required to either design and implement their own application from scratch requiring them to write code which directly interacting with devices, or use a toolkit that hides a lot of the device details from them [2].

However, even with low-level toolkit support for acquiring context, experienced developers are still required to write a large amount of code to develop relatively simple applications. In order for ubiquitous computing applications, the superset of context-aware applications, to truly become ubiquitous, the following two things (among many others) need to occur. The first is that applications have to be easier to design, prototype and test, supporting faster iterations for the design-prototype-evaluation cycle. The second is that designers and end users need to be empowered to build their own applications. Empowering designers will allow people with superior creative skills to build innovative applications without having to be expert programmers. Empowering end users will allow users to build applications that are customized and appropriate for their own use. Rather than leaving control of these systems in the hands of programmers who do not have to live with them, end users should be given this control.

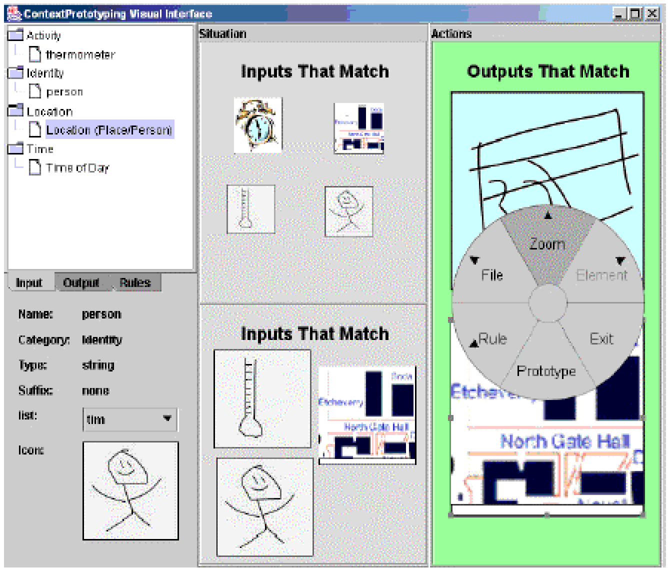
In our previous research, we have looked at making it easier for programmers to build context-aware applications through the use of the Context Toolkit [2], which removed the need to deal with the underlying details of sensors similar to the way that graphical user interface toolkits removed the need to deal with low-level details for building interfaces. While this eased the burden on programmers, it did not remove the burden, and certainly did not open up the space for designers and end-users in the way that systems like AgentSheets [7] and Stick-e Notes [8] did. These are the issues we are now concentrating on. In this paper, we will present an initial implementation of our visual environment for supporting end-user prototyping and present ideas for other end-user prototyping environments.

VISUAL PROTOTYPING: THE iCAP SYSTEM

iCAP is the intermediate layer between low-level toolkits and users, providing a powerful tool for developing interesting, complex context-aware applications, while allowing developers to prototype applications *without writing any code*. A context-aware application typically consists of an infrastructure to capture context and rules governing how the application should respond to changes in this context. iCAP is an informal pen-based tool that allows users to quickly define input devices that collect context and output devices that support response, create application rules with them, and test the rules by interacting with the devices in a *run mode*. The behavior of created devices can either be simulated by this tool, or mapped to actual devices. We built iCAP using the Java 2 SDK version 1.4, on top of SATIN [3], a toolkit for building informal pen-based interaction systems.

THE iCAP INTERFACE

iCAP has one window with two main areas (see Figure 1). On the left is a tabbed window that is the repository for the user-defined inputs, outputs, and rules. The input and output components are associated with graphical icons that can be dragged into the center area, then be used to construct a conditional rule statement.



Name: person
Category: Identity
Type: string
Suffix: none
list: tim
Icon:

Figure 1. The iCAP user interface with an example rule that uses two sheets.

The center area contains the two elements of a conditional rule statement, which is inherent within context-aware applications. An example rule is: *if John is in the office after 5pm and the temperature is less than 50 degrees or if Jane is in the bedroom and the temperature is between 30 and 60 degrees, turn on the heater in the house* (Figure 1). The left side represents the ‘‘if’’ portion of the rule conditional, and can be split into one or more ‘‘sheets’’. Inputs on a single sheet are related by a conjunction and multiple sheets are related by a disjunction. The right side of this area represents the ‘‘then’’ portion of the rule condition. Disjunction amongst different outputs is rare, thus only a single output sheet is currently supported. We implemented Pane and Myers’ matching scheme to allow users to visually specify the Boolean logic of each rule [6].

Instead of traditional pull-down menus for executing commands, we use pie menus to better support pen interaction. In addition, we also support gestures for issuing common commands such as cut, copy, and paste of inputs and outputs.

INTERACTION

iCAP involves specifying inputs and outputs, using these elements to construct application rules, and then testing the entire set of rules in a run mode.

Creating Inputs and Outputs

Each input and output component in iCAP is associated with a graphical icon. These icons are sketches drawn by the user upon creation of each component. Each icon is colored differently depending on whether it is an input or output device. The repository window pie menu supports creation of inputs. Each input contains a suffix (*e.g.* degrees Celsius for temperature), type (*e.g.* integer, string), and four categories or primary types of context: Activity, Identity, Location, and Time. An input's potential values can be provided as a range or list.

Outputs are created in the same manner as inputs, however contain different parameters to specify. Each output is either a binary or a gradient device. By default, the number of levels in a gradient device is between 1 and 10 inclusive. In addition, there are five categories an output device is associated with corresponding to the five human senses: Sight, Sound, Smell, Taste, and Touch.

Constructing Rules

Rules are constructed by dragging and dropping inputs and outputs onto the ''if'' and ''then'' sheets of each rule. For example, if the user were interested in a temperature sensor, he would define a temperature input, and drag the corresponding icon onto the respective sheet. After dragging each corresponding icon, the user needs to setup certain parameters, or *conditions*, governing the behavior of the input. Using our temperature sensor, the user may want to know when the temperature is less than 50 degrees, or possibly between 30 and 60 degrees. We allow the user to specify a conjunction of up to three conditions using the following operators: less than, less than equal, greater than, equal, not equal. Multiple condition sets can be defined, and are all related by a disjunction.

Evaluating the Application

After a number of rules have been defined, the entire rule set can be tested using the iCAP engine in run mode. The engine can either be set to simulate the context-aware environment, or be used in conjunction with a real context-aware environment [2]. Users can interact with the engine to change the value of defined inputs, and evaluate the behavior of the rules being tested. With the engine, users are able to quickly design and test their applications, without having to create an entire infrastructure for collecting or simulating context and without writing any code.

OTHER PROTOTYPING ENVIRONMENTS

While we have focused our attention on a visual environment for supporting end users and designers in building context-aware applications, we have some other ideas for other prototyping environments. Since context-aware applications are often focused on physical phenomena, future prototyping environments should exist outside the graphical world and in the physical world.

One idea for an environment would be to create physical representations of the graphical icons. The physical representations could be more detailed and familiar than graphical icons and the use of them would leverage off of the known benefits of tangible user interfaces [4]. The prototyping environment would be similar to iCAP, where rules are constructed out of basic input and output elements. We believe that the physicality of the elements are more appropriate for physically-based applications and implementing this environment would let us test this hypothesis.

A further idea for a physically-based context-aware prototyping environment is to allow users to create rules by "acting" them out or by behaving naturally. This is commonly known as programming-by-demonstration [1]. To create a rule that would turn on the light when the user entered the room, the user would enter the room and turn on the light. A number of training examples will be necessary (a greater number as rule complexity increases) for an underlying learning system to understand the rule. The implementation of such an environment is complex, requiring instrumentation of the user's environment and a sophisticated learning system. The first requirement is necessary for executing a context-aware application, so this should not be a burden. The need for the second is balanced with the advantage of more natural programming by the end user.

FUTURE DIRECTIONS

While we have received informal feedback from local designers of context-aware systems, we are planning to conduct a more formal study of our iCAP with a number of real users to see what features are used, and how to improve interaction with the system. Our goal is to enable both designers and end-users with the ability to create and modify context-aware applications, giving them the power that only programmers enjoy today. Once we gain experience with iCAP, we will look to building our more physically-based prototyping environments.

REFERENCES

1. Cypher, A. "Eager: Programming Repetitive Tasks By Example." In Proceedings of CHI '91, pp. 33-39. 1991.
2. Dey, A.K., Salber, D. and Abowd, G.D. ''A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications.'' Human-Computer Interaction Journal, 16 (2-4), pp. 97-166. 2001.
3. Hong, J.I. and Landay, J.A. ''SATIN: A Toolkit for Informal Ink-based Applications." In Proceedings of User Interface and Software Technology, pp. 63-72. 2000.
4. Ishii, H. and Ullmer, B. "Tangible Bits: Towards Seamless Interfaces Between People, Bits and Atoms. In Proceedings of CHI '97, pp. 234-241. 1997.
5. Mozer, M.C. "The Neural Network House: An Environment That Adapts to its Inhabitants." In Proceedings of the AAAI Spring Symposium on Intelligent Environments, pp. 110-114. 1998.
6. Pane, J.F. and Myers, B.A. ''Tabular and Textual Methods for Selecting Objects from a Group.'' In Proceedings of International Symposium on Visual Languages, pp. 157-164. 2000.
7. Pascoe, J. ''The Stick-e Note Architecture: Extending the Interface Beyond the User.'' In Proceedings of Intelligent User Interfaces, pp. 261-264. 1997.
8. Repenning, A. ''Creating User Interfaces with Agentsheets.'' In Proceedings of Symposium on Applied Computing, pp. 190-196. 1991.

Some Generic Mechanisms for Increasing the Usability of EUD Environments

Simone Diniz Junqueira Barbosa
Departamento de Informática, PUC-Rio
R. Marquês de São Vicente, 225
Gávea, 22453-900
Rio de Janeiro, RJ, Brazil
simone@inf.puc-rio.br

Philippe Palanque
LIIHS-IRIT
Université Paul Sabatier
118, route de Narbonne, 31062
Toulouse Cedex, France
palanque@irit.fr

Rémi Bastide
LIIHS-IRIT
University of Toulouse I
Place Anatole France, 31042
Toulouse Cedex, France
bastide@irit.fr

1 INTRODUCTION

Successful EUD environments need to support users' transition from passive users to active "end-user-designers". On the one hand, macro recording facilitates the creation of extensions, but fails at providing the means for changing what has been recorded. Approaches based on programming by demonstration (Lieberman 2001, Cypher 1993) go one step further, but the proposed mechanisms are typically too close to the application domain to be easily generalized and reused. On the other hand, scripting and programming languages allow both the creation and maintenance of user extensions, but they require users to know how to program. There is a growing need for alternative EUD approaches that fill the gap between these two extremes, providing usable generic EUD mechanisms that have the computing power of programming languages and yet do not burden users with having to learn a wide range of programming constructs and syntax.

This paper illustrates the use of application models at different levels of abstraction and following different interaction styles (direct manipulation and conversational) as a basis for generic EUD mechanisms. These mechanisms are introduced here according to the Norman's seven stages of action (see figure 1). The claim is to propose "basic principles" to be embedded in EUD environments in order to reduce potential difficulties.

The model-based approaches to EUD described here are targeted to motivated end users who would like to be active domain designers (Fischer 2002), i.e., users that are knowledgeable in specific domains and are enough interested in their work so as to frequently engage in personally meaningful activities. We assume that, not only would these users go a little out of their way during interaction to create an extension they deem useful, but also that they are able to understand and manipulate formal languages (Nardi 1993).

Pane et al. (2001) have conducted experiments with the language used by nonprogrammers in programming tasks, and their findings suggest that the usability of programming languages may be improved by providing different language styles, each one more natural for a certain (part of) the programming task. Inspired by this idea, the approaches described here are not meant to be used in isolation, but rather to go together with other EUD approaches.

2 UCD and EUD

User-centered design aims to build applications in which the user, by interacting with the system image, is able to build a conceptual (mental) usage model compatible with the designer's (,). Figure 1 presents Norman's seven stages of action, which describes the steps users go through during interaction. The left-hand side of the figure represents the execution path, i.e. the set of activities that have to be carried out by the user in order to reach the goal, whereas the right-hand side represents the evaluation path, i.e. the set of activities that have to be carried out by the user in order to interpret the changes resulting from his/her actions with respect to his/her goals.

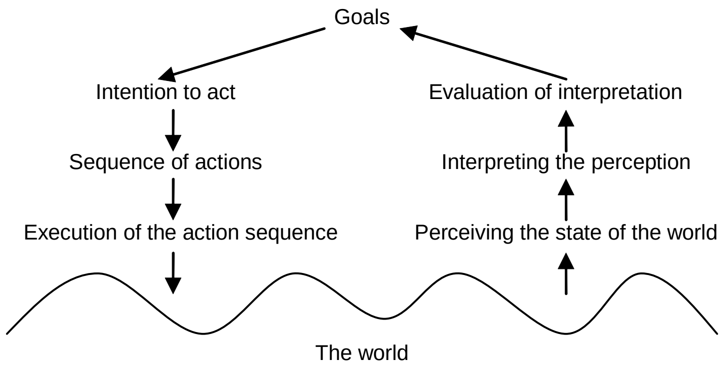

Goals
Intention to act
Evaluation of interpretation
Sequence of actions
Interpreting the perception
Execution of the action sequence
Perceiving the state of the world
The world

Figure 1. Norman's seven stages of action (Norman 1988, p.47).

Based on this model, Hutchins et al. (1986) have studied the directness of a user interface by analyzing two dimensions when traversing both execution and evaluation gulfs: distance (semantic and articulatory) and engagement (direct or conversational). The semantic distance is the (lack of) correspondence between the user's view of the domain and the perceived application model, whereas the articulatory distance is the (lack of) correspondence between this model and the user interface. The concept of engagement is related to how close do users feel involved with a world of objects: *direct* means they are engaged with the objects themselves, as in direct manipulation, whereas *conversational* means they are engaged in conversation(s) with the system, and the system will in turn act on these objects. In the next two sections, we will explore how these forms of engagement may be used in EUD.

In UCD, it is the designer's responsibility to build applications with gulfs as narrow as possible at design-time, striving for short semantic and articulatory distances. During interaction, users must learn how to overcome these distances and traverse the gulfs. It is thus essentially important that designers effectively communicate, through the user interface, their interpretations and assumptions about the application domain and its potential contexts of use, in order to increase the quality of the match between the users' models and their own, thus reducing the semantic distance in both gulfs (). In EUD, this is critical because users will assume the role of designers, albeit limited. During EUD, users must themselves shorten the distances and narrow the gulfs, instead of just traversing them. They should be able to modify the application model so it matches more closely their view of the domain, thus shortening the semantic distance. They should also be able to make changes in the user interface in order to both reflect the modified application model and provide a form of interaction that is more "natural" to users with respect to this model, thus shortening the articulatory distance.

At design-time, the semantic and articulatory distances may be shortened by designing an application model and user interface closer to the application domain. However, this reduces the systems' applicability to other domains. In EUD systems, this problem is mostly due to the tight coupling between the domain-related part of the application and its EUD part. This is one of the major challenges faced by EUD based on programming by example (Lieberman 2001, Cypher 1993). In order to avoid losing generality, we propose to keep these two parts more loosely coupled. One way to achieve this is to build usable generic EUD mechanisms that make use of specific domain and application models. In the next sections, we assume that the spectrum of possible interactions is represented in a semantic application model that is made available to users via the EUD mechanism.

3 MODEL-BASED EUD with DIRECT ENGAGEMENT

Although the expressions "model-based" and "direct engagement" may seem contradictory, we view the application model as *the* object with which users (domain designers) are engaging during EUD. In EUD, execution may be thought of as program creation and editing, whereas evaluation comprises program execution and appraisal. These activities are usually made available to the user in a modal, sequential way: first the user writes the program and then (often in a different context) runs it. Making these two activities separate and modal introduces difficulties in both perceiving the program execution and interpreting its behaviour (). This problem is not specific to EUD and even professional developers encounter the same difficulties. In order to promote direct engagement in software development and overcome these artificially-modal activities, a programming environment called PetShop was developed to support these activities in parallel ().

3.1 The PetShop Environment and ICO Language

PetShop allows for rapid prototyping, iterative and modeless construction of applications, by providing a way of having both program execution and program editing at a time. Figure 2 presents a snapshot of Petshop at runtime. The small window on top of the picture (whose caption reads "RangeSlider") corresponds to the execution of the visual program (represented by the Petri net) in the bigger window underneath (the main PetShop window).

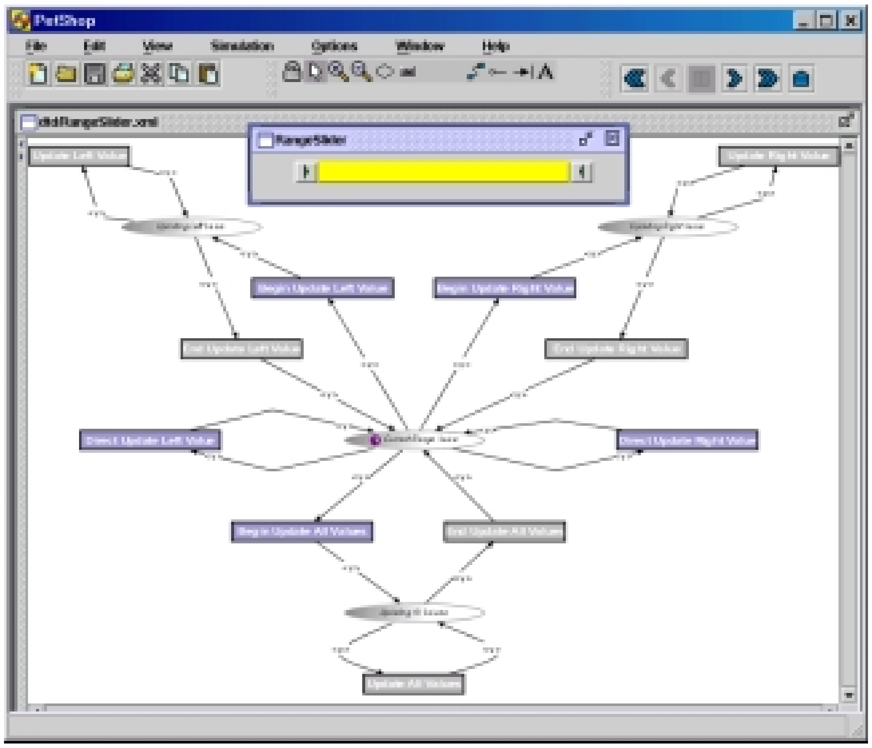


Figure 2. Integrated program editing and execution within PetShop.

The program illustrated in Figure 2 corresponds to the behaviour of a range slider, shown in Figure 3.

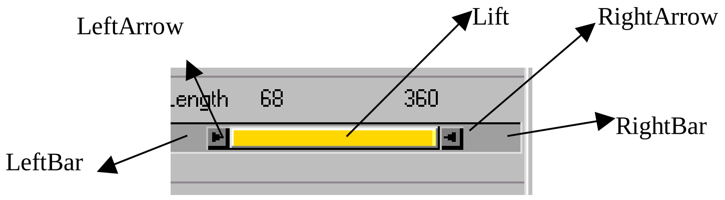


Figure 3. A simple application: a range slider (Ahlberg & Shneiderman 94)

A range slider is a basic interactive component that allows the user to select values within a range (between a lower and an upper bound). The range slider belongs to the hybrid category of interactors as it can be manipulated both in a discrete and continuous way. Quite complex interactors such as this have been increasingly used in interactive applications and companies building user interface toolkits (such as Microsoft and Ilog) have already invested in component technology. However, the more complex the components, the less reliable they are.

Petshop makes use of an object-oriented, distributed and concurrent language called Interactive Cooperative Objects (ICO) (). This language is dedicated to the construction of highly interactive distributed applications. It is to be used by expert programmers skilled in formal description techniques, object-oriented approaches, distributed and interactive systems. Even though the initially targeted programmers were not end users of the applications to be constructed, it has always been a goal to increase the usability of the notation by providing ways of reducing the evaluation gulf.

The behavioural description of the range slider in ICO models the set of events it can react to (mouse up, mouse move and mouse down), the set of states it can be in (the distribution of tokens in the places (ellipses) Petri net) and the set of actions the range slider can perform (the transitions (rectangles) in the Petri net). In the Petshop environment, the programmer can simultaneously interact with the application (use the range slider) and see the impact of his/her action on the behaviour of the visual program. Another possibility is to modify the visual program and immediately see the impact on the program execution. For instance, in Figure 2, the transition "begin Update Left Value" is associated to the left button of the range slider. If, by modifying the Petri net, the programmer makes this transition unavailable (for instance by adding an input place without any token in it), then the button will immediately appear as disabled (greyed out), and acting on it will have no effect. Figure 2 showed only a subset of this interactor's behaviour; the complete description of the case study can be found in (Navarre et al. 2000).

One of the problems of building a concurrent program is, first, to understand its behaviour, and then to understand whether this behaviour is similar to the expected one. Some preliminary evaluation of ICO language and PetShop have been conducted as part of the Mefisto LTR Esprit Project. However, in order to quantify and confirm the results of this early evaluation, more usability tests must be conducted. This will be done in the framework of a military funded project starting in January 2003.

4 MODEL-BASED EUD with CONVERSATIONAL ENGAGEMENT

Most work dedicated to EUD has focused on tasks involving procedural knowledge, e.g. automation of repetitive tasks. However, declarative knowledge is also present in many of today's applications, among which the most well-known is probably the definition of formatting styles in word processors, stylesheets, and graphical editors. The definition of default values for parameters in an application also fall in this category, and this aspect has seldom been explored.

In EUD, declarative knowledge is typically left to the realm of scripting and programming languages. In this section, we present a way for end users to declaratively employ certain figures of speech (namely: analogies, metaphors and metonymies) that operate on an application model to extend it (Barbosa & de Souza 2001). Figures of speech are best explored in language-based interfaces, where users have a wider range of linguistic resources to express themselves. Hence the choice to take a conversational stance in this kind of EUD. This work is based on research in the field of Cognitive Science which suggests that we (humans) think and express ourselves extensively in non-literal ways (Lakoff & Johnson 1980, Ortony 1993). In particular, we make use of metaphors and metonymies in order to understand or explain a (typically unfamiliar) concept in terms of other (more familiar) concepts, by highlighting a concept's characteristics or relations, and concealing others.

The EUD approach described in this section uses a particular kind of linguistic mechanism that achieves the same effect as practical extensions. It is called *extension by interpretation* (Barbosa & de Souza 2001), and it may be viewed as a shortcut to performing a series of extensions to the application model. It presents an EUD mechanism in which the interface language allow users to produce metaphorical or metonymic expressions, and the language interpreter attempts to make sense of such expressions. When it succeeds, it behaves according to the interpretation it derived; when it doesn't, it interacts with the user based on partial interpretations. If no interpretation is derived, the interface reacts as any typical interface in non-extensible software, issuing an error message.

The sense-making process is a kind of abductive process (Peirce 1931) that generates possible interpretations to users' expressions by means of specific metaphorical and metonymic operators. One of the hardest EUD challenges is related to the users' lack of knowledge about the underlying application models. By drawing on the works of French (1995) and Holyoak and Thagard (1996), this EUD approach helps fill this knowledge gap with an enhanced representation of domain and application models, which are manipulated by abductive mechanisms that interpret metaphorical and metonymic utterances. These utterances often account for what can be diagnosed as imprecise and incomplete knowledge in traditional approaches. In order to help users acquire a better understanding of the application, we calculate possible interpretations for their utterances and give them feedback about the reasoning process involved in each interpretation. This feedback progressively unfolds to the user aspects of the internal application structure (DiGiano 1996, DiGiano & Eisenberg 1995).

In order to be able to generate and interpret metonymic and metaphoric expressions, designers need to represent the application model both as an ontology, i.e. relations among elements and identify which ones may be part of a metonymic chain, and as an interaction model, which describes the dynamic behaviour of the application.

4.1 Metonymies and Metaphors

A metonymic utterance occurs when reference to an element is made by another element with which the first has a relation of part-whole, content-container, cause-effect, producer-product, among other possibilities. For example, when we say "He's got a *Picasso*", we mean he's got a work of art produced by Picasso. In a computer application, we might express "copy the *boldface*", to mean "copy the text formatted in boldface". Still regarding computer applications, metonymies can also be used to generate iterations and recursions. For instance, in a graphical editor that allows users to group objects, if a user selects a group and chooses a different fill color, it iterates through all elements in the selected group and applies the chosen fill color to each element that can be filled, individually. This is clearly figurative speech, where of course that only the "colorable" elements in the group should be affected. Nevertheless, such usage of metonymies is typically *ad hoc*, or incidental, not to be consistently found elsewhere in the application. Users must learn where such metonymies may be used in isolation, and shouldn't expect a predictable behaviour of seemingly analogous situations.

Composition and aggregation relations, such as part-of, are natural candidates for metonymy. Other relations must be explicitly declared as having metonymic potential, such as: location, ownership, possession, creation, and many others. When interpreting a user's utterance, the metonymic chains in the application ontology are traversed, making paradigmatic substitutions in it and checking if the resulting expression has a literal interpretation. A valid substitute becomes the *metonymic target*. The utterance interpretation is the result of an iteration through every element in the original expression obtained by following the chain to the metonymic target, or every metonymic target reached from the original expression, depending on the direction traversed: from “whole” or “producer” to “part” or “product” (typically 1-to-n); or from “part” or “product” to “whole” or “producer” (typically n-to-1).

Metaphors may arise when comparing the relations between pairs of elements. For example, there may be a relation “written by” linking a text to its author, and the instances “*O Cortiço* written by Aluísio de Azevedo”, and “*O Guarani* written by José de Alencar”. The expression “Aluísio de Azevedo’s *O Guarani*” will result in retrieving an instance of “Something written by Aluísio de Azevedo”. If the underlying domain representation is rich enough to single out Alencar’s *O Guarani* as his most famous novel, the metaphorical representation can qualify “something written by” with the attribute “most famous”, and thus retrieves Azevedo’s novel that is, in the representation, as remarkable as *O Guarani* is for Alencar.

The appropriateness and sophistication of metonymic and metaphoric interpretations is directly proportional to the expressiveness of the underlying domain models. We may use classifications, relations, and attributes in the application ontology to generate and (try to) disambiguate metaphorical interpretations for users’ non-literal expressions. However, when it is impossible to disambiguate terms or when there are many alternatives for interpretation, the application should present choices to users, along with an explanation about how they were generated, and have users select the one they mean, or discard them all and try to use another form of expression.

4.2 Achieving EUD using Metaphors and Metonymies

In order to achieve EUD using metaphors and metonymies, it is necessary to interpret utterances that modify the application ontology and/or its behaviour. An example of such an extension is as follows: suppose A, B, and C are elements defined in the ontology, and that A has a relation R1 with B. A user’s utterance of the form **D is the B of C** triggers a search for an element of a “similar nature” of C, i.e. that is classified together with C in one or more groups). This search may be illustrated by Figure 4:

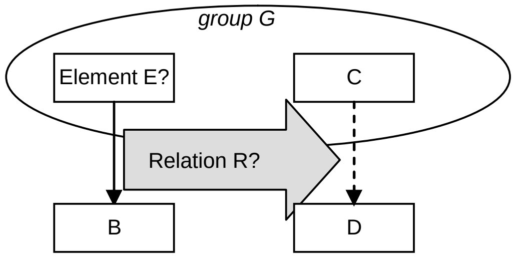


Figure 4. Illustration of a search mechanism for calculating metaphors

If the unknown element E is matched to A and relation R? is matched to R1, then the EUD mechanism creates relation R1 from C to D. This is of course a very simple example, for there are usually many candidates for element E or a set of relations from E to B and other elements. In this case, interaction with the user is necessary to disambiguate and choose or even redirect the relevant relations. This process can be abbreviated if the user’s utterance is not a metaphor, but an analogy in the form “D is to C as B is to A”. However, the above example illustrates a radical attempt to shortcut the extension, which may be necessary if the user does not have a clear, complete understanding of the underlying ontology (e.g. forgets the name of A). The situation worsens as the application’s semantic distances increase, because a basic requirement for effectively using this kind of figurative language in our communication, is that the user shares the same knowledge, assumptions, and cultural background (Lakoff 1987) as embedded in the application ontology. More complex extensions, involving attributes, relations with attributes, and dynamic behaviour, can be found in (Barbosa & de Souza 2001).

It is important to notice that this approach does not aim to substitute procedural EUD, but rather complement it. For instance, it could be used together with the work on Programming by Analogous Examples (Repenning & Perrone 2001), which already enables users to directly input a simple kind of analogical expression.

5 CONCLUDING REMARKS

The mechanisms described here for EUD are generic, but they are applied to domain and application-specific models. They may be used in a variety of domains, but it is the richness of representation that will determine the opportunity for interesting EUD. We believe that the integration of one or both of the presented model-based approaches (by direct manipulation and following a conversational paradigm) to other EUD techniques may not only bring more power to users, but also soften the learning curve necessary for effective EUD to take place. Using a variety of integrated EUD mechanisms will make it easier for a wider class of users to effectively work as domain designers.

6 REFERENCES

Ahlberg C. & Shneiderman B. The Alphaslider: A Compact and Rapid Selector. ACM SIGCHI conference on Human Factors in Computing Systems, CHI'94, Boston, pp. 365-371. 1994.

Barbosa, S.D.J., de Souza, C.S. Extending Software through Metaphors and Metonymies. Knowledge-Based Systems 14, pp.15–27. 2001.

Bastide R. & Palanque P. A Visual and Formal Glue between Application and Interaction. International Journal of Visual Language and Computing, Academic Press Vol. 10, No. 5, pp. 481-507. 1999.

Butler R., Miller S., Potts J. & Carreño. A formal method approach to the analysis of mode confusion. In proceedings of 17th AIAA/IEEE Digital Avionics Systems Conference, Bellevue, 1998.

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

de Souza, C.S. The Semiotic Engineering of User Interface Languages. *International Journal of Man-Machine Studies*. No. 39. pp. 753-773. 1993.

DiGiano, C. A vision of highly-learnable end-user programming languages. *Child's Play '96 Position Paper*. 1996.

DiGiano, C. and Eisenberg, M. Self-disclosing design tools: A gentle introduction to end-user programming. In *Proceedings of DIS'95*. Ann Arbor, Michigan. ACM Press. 1995.

Fischer, G. "Beyond 'Couch Potatoes': From Consumers to Designers and Active Contributors". First Monday, volume 7, number 12 (December 2002), http://firstmonday.org/issues/issue7_12/fischer/index.html (last visited in January 2003)

French, R. *The Subtlety of Sameness*. Cambridge, MA: The MIT Press. 1995.

Holyoak, K.J. and Thagard, P. *Mental Leaps: Analogy in Creative Thought*. Cambridge, MA. The MIT Press. 1996.

Hutchins, E.; Hollan, J. and Norman, D. 86. Direct Manipulation Interfaces. in D. Norman and S. Draper (eds.) *User Centered System Design*. Hillsdale, NJ. Lawrence Erlbaum. pp.87-124. 1986.

Lakoff, G. and Johnson, M. *Metaphors We Live By*. The University of Chicago Press. Chicago. 1980.

Lakoff, G. *Women, Fire, and Dangerous Things.* The University of Chicago Press. Chicago. 1987.

Lieberman, H. (ed.) Your Wish is My Command: Programming by Example. San Francisco, CA: Morgan Kaufmann Publishers. 2001.

Nardi, B. *A Small Matter of Programming*. The MIT Press. Cambridge MA. 1993.

Navarre D., Palanque P., Bastide R. & Sy O. A Model-Based Tool for Interactive Prototyping of Highly Interactive Applications. 12th IEEE, International Workshop on Rapid System Prototyping ; Monterey (USA). IEEE ; 2001.

Navarre D., Palanque P., Bastide R. & Sy O. Structuring interactive systems specifications for executability and prototypability. 7th Eurographics workshop on Design, Specification and Verification of Interactive Systems, DSV-IS'2000; Springer Verlag LNCS. n° 1946. 2000.

Norman, D. *The Psychology of Everyday Things*. New York:Doubleday, 1988.

Norman, D.A. Cognitive Engineering. In D. Norman and S. Draper (eds.) *User-Centered Systems Design*. Lawrence Erlbaum and Associates. Hillsdale, NJ. pp.31-61. 1986.

Ortony, A. Metaphor and Thought, 2nd Edition. Cambridge: Cambridge University Press. 1993.

Pane, J.F., Ratanamahatana, C., Myers, B.A. Studying the language and structure in non-programmers' solutions to programming problems. International Journal of Human-Computer Studies, Academic Press Vol 54, No.2, pp.237–264. 2001.

Peirce, C.S. *Collected Papers*. Cambridge, Ma. Harvard University Press. (excerpted in Buchler, Justus, ed., Philosophical Writings of Peirce, New York: Dover, 1955). 1931.

Repenning, A., Perrone, C. Programming by Analogous Examples. Henry Lieberman (ed.) Your Wish is My Command: Programming by Example. San Francisco, CA: Morgan Kaufmann Publishers, pp.351–369. 2001.

End user Development by Tailoring

Blurring the border between Use and Development

Yvonne Dittrich, Lars Lundberg and Olle Lindeberg

Blekinge Institute of Technology, Dept. of Software Engineering and Computer Science

{yvonne.dittrich, lars.lundberg, olle.lindeberg}@bth.se

1 Introduction

With the development of distributed and networked systems, single programs become more and more part of a computer infrastructure, supporting not longer isolated tasks but interdependent work and business practices. This puts new requirements on software. Software has to adapt to changes in the technical environment, in the business and in the organization it supports, and in the work practices of the people using the software. Flexible software provides one answer to this pressure for change. If a specific kind of changes can be anticipated, part of the software can be designed so that the users can adapt the software to changing requirements. This part or aspect of the software system can then be further developed by users. Tailorable software provides possibilities for domain or application specific End-User Development.

In this position paper we report experiences from two cases: the Billing Gateway, a system that sorts and distributes call data records produced by phone calls to billing systems, statistics and fraud detection, and a back office system of a telecommunication provider, administrating contracts and computing payments based on certain events. The phenomenon, however, is not restricted to the telecommunication area. Similar requirements for adaptability can be observed for municipal information systems [1], computer supported co-operative work [9, 10], or in general for emergent organizations [11].

Comparing our cases, we see that the technical possibilities blur the border between use and development and challenge the traditional classification of software practices. Tailorable software trades the complexity of adding possibilities for end-user development against easy maintenance, no further development by software engineers will be needed (for changes within the tailoring capabilities).

2 Experiences

This section reports experiences from two different projects. Each of them is related to a sharp industrial application. In each of the projects we experimented with different solutions. Requirements from work and business contexts as well as from the technical context of the applications guided the evaluation of the respective prototypical solutions. In each case a specific solution optimizes the deployment of available technology according to the situated requirements and constraints. These solutions raise a set of HCI issues that will be discussed in the following section.

2.1 Flexibility in Large Telecommunication Systems

This case is from a research project focusing on performance aspects of multithreaded large telecommunication systems. The need for customization after delivery is increasing in many performance demanding real time systems. An example is the Billing Gateway that function as a mediation device connecting network elements with post processing systems like billing systems, statistical analysis and fraud detection. It contains an interface that allows tailoring of the filters and formatters that sort and re-format the incoming call data records to the interfaces of the post processing systems. The tailoring of the filters and formatters is done with help of a special purpose language that is then interpreted by the system. This interpretation turned out to be a performance bottleneck for the multiprocessor system, as it was dynamically allocating space using a common heap. Implementing a compiled solution solved the problem. [6] Here flexibility could be kept while a good solution for the performance problems was developed.

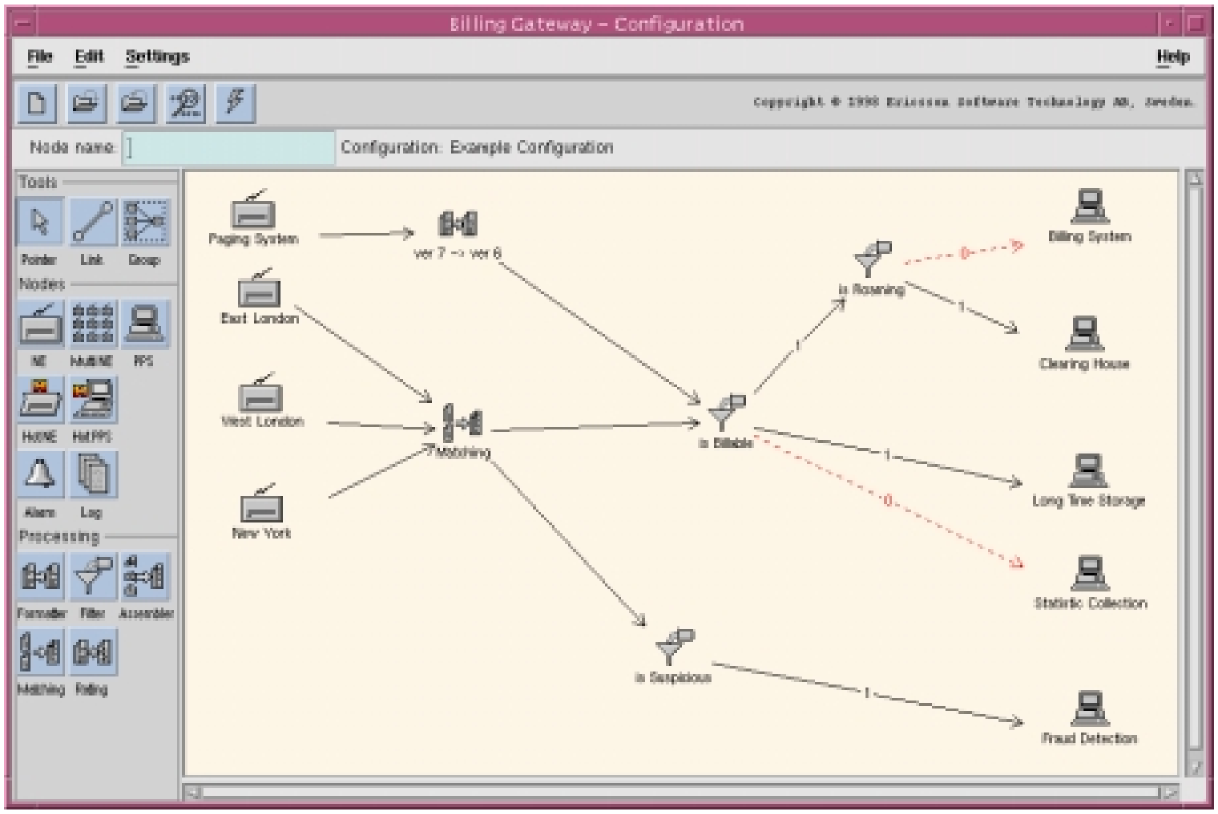


Figure 1: The Billing Gateway configuration view

2.2 Design for Change

In this chapter we report from a research co-operation with a telecommunication provider and a small software developing company around the development of a software system for a rapidly changing business area.[1] Providing mobile communication is a competitive and rapidly changing business. The application that is subject to the research co-operation is a system administrating certain payments. The system computes the payments based on contracts. They are triggered by events.[2] With the former application only payments based on a certain event could be handled automatically. The business practice requires payments based on other events as well as new contract types. Other aspects of the computation, that today are hard coded should be subject to manipulation also.

[1] The project is funded to 50% by the industrial partners and to 50% by KKS (The Knowledge Foundation). For more detailed information see [2].

[2] To protect the business interest of our industrial partner, we do not tell about the character of contracts.

The existing software has turned out to be too cumbersome to change. Beside specific restrictions in the interface the adaptation of today's program to new types of contracts and payments is not possible. They have to be handled manually. Implementing a tailorable solution seemed a promising idea. With the help of prototypes we explored different implementation possibilities. The program that now is used in the company represents a different solution. First a conceptual model of the contract handler is provided. Then two design solutions are presented.

The system can be regarded as two loosely connected parts (Figure 1): the transaction handler and the contract handler. The transaction-handler application handles the actual payments and also produces reports while its database stores data about the triggering events, payments and historical data about past payments. (1)[3] The data describing the triggering events is periodically imported from another system. (2) To compute the payments, the transaction handler calls a stored procedure in the contract handler's database. (3) The event is matched with the contracts; several hits may occur. Some of the contracts cancel others; some are paid out in parallel. We call the process of deciding which contracts to pay 'prioritization'. (4) The result is returned to the transaction handler. (5) Payment is made by sending a file to the economic system.

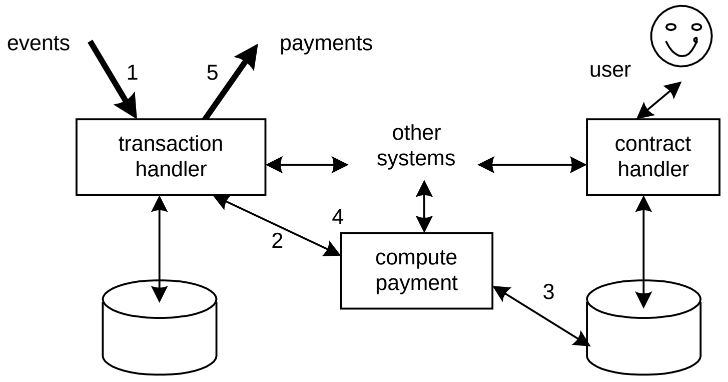


Figure 1

In order to make the system adaptable for future changes a conceptual model that facilitates a meta-model description of the system is needed. We first noted that a condition is meaningful in a contract only if the transaction handler can evaluate it when payment is due. This leads to the concept of event types; a payment is triggered by an event and all contract types belong to a particular event type. Each event type has a set of attributes associated with it that limit what a contract for such events can be based on. Contract types that a handled similarly are put together in one group. Secondly, we split up the computation of payments into two consecutive parts: first find all matching contracts and thereafter select which to pay (prioritization).

Flexibility Light

The design of the finally implemented contract handler incorporates some meta-modeling features while using a normal relational database. The result was a flexible system without using any complex or nonstandard software. The flexibility comes primary from three features in the design. The first is to use a non-normalized database. The contract types all have different parameters but they where anyway all stored in the same database table which had fields for all parameters in any contract. This made a sparse table wasting some disc space. The second feature was to group the contract types into groups. In most

[3] The numbers refer to figure 1.

cases the program could handle all contracts belonging to the same group in an uniform way, simplifying the program.

The third feature was to use the object oriented capabilities in PowerBuilder which was used to build the graphical user interface. The user interface is constructed with one window for each contract-group type. The windows were built as sets of interface objects, each taking care of one or, occasionally, a few parameters. Since the parameters are treated in the same way in all contracts, this reduces the effort required to construct the interfaces and facilitates addition of new ones. The interface objects also guarantee that the user interface handles parameters in a consistent way.

The design makes it easy to add new contract types to the system. Some changes can be done directly by tailoring in an administrator interface. Most changes will also need some programming but with the system structured as it is the programming needed will be small and simple.

The design combines different techniques for implementing flexibility. When regarding the specific situation with respect to use, operation and maintenance of the system the overall evaluation was that design fitted well with the specific contexts of use and development at the telecommunication provider. [4]

Why not using Meta Object Protocol

Inspired by the concept of meta-object protocols, we implemented a prototype that uses reflective attributes of Java. [5] We wanted to gain an understanding of the complexities related to this approach. The prototype does not implement the whole system but only a part of the contract handler application.

The prototype is divided into two levels, the meta-level and the base-level. Two catalogues, one storing contract type and the other parameter classes implement the connection between the two levels. In the meta-level of the prototype, the new contract types are created and stored in the contract type catalogue. In the base-level the same classes are used as part of the program. The parameter class catalogue is used by the meta-level to know which parameters exist and by the base-level as part of the program.

The metaobject protocol prototype can be implemented with a traditional, sparsely populated database or with a database system that allows for changing the data model during runtime.

3 What if maintenance becomes use?

In the cases above tailoring features were implemented to allow end-users to adapt and further develop the existing application to fit evolving requirements. Tasks that would otherwise require changes in the source code implemented by software engineers from the company or unit that developed now in principle could be done within the user community. Requirements for maintainability partly might become requirements regarding the usability of the tailoring aspect of the application. This raises a set of Human Computer Interaction issues:

How to decide on what part to make tailorable?

In both cases the development organization had built similar software before. Experiences had shown which aspect of the application domain is likely to change. Familiarity with the application domain contributed to a good estimation of for which part of the software new requirements might evolve. In the contract-handler project future users and business experts were involved throughout the whole project. This matches with experiences regarding the development of flexible middleware: Experiences from use (in this case application programmers) give indication regarding what aspect of the system open up for adaptation.

Designing a tailoring language

As normal interfaces, tailoring interfaces have to be understandable from a users' perspective. They have to represent the computational possibilities not only in a way that makes them accessible for use but helps the user to understand, how to combine them. That also implies that at least the tailorable aspects of the software have to be designed - even on the architecture level – that matches with a use perspective on the domain. The presentation of the building blocks and the possible connections between has to be presented in a comprehensible way as well. Mørch's application units [7, 8] and Stiemerling et al's component based approach [9] are examples for such architecture concepts.

In the billing gateway interface partly provides a very intuitive interface from the user's point of view. The language for tailoring filters and formatters relates well to the technical education of its users. Nonetheless, end-users have shown some reluctance to tailor the application. The contract handler did not address that question, as it became clear quite early that the users were reluctant to change the system. In the latter case the users seemed to feel insecure regarding the correctness of the results of the adaptation.

The challenge is to finding ways to structure the tailoring capabilities of the application so that it is both easy to implement and easy to understand for the user. In the cases we have shown this was no problem the structures was natural from both perspectives. More investigations are needed for to se if this is the normal case or an exception.

Reliability and testing

Personal performance tools like editors, work processors or even search tools in CSCW applications are relatively save environments for tailoring. Errors just affect the outcome of the own work. In the cases above, the tailoring effects billing data respectively payments. If these kind of production systems have to be adapted, it requires a system test. That means a tailoring interface has to include testing facilities as well. Results from test automation might be adaptable. But then again: how to design a test tool so that non-computer scientists are able to make sense of it.

4 References

[1] Dittrich, Y., Eriksén, S. and Hansson, C. PD in the Wild; Evolving Practices of Design in Use. Accepted for the Participatory Design Conference 2002.

[2] Dittrich, Y. and Lindeberg, O. Designing for Changing Work and Business Practices. In Patel, N. (ed.) *Evolutionary and Adaptive Information Systems*. IDEA group publishing (forthcoming).

[3] Kiczales, Gregor 1992: "Towards a New Model of Abstraction in the Engineering of Software", in *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, Tama City, Tokyo, November 1992.

[4] Lindeberg, O. and Diestelkamp W. How Much Adaptability do You need? Evaluating Meta-modeling Techniques for Adaptable Special Purpose Systems. In *Proceedings of the Fifth IASTED International Conference on Software Engineering and Applications*, SEA 2001.

[5] Lindeberg, Olle & Eriksson Jeanette & Dittrich, Yvonne 2002: "Using Metaobject Protocol to Implement Tailoring; Possibilities and Problems", in *The 6th World Conference on Integrated Design & Process Technology (IDPT* '02), Pasadena, USA, 2002.

[6] Mejstad, V., Tångby, K.-J. and Lundberg, L. Improving Multiprocessor Performance of a Large Telecommunication System by Replacing Interpretation with Compilation. Accepted for the ???

[7] Mørch, Anders I. 2003:" Tailoring as Collaboration: The Mediating Role of Multiple Representations and Application Units", in *N. Patel: Adaptive Evolutionary Information Systems*. Idea group Inc. 2003.

[8] Mørch, Anders I. & Mehandjiev, Nikolay D. 2000:" Tailoring as Collaboration: The Mediating Role of Multiple Representations and Application Units", in *Computer Supported Work 9:*75-100, Kluwer Academic Publishers.

[9] Stiemerling, Oliver, Kahler, H. & Wulf, V. 1997 How to make software softer- Designing tailorable applications. Proceedings of the Disigning Interactive Systems (DIS) 1997.

[10] Stiemerling, Oliver, & Cremers, Armin B. 1998: "Tailorable component architectures for CSCW-systems" in *Parallel and Distributed Processing, 1998. PDP '98.* Proceedings of the Sixth Euromicro Workshop pp: 302-308, IEEE Comput. Soc.

[11] Truex, D. P., Baskerville, R., & Klein, H. 1999 Growing Systems in Emergent Organisations. *Communications of the ACM* vol. 42, pp. 117-123.

User-Programming of Net-Centric Embedded Control Software

Jens H. Jahnke, Marc d'Entremont, Mike Lavender, Andrew McNair
UNIVERSITY OF VICTORIA
Department of Computer Science
Victoria, B.C, V8W 3P6, Canada
[jens|mdentrem|jstier|amcnair@cs.uvic.ca]

Abstract

The ongoing miniaturization and cost reduction in the sector of electronic hardware has created ample opportunity for equipping private households with inexpensive smart devices for controlling and automating various tasks in our daily lives. Networking technology and standards have an important role in driving this development. The omnipresence of the Internet via phone lines, TV cable, power lines, and wireless channels facilitates ubiquitous networks of smart devices that will significantly change the way we interact with home appliances. Home networking is considered to become one of the fastest growing markets in the area of information technology. However, interoperability and flexibility of embedded devices are key challenges for making "Smart Home" technology accessible for a broad audience. In particular, the software programs that determine the behavior of the smart home must facilitate customizability and extensibility. Unlike industrial applications that are typically engineered by highly skilled programmers, control and automation programs for the smart home should be understandable to laypeople. In this article, we discuss how recent technological progress in the areas of visual programming languages, component software, and connection-based programming can be applied to programming the smart home. Our research is carried out in tight collaboration with a corporate partner in the area of embedded systems.

Keywords
Embedded Software Engineering, End-User Programming, Autonomous Systems, Home Automation, Connection-based Programming, and Component Software.

1. Programming Challenges for the Smart Home

The ongoing miniaturization and cost reduction in the sector of electronic hardware has created ample opportunity for equipping private households with inexpensive smart devices for controlling and automating various tasks in our daily lives. Networking technology and standards play an important role in driving this development. The omnipresence of the Internet via phone lines, TV cable, power lines, and wireless channels facilitates ubiquitous networks of smart devices that will significantly change the way we interact with home appliances. Home networking is considered to become one of the fastest growing markets in the area of information technology. Interoperability and flexibility of embedded devices are key challenges for making "*Smart Home*" technology accessible for a broad audience. An increasing number of connectivity standards for net-centric smart devices have been proposed by companies and industrial consortia such as *HAVi* (Home Audio-Video interoperability), *JetSend* (intelligent service negotiation), *Jini*, and *Bluetooth* (proximity-based wireless networking) [1].

Still, connectivity standards solve only the first part of the integration problem. Connectivity standards deal with the creation of a common *channel* for communicating among various smart appliances. The second part of the problem is to establish a common language so that home appliances can actually understand each other and function in a collaborative manner. In general, this problem of *semantic interoperability* is much harder to solve than the realization of the physical transport channel for data. The main reason for these difficulties is the great *heterogeneity* of home appliances and the large variety of their embedding context. Home appliances cover all aspects of our daily lives including environmental controls, lighting, alarm systems and security, telecommunication, cooking, cleaning, entertainment, etc. There exist a vast number of potential scenarios for integrating such appliances. It is not possible for vendors to foresee all these applications and equip their devices with functionality that enables collaboration with every other device a customer would like to integrate. Consequently, there is the need for customization mechanisms that can be used for integrating different appliances and sensors into a common process that controls the smart home.

Such customization mechanisms can be seen as the "programming language" for the smart home. Primary requirements for such a programming language are *ease of use* and *rapid deployment*. Unlike industrial applications that are typically engineered by highly skilled programmers, control and automation programs for the smart home should be understandable by laypeople. Analogously to other "do-it-yourself" maintenance activities around the home, programs for the smart home should be changeable by third-party service providers as well as the homeowner herself. There is a good chance of achieving this goal because applications in home automation tend to have lower complexity compared to industrial automation systems. Still, traditional programming paradigms like textual programming languages appear inadequate for this purpose. Effective programming mechanisms for the smart home require innovative paradigms that lift programming to a level of abstraction that is similar to plugging in a new stereo or TV set. We will shortly outline three such innovative paradigms in the following section. Then, we will describe an example solution for programming the smart home in Section 3.

2. Enabling Paradigms

In this section, we will shortly introduce three emerging software engineering paradigms that, in combination, have great potential for facilitating the end-user programming of the smart home. These paradigms are *visual programming*, *component-based software construction* and *connection-based programming*.

Visual Programming Languages

The development of visual programming languages (VL) has been driven by the experience that laypersons tend to understand pictures better than plain program text. Today, visual programming languages are often used in combination with textual languages. Moreover, visual languages and software visualization paradigms are increasingly used for increasing human understanding of the existing program code in legacy systems. Apart from considerations about the program *business logic*, the area of visual languages was equally driven by progress in the domain of user interface design and human-computer interaction. This paradigm has been broadly adopted with popular programming tools like Microsoft's Visual Basic or, more recently, IBM's VisualAge for Java. Such visual programming languages typically promote event-driven architectures. This means that the programmer does not explicitly define the control flow, but it is implicitly determined by the occurrence of user interface events, e.g., a mouse click on a button. Both visual programming paradigms, flow-logic diagrams and event-driven user interface designs, are complementary rather than competing approaches. They can be integrated into a holistic solution for visual programming.

Component Software

The idea of component software has its roots in the great success that component-based manufacturing has had in the hardware sector. Component-based software systems are assembled from a number of pre-existing pieces of software called *software components*. Software components should be (re)usable in many different application contexts. Particularly, users should be able to use software components without understanding their internal makeup. Thus, component-oriented software composition provides means for reducing the complexity of software development tasks. The term Commercial-Off-the-Shelf (COTS) component was coined in the mid 90's as a concept for a binary piece of commercial software with a well-defined application programmer's interface and documentation. The component market has gained momentum from the introduction of infrastructure for deploying components in programming languages and operating systems, such as Sun Microsystem's (Enterprise) Java Beans and Microsoft's .NET Framework. Using the component-paradigm for software construction has various benefits: it increases the degree of abstraction during programming, provides proven (error-free) solutions for certain aspects of the application domain, increases productivity, and facilitates maintenance and evolution of software systems.

Connection-based Programming

Traditional software programs have followed the procedure-call paradigm, where the procedure is the central abstraction that is called by a client to accomplish a specific service. Programming in this paradigm requires that the client has intimate knowledge about the procedures (services) provided by the server. However, this kind of knowledge is not present in component software because it is based on components from third parties that were separately developed. That is why component software requires a new programming paradigm called *connection-based programming*. In connection-based programming, connections between pieces of software are not implicitly defined by procedure calls but they are explicitly programmed. Connections represent the glue that binds together interfaces of different software components.

3. User-Programming of Embedded Control Software in Home Automation

In this section, we outline preliminary results of an industrial-driven, collaborative research project carried out between the University of Victoria in B.C. Canada and Intec Automation Inc., a Victoria company in the area of embedded systems. The project is supported by the Advanced Systems Institute of British Columbia and the National Science and Engineering Research Council of Canada.

3.1 Embedded Programming with Visual Components

Over the last few years, Intec has investigated how the visual programming paradigm can be used to facilitate the development of embedded control applications for industrial as well as private applications. As a result, Intec has developed *microCommander*, an application for presenting a visual programming interface to a system of embedded devices (www.microcommander.com). The software runs on a personal computer with access to a network connecting any number of devices. All of the devices must conform to a predefined component architecture that is recognized by microCommander. MicroCommander then allows a user to visually program these off-the-shelf software components without having to write any source code. Behind the scenes, each component consists of an *embedded code*, an *interface and behavior definition*, a *configure dialog*, and *operating dialogs* (Figure 1). The *embedded code* containing the logic that operates a device usually executes on micro controllers located at various places within the automated home. Depending on the complexity of the device, a single micro controller may host the *embedded code* for a number of components. The *interface and behavior definition* describes how to interact with the device in terms of input and output messages. Messaging formats and policies are part of the component architecture, and it is critical that every device in the system strictly conforms to these rules. This way it is guaranteed that every device is addressable and properly controllable.

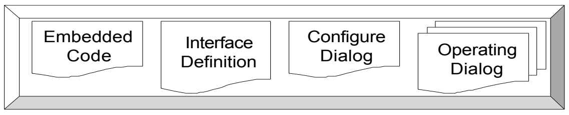


Figure 1 *Aspects of microCommander Component*

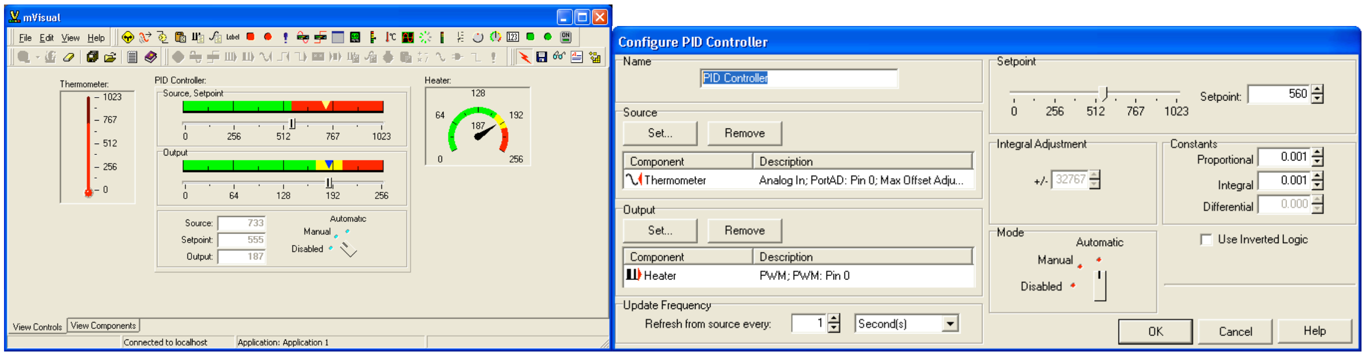
The *configure dialog* (Figure 2, right) is a visual interface for programming properties such as micro controller input and output assignments, default values, states and so forth. This configuration setup is generally done only once during system installation, after which the component is exclusively controlled via the *operating dialogs*. The use of *configure dialogs* requires some domain knowledge and, thus would typically be done by third party vendors during installation. For example, Figure 3 shows the set-up of a new heater system controlled by a PID control component [2].

The *operating dialogs* (Figure 2, left) provide visual interfaces to the devices embedded within the home. Each *operating dialog* is customized towards the day-to-day usage of a device by a layperson. Both *configure dialogs* and *operating dialogs* reside within the microCommader application, and are part of its user interface. The application contains an extensible library of visual controls that the Operating Dialogs may utilize. MicroCommander thus acts as PC-based remote control console to the device allowing a home owner to manipulate and visually program a home from any internet-ready PC running microCommander.

3.2 User-programming of multiple-device interaction

The tools and paradigm outlined so far are geared towards end-user programming and controlling single embedded devices, however they do not provide means for programming automatic collaborations among multiple devices. For this purpose, the University of Victoria and Intec Automation have jointly developed a technology prototype called microSynergy. MicroSynergy facilitates the development and execution of logic described with a subset of the specification and description language (SDL) [3]. Our subset of SDL has unambiguous formal semantics and can easily be understood by a layperson. MicroSynergy consists of a *microSynergy editor* and a *microSynergy runtime engine*. Specifications created using the editor are downloaded to the runtime engine, which then controls the corresponding embedded devices accordingly.

Collaboration logic is implemented in terms of input and output messages rather than visual controls. Editing microSynergy diagrams and establishing connections among the entities on the screen is done by simply clicking and dragging objects using the mouse and keyboard.



Source: 733
Setpoint: 555
Output: 187
Connected to localhost
Application: Application 1

Configure PID Controller

Name: PID Controller

Source

Component	Description
Thermometer	Analog In; PortAD: Pin 0; Max Offset Adju...

Output

Component	Description
Heater	PWM; PWM: Pin 0

Update Frequency
Refresh from source every: 1 Second(s)

Setpoint: 560

Integral Adjustment: +/- 32767

Constants
Proportional 0.001
Integral 0.001
Differential 0.000

Mode: Automatic / Manual / Disabled

Use Inverted Logic

Figure 2 *The configure dialog (right) allows for the composition of components and customization of component parameters. The operation dialog (left) is used for day-to-day use and control of embedded devices via the Internet.*

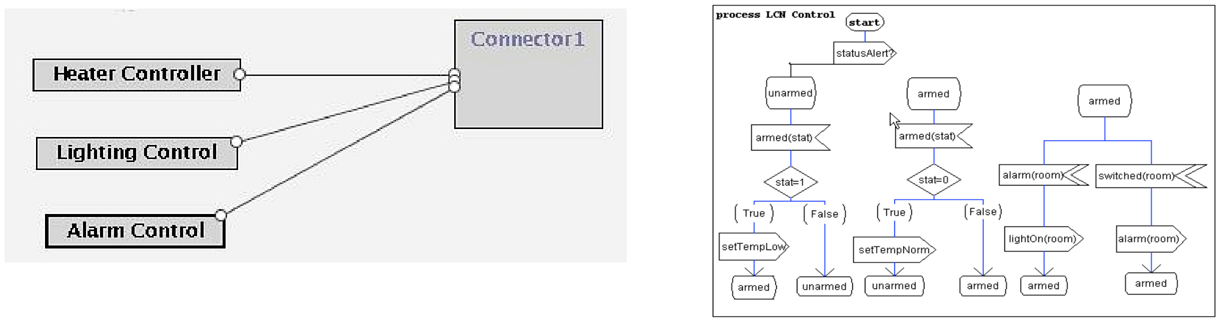


Figure 3 *microSynergy connector integrating three embedded control devices (left)*
microSynergy SDL description that programs the internal behavior of the connector (right)

A home alarm system is an example with the need to establish elaborate logical dependencies in such a way that the events triggered by one device cause a response in another. The home alarm system, for example, once activated ought to trigger the lighting and video surveillance system. It should also automatically deactivate these systems once the threat to the home has passed. This type of scenario is easily programmed using microSynergy. Figure 3 illustrates the visual language of microSynergy. The left side shows the *system view* including all collaborating devices and connector components between them. The right side shows the internals of a sample connector. A more detailed description of the microSynergy language is out of the scope of this paper and can be found in [3,4].

Acknowledgements
We would like to thank Intec Automation for their collaboration with this research project. Furthermore, we thank the Advanced Systems Institute of British Columbia (ASI) for supporting the research. Finally, thanks to Andrew McNair for his support in implementing the microSynergy editor.

References

1. *Jini Technology and Emerging Network Technologies*. 2001, Sun Microsystems. Online at http://www.sun.com/jini/whitepapers/technologies.html.
2. Goodwin, Graebe, Salgado. *Control System Design*. Prentice Hall, 2000.
3. Mitchele-Thiel. *Systems Engineering with SDL*. Wiley, 1997.
4. J. Jahnke, M. d'Entremont, J. Stier. Facilitating the Programming of the Smart Home. IEEE Wireless Communications. Vol. 9, no 6. December 2002.

EUD-Net’s Roadmap to End-User Development

Markus Klann
Fraunhofer Institute for Applied Information Technology (FhG/FIT)
Schloss Birlinghoven
53754 Sankt Augustin, Germany
+49 2241 14 2152
markus.klann@fit.fraunhofer.de

ABSTRACT
The article gives an overview of aspects, current approaches and promising strands of research for the subject of End-User Development (EUD). It is a condensed version of the EUD roadmap which has been produced within the European research project EUD-Net whose aim is to foster research and development in this field. The article concludes that empowering end-users to carry out substantial adaptations of IT-systems is an important contribution to letting them become active members of the information society.

Keywords
End-User Development, co-evolution, requirements engineering, information society

INTRODUCTION

The subject of End-User Development (EUD) is the focus of the ongoing European research project EUD-Net [3, 7]. The project’s definition of EUD is as follows [2]: *“End User Development is a set of activities or techniques that allow people, who are non-professional developers, at some point to create or modify a software artefact.”*

The goal of EUD-Net is to create a joint vision of researchers and industry partners in this field and to provide ideas and guidelines for future research and development. A first step in this process has been to create a roadmap for the field. This roadmap gives an introduction to the topic of EUD, a survey of current approaches, methods and areas of application and points at promising strands of research.

In order to make EUD-Net’s ongoing work available to a wider audience, this article provides a condensed view of the central aspects presented in EUD-Net’s roadmap.

WHAT IS EUD AND WHY IS IT IMPORTANT?

People want IT-systems to meet their requirements. Capturing these requirements and letting software-professionals implement them is a workable approach only if the requirements can be identified and remain stable over time. Very much in contrast to this the current development in professional life, education and also in leisure time is characterized by an increasing amount of change and diversity. Changes in work and business practices, changes concerning individual qualifications and preferences and changes in the dynamic environment, organizations and individuals act in. Diversity concerning people with different skills, knowledge, cultural background and physical or cognitive abilities, as well as diversity related to different tasks, contexts and areas of work. As most of the work done in organizations, and an increasing amount of peoples’ activities outside of organizations is supported by IT-systems, there is a need for substantially more flexible systems that can easily be adapted to meet the changing and diversified requirements.

This insight, which developed in various fields of human-computer-interaction (HCI) and software-engineering, has now become focused in the new research paradigm of End-User Development (EUD). The goal of EUD is to empower end-users to adapt IT-systems themselves as much as possible, thus letting them become the initiators of a fast, cheap and tight co-evolution between themselves and the systems they are using. To allow for this level of end-user development, IT-systems must be made considerably more flexible and they must support the demanding task of EUD in various ways: they must be easy to use, to teach, understand, and learn. Also, users should find it easy to test and assess their EUD activities.

EUD has now found its first widespread use in commercial software, and end-users have taken it up with some success: recording macros in word processors, setting up spreadsheets for calculations and defining e-mail-filters. While these applications only realize a fraction of the EUD potential and still involve many issues, they illustrate why empowering end-users to develop the systems they are using is an important contribution to letting them become active citizens of the information society. One example for future use of EUD technology is the field of home appliances, i.e., all sorts of electronic devices that people will use at home and that will become interconnected and very flexible in the near future. This creates a mass-market where people will want to adapt systems to their specific contexts and requirements and where they will value personalized, adaptive and anticipatory systems.

Given estimates like that of Brad Myers [6] (Carnegie-Mellon-University) that in 2005 there will be 55 million end-users doing EUD while there will be only 2.75 million software professionals, the importance of research on EUD becomes apparent. Not only to limit the damage caused by erroneous EUD activities but also to fully exploit the potential benefits of quick and precise system adaptations that only end-users can perform at a reasonable cost.

ASPECTS OF END-USER DEVELOPMENT

Enhancing user-participation in the initial design process of IT-systems is one step towards better capturing user requirements. Research is done on providing domain-specific, possibly graphical modeling languages that users find easy to express their requirements in. Such modeling languages are considered an important means to bridge the 'communicational gap' between the technical view of the software professionals and the domain-expert view of the end-users.

But as stated above, end-user requirements are increasingly diversified and changing and even at a given point in time they may be difficult to specify. Consequently, an initial design tends to become outdated or insufficient fairly quickly. Going through conventional development cycles with software-professionals to keep up with evolving end-user requirements would be too slow, time-consuming and expensive. While end-users are generally neither skilled nor interested in adapting their systems at the same level as software professionals, it is very desirable to empower users to continuously adapt their systems at a level of complexity that is appropriate to their individual skills and situation. Challenging the conventional view of 'design-before-use', new approaches try to establish 'design-during-use', leading to a process that can be termed 'evolutionary application development' [5].

System changes during use might be brought about by either explicit EUD activities of the end-users or by the system automatically changing itself to better meet its users' requirements. In the first case, the system is called adaptable, whereas in the second, adaptive.

Adaptability in the sense of EUD calls for a system flexibility that allows for adaptations that extend well beyond simple parameterizations, while being substantially easier than (re)programming. More precisely, a system should offer a range of different adaptation levels with increasing complexity and power of expression. This is to ensure that users can do simple adaptations easily and that they only have to accept a proportional increase in complexity for more complicated ones. This property of avoiding big steps in complexity to keep a reasonable trade-off between ease-of-use and expressiveness is what is called the 'gentle slope' of complexity [1, 6]. As an example, a system might offer 3 levels: on the first, the user can set parameters and make selections; on the second the user might integrate existing components into the system; on the third level the user might extend the system by programming new components.

But adapting systems to users during usage does not necessarily require dedicated EUD activities by the user. Adaptive systems monitor their users' behavior and other contextual properties, like the current task or situation and use different approaches, notably from Artificial Intelligence, to automatically adapt themselves. One important approach to increase system adaptivity is to increase this contextual awareness by taking more contextual properties into account and to set up user models to better assess how the users' requirements relate to different contexts.

However, the distinction between system adaptability and adaptivity is not so sharp in practice. Users may want to stay in control of how systems adapt themselves and might have to supply additional information or take certain decisions to support system adaptivity. Conversely, the system might try to preselect the pertinent EUD options for a given context or choose an appropriate level of EUD complexity for the current user and task at hand, thus enhancing adaptability through adaptivity.

Apart from the system, an individual person might also be assisted by other people in its EUD activities. Such collaborative EUD activities [10] within groups of end-users can be supported by repositories for sharing EUD artifacts, as well as recommendation and awareness mechanisms for EUD-artifacts and expertise. It is one goal of current research to understand how to foster the building up of communities of end-user developers in which knowledge and artifacts can effectively be shared.

As for presenting EUD functionality to the end-user it is generally acknowledged that the adaptation interface should be unobtrusive so as not to distract user attention from the primary task. At the same time, the cognitive load of switching from using to adapting should be as low as possible. There seems to be a consensus that the adaptation functionality should be made available as an extension to the existing user interface.

Finally, the described level of system adaptability requires highly flexible software architectures. Various approaches exist, ranging from simple parameters, rules and constraints to changeable descriptions of system behavior [8] and component-based architectures [10]. A key feature of the more advanced architectures is to allow for substantial changes during run-time, i.e., without having to stop and restart or even rebuild the system.

Practical Implications

Understandably, industry players interested in EUD are looking for practical applicability and fast deployment, while not being enthusiastic about major changes to their development processes. This must be taken care of by integrating EUD with existing development practices. Nonetheless, finding the right processes and organizational structure for EUD development and making appropriate changes will still be necessary. To this end, results from EUD research must be validated in real-world projects within the industry and the acquired experience must effectively be disseminated in adequate communities within industry and research.

This concerns the costs of providing EUD systems and, for example, whether there is a market for selling software components that can be used and adapted in EUD systems. Competition between various component vendors may cause interoperability issues when they choose to add proprietary extensions to their components to defend or extend their market share. This has not been uncommon in the software industry and as it constitutes a serious threat to a widespread success of EUD, industrial standardization efforts will be very important.

RESEARCH ON EUD

There are a fairly large number of research fields pertinent to the subject of End User Development. A selection of the most important ones is presented below, while such fields as supportive technology (e.g. development tools) or quality assurance for EUD (e.g. simulation environments, undo mechanisms) had to be left out because of space constraints. While there is not yet a stable and well-established classification of the field of EUD, first proposals have been presented in [2] and [9].

Understanding people as EUDs

As noted above, people adapting IT-systems are at the center of EUD research. Individuals carrying out EUD operations have to invest time and attention that they would normally focus on the task at hand. While being responsible for their operations they run the risk of committing errors. Accordingly, research on EUD has to provide the means for end-users to understand the consequences of their EUD operations, carry them out as safely as possible, and to exercise an appropriate level of control. Also, end-users must be motivated to pay the (cognitive) cost of performing EUD operations. To this end, EUD research has to find ways of keeping these costs at a minimum, to make operations intuitive, to provide assistance and to make the benefits transparent and assessable. Another issue to be resolved is that EUD beyond a certain level of complexity will require people to acquire additional skills beforehand, which they will have to be willing to do. Finally, doing EUD in collaboration with other people will involve new communication and work processes, as well as privacy issues, requiring appropriate solutions.

Organizational environment

EUD-systems must be properly embedded into their organizational environment to be interoperable with existing IT-systems to fully exploit the benefit of widespread EUD activities within the organization and to motivate end-users to actually carry out such activities. Conversely, EUD will have an impact on organizational structure and processes, allowing faster and more precise adaptations of IT-systems to support, for example, the setting up of project-specific team structures and collaborative processes. Research is needed to determine how organizations must change to exploit the full potential of EUD for becoming more flexible and powerful.

Interfaces

As EUD wants to empower end-users to perform substantial modifications to IT-systems, while not hampering them in their every-day work, extending user-interfaces with EUD-functionality is as important as it is difficult. Users must be able to understand and assess the existing system and to specify and test their own EUD operations. Therefore, representational formats must be devised that are especially suitable for end-users, keeping them from making errors typical of conventional programming languages. Research is necessary on creating and evaluating domain-specific and graphical (2D and 3D) formats. Interfaces should proactively assist the users to explore and understand the systems and to create and annotate new EUD artifacts. To this end, various interesting approaches exist, like ‘interactive microworlds’, zoomable multi-scale interfaces, tangible user-interfaces (TUIs), AR, etc. Another requirement is that EUD functionality has to be presented as unobtrusively as possible and only when needed, so as to deviate as little of the users’ attention as possible from their primary task.

Generally speaking, interfaces and representational formats play an important role in mediating communication processes between different actors (e.g. software professionals and end-users) during initial system design as well as between groups of end-users during collaborative EUD activities.

Context-awareness

As noted above, interfaces should provide users only with such an amount of EUD-functionality that is appropriate to their current context. In particular, for normal use requiring no adaptations the interfaces should generally provide no EUD-functionality at all. Moreover, systems should proactively assist their users by adapting themselves automatically if sufficient information is available, or at least generating suggestions for partial solutions for the users to choose from. In order to do this, research is needed on how systems can build up a knowledge base by monitoring their environment (e.g. user, task, place, time) and on how this context-awareness can be turned into adaptive system behavior. One promising approach is to investigate how an EUD-system might build up a history of its own use and of EUD operations it has been subject to in order to generate suggestions for future EUD operations in similar situations.

Architectures

In order to have IT-systems that are changeable at run-time while remaining maintainable and interoperable with other systems, it is quite obviously crucial to have appropriate software architectures. Loose coupling between software components through well-defined general interfaces is a promising approach. One challenge here is to combine general interfaces which may not be very intuitive for end-users with domain-specific components which users know how to handle within their domain of expertise. Another promising approach is to add a model-layer to the architecture of IT-systems allowing for a relatively easy modification of the underlying system. A similar approach is not to build the system behavior into the actual architecture and implementation, but to separate it into a sort of meta-description which the system interprets during run-time. Finally, in order to be able to make the current system-status understandable and to let end-users assess the consequences of their operations the architectures for EUD must allow for reflexivity and inspection.

Issues and Trade-offs

Enabling end-users to substantially alter IT-systems creates a number of obvious issues concerning correctness and consistency, security and privacy. One approach to handle these issues is to let the system monitor and maintain a set of desired system properties during EUD, like integrity and consistency by, for example, allowing only safe operations. But as H. Lieberman (Massachusetts Institute of Technology) points out [4], user errors and incompleteness of information cannot be ruled out altogether, whereas users may often be able to supply missing information or correct errors if properly notified. For this reason, handling the issues above may often best be done by a cooperation of both user and system. Another issue of EUD is how to make users aware of existing EUD functions and how to make these functions easily accessible.

Finally, EUD research must find good solutions for a number of trade-offs created by empowering end-users to carry out substantial adaptations at a complexity-level no higher than needed for the task at hand. These trade-offs exist between expressiveness, freedom, and being general-purpose on the one hand and usability, learnability, control, and being domain-specific on the other.

CONCLUSION

EUD can be seen as an important contribution to create a user-friendly information society where people will be able to easily access information specific to their current context and to their cognitive and physical abilities or disabilities. People will have access to adapt IT-systems to their individual requirements and the design of IT-systems can be made more socially acceptable by collaboratively involving all actors. Apart from empowering individuals to take part in design processes, EUD can also support communities by letting them share experience on how to adapt IT-systems. In particular, communities might share EUD artifacts by way of repositories for reusable and potentially domain-specific components. These repositories will help people in choosing and assembling components appropriate for their requirements by making available the explanations, recommendations and critique of their peers.

On the economic side, EUD has the potential to enhance productivity and create a competitive advantage by empowering employees to quickly and continuously adapt IT-systems to their specific business requirements.

In EUD research much needs to be done, notably to conduct empirical research, to develop a sound theoretical basis and last but not least to establish a consistent and stable terminology. Suggestions on concrete research and development activities for EUD are currently being developed within EUD-Net and will be made available as a research agenda for the field.

ACKNOWLEDGMENTS

This article owes very much to the input of the members of EUD-Net [3]. I'd like to thank all members of EUD-Net for valuable documents and discussions. Funding of EUD-Net is provided by the European Commission.

REFERENCES

1. Blackwell, A. *User priorities for EUD: Review and research agenda.* Presentation at first EUD-Net workshop in Pisa, Italy, 23./24. Sep. 2002. Available at [3].
2. Costabile, M. F. *End-User Development - Empowering people to flexibly employ advanced information and communication technology*. Report of the 1st EUD-Net workshop at Pisa, Italy, 23./24. Sep. 2002. Available at [3].
3. EUD-Net Network of Excellence. http://giove.cnuce.cnr.it/eud-net.htm
4. Lieberman, H. *Your Wish is My Command: Programming by Example for End-User Development*. Presentation at first EUD-Net workshop in Pisa, Italy, 23./24. Sep. 2002. Available at [3].
5. Mørch, A. *End-user participation in evolutionary development*. Presentation at first EUD-Net workshop in Pisa, Italy, 23./24. Sep. 2002. Available at [3].
6. Myers, B. *Making Programming Easier by Making it More Natural*. Presentation at first EUD-Net workshop in Pisa, Italy, 23./24. Sep. 2002. Available at [3].
7. Paternò, F. *Introduction to the EUD-Net EU Network of excellence.* Presentation at first EUD-Net workshop in Pisa, Italy, 23./24. Sep. 2002. Available at [3].
8. Repenning, A. *End User Development: Who needs it?* Presentation at first EUD-Net workshop in Pisa, Italy, 23./24. Sep. 2002. Available at [3].
9. Sutcliffe, A. *A Comparative Framework for End User Development*. Presentation at first EUD-Net workshop in Pisa, Italy, 23./24. Sep. 2002. Available at [3].
10. Wulf, V., and Won, M. *Supporting End-User Tailoring: Component-Based Approaches.* Presentation at first EUD-Net workshop in Pisa, Italy, 23./24. Sep. 2002. Available at [3]

Objects for Users End-User Development of a Cooperative Information Model

Barbara Kleinen
Institute for Multimedia and Interactive Systems
University of Luebeck,
Media Docks, Willy-Brandt-Allee 31a, 23554 Lübeck
+49 451 2803 4204
kleinen@imis.uni-luebeck.de

ABSTRACT
Object oriented methods started with the vision of making computer power available to all users. This paper introduces an approach in which again malleable objects are the key element of an empowerment strategy. The goal is to offer a cooperation support to virtual teams, which is adaptable to evolving structure and organization of the cooperative work. In the described approach, the common information space offers a generic set of functionality to access (display, alter and store) shared objects, whereas the users define and alter the object structure incrementally to adapt the information model to their needs and goals.

Keywords
tailorable groupware, information modeling, flexible objects, common information space

INTRODUCTION

In this position paper, I'll introduce the approach to end-user development taken within the Cooperation Infrastructure. The Cooperation Infrastructure (CI)[1] is a collaborative platform offering an information space based on a tailorable information model. The CI is part of my dissertation on CSCW and Virtual Teams, as well as the underpinning of a knowledge archive system used to support collocated learning at universities.[2]

Information exchange as well as flexible cooperation support have been identified as key functionalities of groupware systems supporting Virtual Teams. The Cooperation Infrastructure provides a shared information space which allows for user's definition and modeling of its information structure. As the modeling can be done throughout the course of the work with the platform, it allows for an evolutionary refinement of the structure, based on its application and experiences in the work situation.

The approach taken within the Cooperation Infrastructure is rooted in three lines of thought:

First, it builds upon the visions of object-oriented programming, aiming at making computer power accessible to all users by offering them the simple metaphor of everything being an object accessible through a generic set of functions.

Second, it builds upon the idea of meeting facilitation techniques, which support a collection of ideas prior to their categorization or structuring, resulting in a structure that is based upon actual data/experiences and is supported by all participants.

Third, the approach was informed by refactoring techniques known and used in the context of Extreme Programming.

In the following, after motivating the need for flexibility and tailoring in Virtual Teams, I'll describe the tailorability of the information model within the CI in respects to the mentioned traditions.

MOTIVATION: FLEXIBILITY IN VIRTUAL TEAMS

Teams which organize themselves dynamically in order to accomplish a special task (I call Virtual Teams), need corresponding flexible cooperation support. Flexible support has been exploited for process structuring in the field of workflow management (see, for example [1]), by techniques like feature composition (e.g., [2]), as well as for the filing of documents [3]. According to Harrison and Dourish, the ability to change an information space is also an important precondition for the appropriation virtual spaces, turning them into an actual place for cooperation [4]. In contrast to these approaches, the approach taken within the Cooperation Infrastructure aims at flexibility of the information model or domain model.

OBJECT METAPHOR

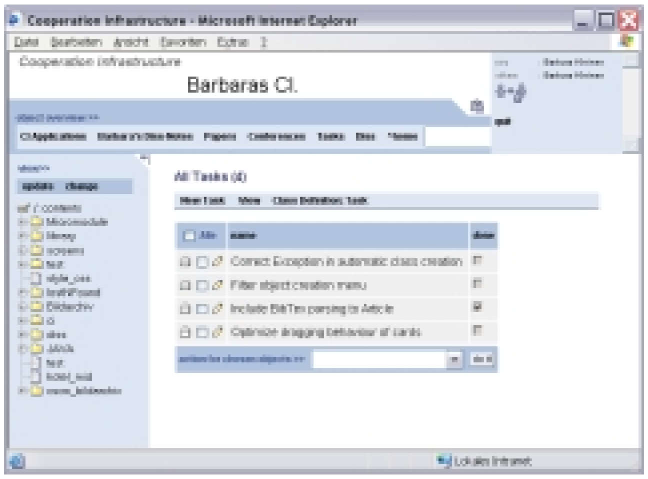
Early implementations of the object metaphor aimed at providing "computer support for the creative spirit in everyone" [5]. They tried to accomplish this by providing a simple and strong metaphor to computer systems, the metaphor of communicating objects. While object oriented methods have proven to be useful for software engineering, they have not been widely adopted to support personal work, neither by end users nor programmers.

[1] http://www.infrastructure.de

[2] http://www.wisspro.de

The basis for the tailorability of the Cooperation Infrastructure lies in the observation that many groupware tools consist of three basic functionalities: persistent storage of objects, their display in collections or in detailed views, and means to create, edit or delete objects. Between different applications, only the type of objects changes; as do the fields (properties) of the objects according to their type. For instance, a message board application would display a collection of messages, listing their sender and subject, a task coordination tool a list of tasks with due dates, committed participants and dependencies on other tasks etc. Accordingly, the Cooperation Infrastructure offers the generic functionality of storing, displaying, and altering shared objects, leaving the definition of the object structure open to the end user. As a common information space, these functionalities are accessible to several users simultaneously with common features like conflict management; awareness support etc. Figure 1 shows the display of a collection of objects. The mentioned generic functionalities are accessible via a web interface.

The CI information space is tailorable during the course of its use in three main aspects: New classes of objects may be added, the class fields may be defined and altered using a class definition and – possibly several – views may be defined for the class. Views specify the set of fields displayed in a specific context.


Cooperation Infrastructure - Microsoft Internet Explorer
Cooperation Infrastructure
Barbaras CI.
All Tasks (4)
Lokales Intranet

name	done
Correct Exception in automatic class creation	☐
Filter object creation menu	☐
Include BibTex parsing to Article	☑
Optimize dragging behaviour of cards	☐

Figure 1: Display of a collection of objects

As class definitions are themselves objects in the CI, they can be created, altered and displayed using the generic functionality for handling objects. Furthermore, changes to class definitions can be made during runtime taking immediate effect. Class definitions are global to the application and therefore immediately visible to all users (depending on access rights) and are thus group-wide adaptations. Individual tailoring can be achieved by subclassing or definition of individual views.

The generic functions enable users to immediately start working with shared objects, without the need for any programming beforehand. To support an evolutionary development of the classes, objects *may hold arbitrary fields*, independent of their class definition, supporting the pattern of finding a structure based on collected information, as will be described in the next section.

COLLABORATIVE INTERPRETATION

Computers usually impose a need for prestructuring information, most prominent in the definition of an information model in an early phase of system development, or the hierarchical organization of file systems [3].

The ubiquitous need to classify information prior to its storage, often hinders the flow of work with an information system. We experienced this with a cooperative hypermedia system[3], were the sheer need to name a new node prior to creating it severely hindered its use for creative work. As Dourish points out, "prestructuring information was a performance hack" [6], at times when computing power was expensive. With the availability of cheap computing power, we should start optimizing for people. According to the prior described experiences, the optimization for users includes the possibility to store something without the need to categorize it [3].

A process of collecting information fragments first and collaboratively finding a structure based on these fragments later has been described in various contexts. It is known in the context of meeting facilitation, using colored paper cards and a pinboard to generate, structure and judge ideas. Similar facilitation and brainstorming methods have also been implemented in electronic meeting systems. Most tools for creative work allow for or foster a creation and collection of ideas without prior judgment, classification or structuring, as an immediate or prior classification would impede the flow of thoughts.

According to Cox and Greenberg, who presented a tool to support a similar group process they call "collaborative interpretation" [7], I name the process of collecting information first and finding a structure for it later "collaborative interpretation".

In several other contexts, processes of alternating collecting and structuring are described. Wenger described the duality and interplay of participation and reification as a fundamental process to produce meaning in Communities of Practice [8].

Collecting Information Fragments

Inspired by the above mentioned methods for group facilitation and creative work, allowing the input of information prior to any structuring or even naming has been the main design principle of the Cooperation Infrastructure.

Collecting information may occur as part of a dedicated effort like a brainstorming, or in the course of ordinary

[3] HyperCom, developed at former Daimler Benz Research in Berlin.

work with the system, by adding or altering objects in the information space, like new task descriptions or messages.

Accordingly, the Cooperation Infrastructure supports the collection of information fragments in two ways:

1. With a creative work and facilitation support tool with the visual metaphor of colored cards pinned to a pinboard, called "Linked Sketches" (Figure 2.) The cards can be spatially structured just like the paper cards. The Linked Sketches tool is similar to PReSS [7] with the main difference of being integrated with the hypermedia information space of the Cooperation Infrastructure, and allowing for visual and hypertext links between the cards and to other objects.
2. Adding and editing objects. The creation of objects is highly independent from any predefined structure, as new fields may be added to instances independent of their class definition (Figure 3). New fields may even remain unnamed. In the above mentioned example of a task management application, a member may spontaneously add "This took me 5 hours!!!" during editing the task to mark its completion.

Reflection and Structuring of Collected Information Fragments

To reflect the usage of classes (all tasks, all messages) the Cooperation Infrastructure shows a list of all instances in a table (table of instances). Using this table, the group can reflect on their usage of the class by examining which fields they actually used and which fields were added to instances. As the table also shows the content of the fields, names for yet unnamed fields may be found collaboratively, creating a naming that is understood by the whole group. The class definition can then be altered by the group according to the reflection on actual use of instances of the class.

Note that unnamed fields may be of value for the cooperative work, as the spontaneously added remark "This took me 5 hours" is immediately accessible to all users reading the task description. In other examples, the addition of a new, unnamed field to a single instance may be of little or not the intended use, as if, for example, someone adds "read this immediately" to a message in the message board. Therefore, as a further tailoring, specialized views may be defined for classes, defining which fields are actually shown in the overview and detailed view. After naming the "read this immediately" field and including it in the collection view definition, messages are listed with their respective priority, now being of practical use to the participants.

The reflection on class usage may also reveal inconsistencies, like inconsistent field names or duplicate objects. Resolving those depends on the ability of changing the information structure without loosing data, which I call refactoring.

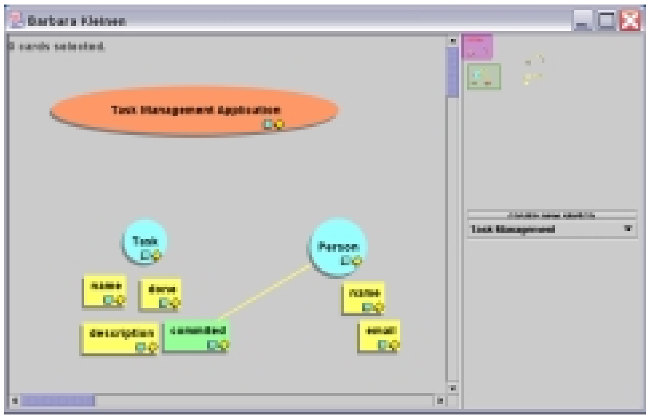
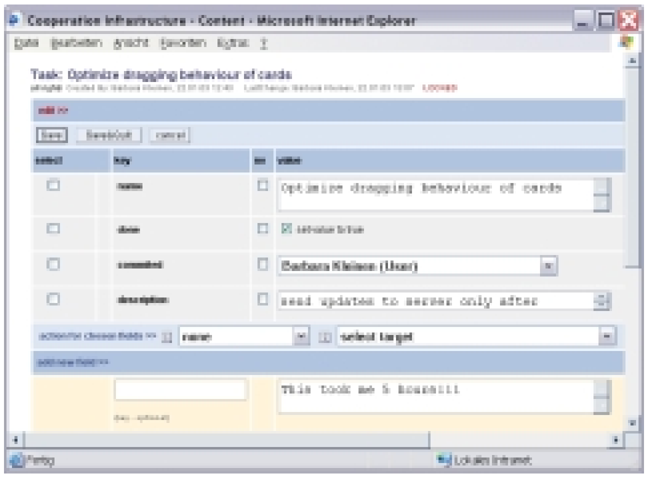


Figure 2: Facilitation Tool "Linked Sketches"


Cooperation Infrastructure - Content - Microsoft Internet Explorer
Datei Bearbeiten Ansicht Favoriten Extras ?
Save
Save&Quit
cancel
select
key
value
Fertig
Lokales Intranet

Task: Optimize dragging behaviour of cards

key	value
name	Optimize dragging behaviour of cards
done	[illegible]
committed	Barbara Kleinen (User)
description	read updates to server only after
	This took me 5 hours!!!

Figure 3: Addition of new fields to an object

REFACTORING: RESTRUCTURING OF INFORMATION

Refactoring denotes the change of a program's architecture without modifying its behavior or functionality [9], which may be a necessary step in agile development process to accommodate the implementation of new requirements. Within the context of the Cooperation Infrastructure, I use the term *refactoring* for the reorganization of the information model without loosing the collected data. Typical refactorings are:

- renaming of fields
- merging of objects
- movement of fields between objects
- reclassification of objects

All but the last refactoring can be applied to instances as well as classes. Changes on the class level trigger corresponding changes of all instances, to preserve the collected information and integrate it into the new information structure.

Refactorings are conducted using the generic object editor, or, in more complex cases, the group facilitation tool. Figure 2 shows a modeling of a task management application.

Using these refactorings; late changes to the information model are possible, reflecting and incorporating already collected data.

INFORMATION MODELING WITHIN THE CI

Using the described techniques, the information model is evolutionary developed in the following way:

In an initial meeting a first draft of the information model is defined. Often predefined applications can be used, e.g. a message board application or a task and file management application. The definition of the classes has to be done by someone who is already familiar with the Cooperation Infrastructure. This might be a member of the team who has worked with the CI before, or an external facilitator. After the primary definition of classes the group starts to collaborate using the tool, creating and editing objects within the predefined structure. Team members unfamiliar with the CI usually find it easy to start using the CI at this point. During this phase, the information model may be changed partially by adding new fields to object instances. At a convenient time – depending on the collaborative setting – the group meets and reflects on their usage of the infrastructure. Classes may then be restructured and refined based on the collected data. Through the discussion of the structure within the group, reflecting and incorporating all occurred uses, the team arrives at an information model understood and supported by all team members.

EXPERIENCES AND DISCUSSION

Until now, the Cooperation Infrastructure was used in several research projects and courses at out institute. Experiences show that it is well possible to create a specialized information model containing 2-3 new classes during a first group meeting, leading to immediate benefit to the group members. Even novice users had no problems using the specialized model by navigation the CI and creating new objects. Some users were confused by the input areas for new fields (Figure 3) as they felt obliged to put something there but did not know what. Consequently, the interface was adapted for some user groups, hiding the altering of structure (new fields, links to class and view definitions) altogether. In these cases, reflection of the class usage only included the usage of predefined fields.

Our experiences showed that the adaptation of the information model led to a quite specialized cooperation support. In some cases, a functional extension was necessary, which was added using the Java /JSP API offered by the CI.

But even without functional extensions, the extension of the information model leads to an immediate benefit to the group. This information model, which is based on actual experiences within the group, may furthermore inform the development of new functionality and ease the communication between users and developers.

CONCLUSION

In this position paper, an approach of a flexible cooperation support, the Cooperation Infrastructure (CI) is presented. In the CI, users can model the information structure during the course of their work with the system. Users may reflect on their usage of the information model and change it according to their needs.

While the CI does not support the addition of new functionality by end users, users can create applications by adding new objects and class definitions, using a predefined set of generic functionality on objects, like storing, displaying and altering them.

The described tension between designing for smooth usage by experienced users and creating comprehensible interfaces for novice users resulted in a division of the interfaces. Further work will concentrate on the issue of providing specialized interfaces and easing the transition from object usage to expanding and altering the information structure.

REFERENCES

1. Divitini, M.; Simone C.: *Supporting Different Dimensions of Adaptability in Workflow Modeling*. Journal of Computer Supported Cooperative Work Vol. 9: 365-397, 2000.
2. Teege, G.: *Users as Composers: Parts and Features as a Basis for Tailorability in CSCW Systems.* Journal of Computer Supported Cooperative Work Vol. 9: 101–122, 2000.
3. Dourish, P.: *The Appropriation of Interactive Technologies: Some Lessons from Placeless Documents.* To appear in: Journal of Computer-Supported Cooperative Work, Vol. 12, 2003
4. Harrison S.; Dourish P.: *Re-Place-ing Space: The Roles of Place and Space in Collaborative Systems.* Proceedings of the 1996 ACM conference on Computer supported cooperative work: 67-76, 1996.
5. Ingalls, D.: *Design Principles behind Smalltalk.* In: BYTE Magazine, August 1981.
6. Dourish, P.: *Information and its Structuring: Problems and Opportunities*. Talk at UCLA, Jan. 25, 2001. Available at http://www.ics.uci.edu/~jpd/
7. Cox D.; Greenberg S. *Supporting collaborative interpretation in distributed Groupware.* In: Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW 2000). Philadelphia: ACM, 289-298, 2000.
8. Wenger, E. 1998. Communities of Practice. Cambridge: Cambridge University Press.
9. Fowler, Martin: Refactoring*: Improving the Design of Existing Code.* Boston: Addison-Wesley, 1999.

Toward a Culture of End-User Programming Understanding Communication about Extending Applications

Cecília Kremer Vieira da Cunha and Clarisse Sieckenius de Souza
Semiotic Engineering Research Group
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de São Vicente 225 - Rio de Janeiro - RJ - 22453-900 - Brazil
+55-21-2512-5984 ext. 123
{ceciliak, clarisse}@tecgraf.puc-rio.br

ABSTRACT
End-User Programming (EUP) research has mainly explored tailoring mechanisms and how to make them accessible to end-users. Complementary, other EUP studies have pointed at the importance of dealing with the collaborative practices related to the process of application extension, whose focus is the social matrix that is the context of application use. Following this direction, we propose the exploration of a research path that directly tackles communication about application extension. The goal is to provide better support to the development of a tailoring culture. As a first step, we present some initial findings on how users go about communicating with designers and suggesting extensions to an application. Further exploration of these initial findings may point out to communicative resources that should be made available in tools and on-line communities developed to support and popularize EUP. Theoretical foundations that can support this research path are also pointed out.

INTRODUCTION
Despite successful reports on the use of End-User Programming (EUP) solutions, this technology has not yet reached widespread adoption [8]. We have not yet achieved an overall "tailoring culture" [10] in which users feel ownership of the software they use and feel in control of changing it and understanding what can be changed. Two routes to make systems more tailorable have been pointed out [10]. One is to make tailoring mechanisms accessible, in which there is ideally a linear and increasing trade-off gradient regarding the effort spent on learning how to use a mechanism and the correspondent tailorability power it provides. Research in EUP has mainly explored this route and several solutions have been proposed [3, 8] ranging from parameter configuration, which requires a low-cognitive effort but usually does not provide high tailorability power, to visual formalisms, which are more powerful but require greater learning efforts. The second route is to approach tailoring as community effort.

This approach is crucial for designers of EUP systems who are willing to foster a tailoring culture, since a culture emerges from socially shared cognition. A culture can be viewed as "a set of thoughts that are shared among group members" [7, p. 258], and this sharing of thoughts and development of common ground [1] strongly depends on communication processes.

Empirical investigations about EUP have shown evidence that these communication processes do happen [11,9,10,12] and have discussed some of their challenges. One of these challenges is what can be called the culture of the worker [10]. It is formed by people who have no interest in computer systems per se, who are focused on their work and have no expectation of tailoring a system or controlling its changes. Thus, they are not in a good position to explore and assess what changes might be possible or how to go about undertaking them. As a consequence, communicating ideas or requirements to cope with an application's limitation is likely to be problematic. They typically do not know exactly what to say or how to say it, or how to make others (e.g. programmers) understand what they need.

Ideally, end-users should have access to the gradient of EUP mechanisms mentioned before and, as they climbed from the simpler and easier extension mechanisms to the more complex and powerful ones, the more articulate and effective they would become in either achieving their extension goals or communicating them to programmers more successfully. But how helpful are the different EUP mechanisms we have today as part of the global representation and communication processes that go on in workplaces? Do the representations they offer support communication within the community that uses it? How can designers of EUP systems better support not only human-computer interaction, but also the development of a culture?

To answer these questions, the EUP research agenda needs to include studies that take a closer look at how end-users communicate about application extensions, broadening the spectrum of EUP analysis with respect of its social scale. This research examines the representations used in communication, provides designers of EUP systems with insights about this process and can ultimately provide a perspective to be explored in the design and evaluation of extensible systems.

As a first step in this direction, we present a summary of a preliminary empirical study we have carried out [5].

EMPIRICAL STUDY
A small group of users was asked to explore a non-extensible application and document, for the application designer, the extensions they would add to it. Communication in paper form was chosen in order to allow

users to reflect about their suggestions before delivering them. Proposing extensions involves creating new meanings and extenders may want to ponder about different alternatives and the impact of introducing an extension to a system. Therefore, asynchronous communication was found more suitable [1]. As a result, users freely expressed themselves and we were analyzed the representations used.

We developed a very simple application in which a user could register the ownership of some material (e.g. book); loan it (recording when it was loaned, to whom, and its return date); register its return; and visualize current loans. Visualization is the only functionality available to both owners and borrowers.

The application was built in a deliberately simplistic form, with only one way of performing these tasks, in order to stimulate users to propose extensions. The application also has a collaborative facet, if one considers that it reflects and therefore impacts on the specific collaborative practice of loaning in this group of users. This facet is beneficial to our purposes because of the crucial importance that communication plays in collaborative settings. It provides a rich environment to indicate which communication processes can emerge in extension tasks, as users tend to use the existing social network to do so [9, 11].

Six users from an academic environment were quickly trained to use the application. They all knew how to build a computer program, although not all of them were experienced programmers. We expected these users to know what extensions and extensible applications are. So, after exploring the application for some time, they were asked to propose extensions they would desire and to document them on paper, knowing their proposals would be read by the application designer. They received a simple form in which they filled out the proposal's author name and brief description. They also received representations of the application that could be used, shown in Figures 1, 2, 3 and 4. Figure 3 shows an abstract representation of the interconnected interfaces that embodied each activity. These representations were offered by the designer just as support material. Participants in the experiment were not required to use them, or even to understand them.

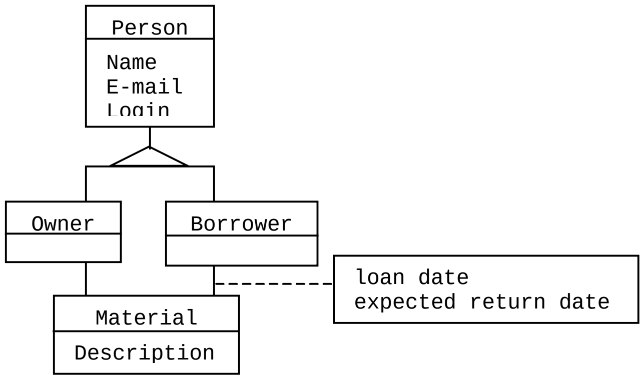


Figure 1. Simplified Conceptual Diagram

We selected these representations because they involve different perspectives of the application as well as different levels of abstraction. We understand our selection is arguable and other representations should be considered in the future, although participants should not be overwhelmed with too many of them.

About an hour later, the participants spontaneously finished their job and we analyzed the material they handed in.

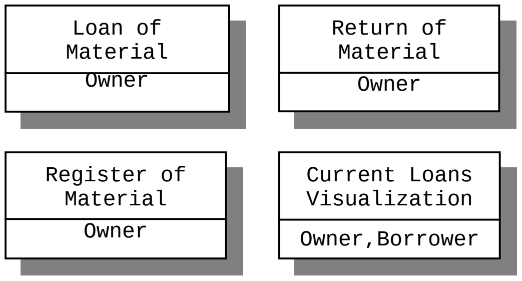


Figure 2. Activity Diagram

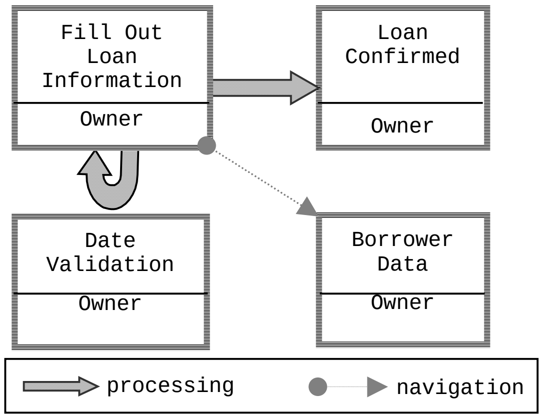


Figure 3. Interface Network of Loan Activity

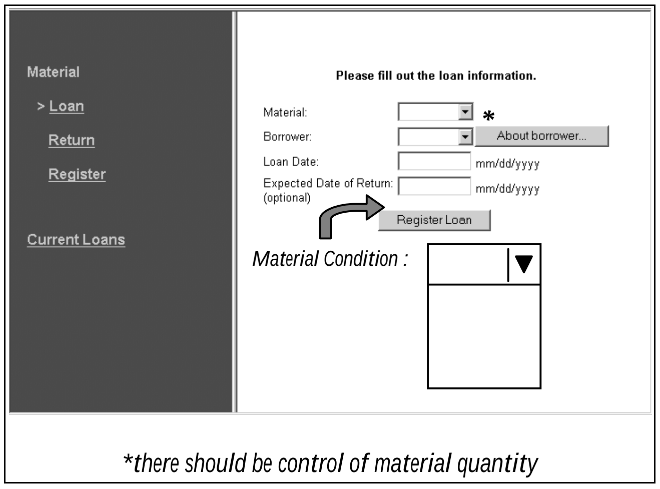


Figure 4. Extension Documentation in a Window Snapshot

Result Analysis

The participants proposed 45 extensions altogether. The extensions' granularity varied considerably, spanning from a button label change to a storyboard of 3 different screens.

All extensions had at least a brief description in natural language (NL), required in the fill-out form. NL was also used to explain and give feedback about the representations; to describe and explain an extension or part of it, its functioning and rules of functioning; to justify the creation of the extension; to make considerations and analogies; and to document synchronous communications with the evaluator during the test. In many cases, NL texts were used within the context of a window snapshot, as depicted in Figure 4. Pointers (e.g. arrows) and markers (e.g. '*') were used to connect interface elements to NL texts describing the extension.

Of the 45 extensions, 17,7% was documented solely in NL texts. All others used from one to three representations in addition to NL, including the representations provided and screen layouts drawn from scratch.

Most extensions (77,7%) were expressed in NL plus 1 or 2 other representations. The distribution of extensions over these representations is as follows (one extension may have been documented in more than one representation):

- Window snapshots: used in 62,2% of the extensions' documentation;
- Screen layouts drawn from scratch: 33,3%;
- Class Diagram: 15,5%;
- Interface Networks: 6,6%; and
- Activity Diagram: 2,2%.

In sum, the extensions' documentation relied more heavily on concrete representations rather than abstract ones.

We have also identified categories of extensions. One extension could fit in more than one category, as follows (parentheses in the end indicate the percentage of extensions that fell into that category):

- Information visualization: users wanted to format information (e.g. table instead of list form), add or hide information, confirmation or error messages. (51,1%)
- Creation of a new class or attribute, such as the borrowers' phone number and loan history. (28,8%)
- Creation of a new function, such as email. (22,2%)
- Data update, achieving higher-order goals (e.g. updating a return date achieves postponing). (13,3%)
- Creation of a new activity, which is different from a new function since it is a high-level functionality, usually a new main menu item (e.g. material reservation, loan transfer or postponing). (11,1%)
- Role change: borrowers were suggested a more active role (e.g. registering loans). (6,6%)
- Extension automatic generation: a participant wanted to add a button to the loaning interface which, when pressed, would add a sub-item to the Loan menu item labeled with the currently selected borrower's name. When triggered, this sub-item would show the loaning interface for the pre-specified borrower. (4,4%)
- Repetitive task automation: the previous extension was also categorized into this one, since it allows users to automate loaning to a frequent borrower (2,2%).

Window snapshots were the most used representation in all categories except for one (creation of a new activity), always immediately followed or introduced by screen layouts drawn from scratch. Not surprisingly, the class diagram was useful mainly for extensions that created a new attribute or class, and the activity diagram was useful for the documentation of new activities. The interface network was used for new functions and data update.

Five of the 6 users used at least one of the non-NL application representations provided. All users used Window snapshots and drew screen layouts from scratch. Four of the 5 users used the more abstract representations: 3 of them used the interface network and the class diagram and one used the activity diagram. One user disregarded all representations and used blank papers to draw storyboards.

In the future, we plan to look for criteria that distinguish users and their preferred representations.

A surprising result was that 2 of the 6 users, who had been exposed to the concept of EUP, demonstrated to be unsure about what extensions were. One of them wrote: "I am not sure whether this is an extension or a new functionality". This shows that bridging the gap from having an idea of what extensions are and actually performing them raises doubts and uncertainty. EUP studies show the importance of having someone within the community to help users bridge such gap, such as translators [9]. Our goal with this work is also to find ways to support this person by examining resources that can be useful for their role. These resources should also be useful in environments of publication and exchange of extensions [14].

CONCLUDING REMARKS AND FUTURE WORKS

We have described some initial findings on how users go about communicating with designers and suggesting extensions to an application. These findings indicate that users will potentially use natural language in conjunction with some other form of more concrete representation, usually existing or new interface layouts that act as referential anchors for their ideas. More abstract representations have been used in specific situations, such as the creation of a new class or attribute within the application domain.

Some of our results may seem similar to previous ones such as [13, 15], but we would like to point out that these studies focused on natural communication about problem solving with the goal of designing more natural EUP languages. The goal of our work is rather to understand how workers, translators and programmers communicate and help each other to master a EUP language and achieve extension. We want to find out how the EUP language integrate with the global representation and communication processes that go on in workplaces and how we can support designers of EUP systems to foster the development of a culture of EUP.A research path that directly tackles communication about application extension may provide better support to the development of a common ground on EUP that can lead to a tailoring culture.

Our preliminary findings demonstrate the potential for such a pursuit, in which guidelines for the development and support of a more widespread use of EUP technology may rise. For instance, from further studies, we may achieve guidelines to the development of tools to assist translators or on-line communities that involve workers, translators and programmers of an extensible application. Guidelines such as the following may emerge: "users prefer to communicate their ideas of extensions along with the original application or its interface snapshots in an editable form, so that they can refer to it, alter it, and compare and contrast their requirements to what has already been built"; or "EUP systems should allow the sharing of incomplete code among different users so they can discuss it" (graceful degradation from completely to incompletely coded extensions to be used as communicative resources has been discussed elsewhere [5]).

For this research path we propose, more empirical and theoretical studies should be undertaken. In order to confirm or revise our initial findings and obtain more useful and reliable results, several other aspects within and outside the scope of this small non-extensible application still need to be investigated. We need to undertake empirical investigations with a different set of participants and in natural settings, as a part of a long-term commitment. This will enable the observation of naturally occurring communication within a user community, involving workers and others. There is also need to observe these phenomena in other application domains.

There are also theoretical foundations to be relied upon. Semiotic Engineering studies the creation and sharing of meanings and signs within the scope of HCI and EUP [2, 15, 4]. Distributed cognition [6] encompasses interactions between people and with resources and materials in the environment. More than that, it looks at how representations in the material world provide opportunities to reorganize the distributed cognitive system, which is the case of extensible applications. Distributed cognition distinguishes at least three kinds of distribution of cognitive processes: across members of a social group; involving coordination between internal and external (material or environmental) structures; and through time in a way that the products of earlier events can transform the nature of later events. EUP studies have tackled distribution across members in a community [11,9,10,12]. We believe these studies can be enriched with a perspective on the coordination between members of a community and the representations they are provided with as (potential) integrants of their culture.

REFERENCES

1. Clark, H. H. and Brennan, S. E. Grounding in Communication. In Resnick, L. B., Levine, J. M. e Teasley, S. D. (eds.) *Socially Shared Cognition*. American Psychological Association. Washington, DC. 1991.
2. de Souza, C.S.; Barbosa, S.D.J.; Silva, S.R.P. Semiotic engineering principles for evaluating end-user programming environments. *Interacting with Computers 13* (2001), 467-495.
3. Cypher, A. (eds.) Watch What I Do: Programming by Demonstration. The MIT Press. 1993.
4. Cunha, C. K. V., de Souza, C. S., Quental, V. S. T. D. B. and Schwabe, D. A Model for Extensible Web-based Information Intensive Task Oriented Systems, in *Proceedings of HCI2000* (2000). Springer-Verlag. 205-219.
5. Cunha, C. K. V. Um Modelo Semiótico dos Processos de Comunicação Relacionados à Atividade de Extensão à Aplicação por Usuários Finais. *Doctoral Dissertation* (2001). Computer Science Department. PUC-Rio. Rio de Janeiro, RJ. Brazil.
6. Hollan, J., Hutchins, E. and Kirsh, D. Distributed Cognition: Toward a New Foundation for Human-Computer Interaction Research. *ACM Transactions on Computer-Human Interaction 7, 2* (2000), 174-196.
7. Levine, J. M. and Moreland, R. L. Culture and Socialization in Work Groups. In Resnick, L. B., Levine, J. M. and Teasley, S. D. (eds.) *Socially Shared Cognition*. American Psychological Association. Washington, DC. 1991.
8. Lieberman, H. (eds.) Your Wish is My Command - Programming by Example. Morgan Kaufmann. 2001.
9. Mackay, W. Patterns of Sharing Customizable Software, in *Proceedings of CSCW'90* (1990). ACM Press, 209-221.
10. MacLean, A., Carter, K., Lövstrand, L. and Moran, T. User-Tailorable Systems: Pressing the Issues with Buttons, in *Proceedings of CHI'90* (1990). ACM Press, 175-182.
11. Nardi, B. A Small Matter of Programming. The MIT Press. 1993.
12. Nardi, B. and Miller, J. Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development. *International Journal of Man-Machine Studies 34* (1990), 161-184.
13. Pane, J. F., Ratanamahatana, C. A. and Myers, B. A. Studying the language and structure in Non-programmers' Solutions to Programming Problems. *International Journal of Man-Machine Studies 54* (2001), 237-264.
14. Repenning, A., and Ambach, J. The Agentsheets Behavior Exchange: Supporting Social Behavior Processing, in *Proceedings of CHI'97* (1997). ACM Press, 26-27.
15. Silva, S. R. P. Um Modelo Semiótico para Programação por Usuários Finais. *Doctoral Dissertation* (2000). Computer Science Department. PUC-Rio. Rio de Janeiro, RJ. Brazil

Feasibility Studies for Programming in Natural Language

Henry Lieberman and Hugo Liu
MIT Media Lab

We think it is time to take another look at an old dream -- that one could program a computer by speaking to it in natural language. Programming in natural language might seem impossible, because it would appear to require complete natural language understanding and dealing with the vagueness of human descriptions of programs. But we think that several developments might now make programming in natural language feasible:

- **Improved language technology**. While complete natural language understanding still remains out of reach, we think that there is a chance that recent improvements in robust broad-coverage parsing, semantically-informed syntactic parsing and chunking, and the successful deployment of natural language command-and-control systems might enable enough partial understanding to get a practical system off the ground.
- **Mixed-initiative dialogue**. We don't expect that a user would simply "read the code aloud". Instead, we believe that the user and the system should have a conversation about the program. The system should try as hard as it can to interpret the what the user chooses to say about the program, and ask then the user about what it doesn't understand, to supply missing information, and to correct misconceptions.
- **Programming by Example**. We'll adopt a show and tell methodology, which combines natural language descriptions with concrete example-based demonstrations. Sometimes it's easier to demonstrate what you want then to describe it in words. The user can tell the system "here's what I want", and the system can verify its understanding with "Is this what you mean?". This will make the system more fail-soft in the case where the language cannot be directly understood.

To assess the feasibility of this project, as a first step, we are studying how non-programming users describe programs in unconstrained natural language. Working with some scenarios from CMU's Natural Programming Project, we are exploring how to design dialogs that help the user make precise their intentions for the program, while constraining them as little as possible.

Using Domain Models for Data Characterization in PBE

José A. Macías and Pablo Castells
Escuela Politécnica Superior, Universidad Autónoma de Madrid
Ctra. de Colmenar Viejo km. 15
28049 Madrid, Spain
+34-91-348{2241, 2284}
{j.macias, pablo.castells}@uam.es

INTRODUCTION

One of the main problems in PBE inferencing is that of deriving abstract data characterizations from concrete objects and values involved in user's manipulations [8]. This involves selecting variables and constants (parameterization), and establishing what variables represent. Some variables will be input parameters of the construct or procedure inferred by the PBE system, and some will be expressions that indicate values or objects that are obtained from input parameters by traversing a data structure made of relations, lists and attributes. Our contribution to this workshop sets the focus on the use of rich descriptions of application data to achieve correct and expressive data characterizations.

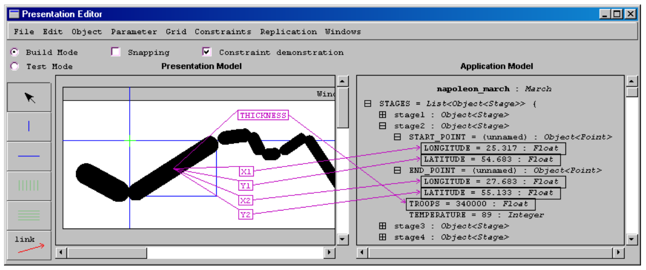
Data models, or the wider and more conceptual notion of domain models [21], provide an understanding of the knowledge behind the visual representations that the user manipulates in a PBE system. This knowledge can be extremely useful to make sense of the user's actions on visual objects. It can help, for instance, select variables, find relations between visual objects, focus attention on special objects, mark out composite display units, build complex data flow expressions, and disambiguate the intended meaning of user's actions. How to best exploit this knowledge; how to keep track of the relation from visual objects back to the domain knowledge items they correspond to; how much of the domain model should the user be exposed to an in what form; what kind, if any, of underlying representation should be used for the user interface, are a few of the difficult problems and issues that we will put forward and discuss here.

VISIBLE APPLICATION DATA EXAMPLES

Our previous experience in this context includes research on an interface development environment, HandsOn [3, 4], where the interface designer can manipulate explicit examples of application data at design-time to build custom dynamic displays that depend on application data at run-time. [3] shows how HandsOn can be used to generate the well-known Minard chart showing Napoleon's march to Moscow (partially shown in Figure 1), consisting of a sequence of line segments whose thickness, color, and endpoints represent the number of troops in Napoleon's army, the temperature, and the geographical coordinates respectively.

Rather than building a generic display, the designer constructs specific displays using specific data and HandsOn generates abstract constructs by generalizing the examples. The data examples disappear when the PBE system infers generic displays, where concrete values are replaced by variables and expressions whose values are computed at runtime.

In HandsOn the user is exposed to the application data model through an explicit view of data examples next to the interface design area (see Figure 1). The design tool allows connecting data to visual components by pointing at and dragging data and display elements. Data examples provide the designer with concrete objects to refer to, and they provide the system with information that the system uses to infer the designer's intent.



```
napoleon_march : March
⊟ STAGES = List<Object<Stage>> {
    ⊞ stage1 : Object<Stage>
    ⊟ stage2 : Object<Stage>
        ⊟ START_POINT = (unnamed) : Object<Point>
            LONGITUDE = 25.317 : Float
            LATITUDE = 54.683 : Float
        ⊟ END_POINT = (unnamed) : Object<Point>
            LONGITUDE = 27.683 : Float
            LATITUDE = 55.133 : Float
        TROOPS = 340000 : Float
        TEMPERATURE = 89 : Integer
    ⊞ stage3 : Object<Stage>
    ⊞ stage4 : Object<Stage>
```

Figure 1. Linking interface objects to data examples in HandsOn

HandsOn analyzes the types and structural properties (e.g. iteration and recursion) of the data to automatically generate presentation constructs. In doing so, the system also examines visual properties and geometric relationships among the objects being manipulated, existing mappings from data to presentations, and the way data is being seen by the designer (e.g. expanded nodes, selected values, focused structures). For instance, if a visual object displays a value that belongs to a list, HandsOn suggests to create a list of visual objects to display the remaining list values. Similarly, recursive display structures can be generated for recursive data structures.

Overall, the types of decisions that HandsOn is able to make include:

- Mapping data structures to custom display structures.
- Generating and adjusting transformation functions (e.g. scaling, type conversion) between display parameters (e.g. endpoints of a line) and application values (e.g. geographic coordinates).
- Propagating changes over replicated objects across display structures.
- Replacing example values by variables and expressions when the design is finished.

HandsOn uses a highly structured and sophisticated internal model of interface displays, based on the presentation model of an existing model-based GUI development tool, Mastermind [2], which supports dynamic presentation functionalities. HandsOn was entirely implemented in Amulet [18], a high-level user interface toolkit that provides a) a constraint system that we used to link display components to application data, and b) an easily inspectable object-based representation of user interfaces that facilitates the exploration of the interface structure.

ONTOLOGY-BASED DOMAIN MODELS FOR DYNAMIC WEB PAGE AUTHORING

In some cases it may be useful to extend the notion of data model to the more conceptual notion of domain model [19], because 1) it provides an even richer description and a deeper understanding of the application knowledge behind what the user sees, and 2) the usage and availability of explicit domain models is becoming increasingly common in knowledge-based applications on the WWW. Under this view, the notion of knowledge base takes the place of application data.

The generalization of the WWW as a universal computing platform, and the unprecedented size of application user communities it bears, has motivated us to take it as an interesting ground for end-user programming research. Starting from the observation that most of today's WWW is made of dynamically generated web pages, and the fact that development of dynamic pages is considerably difficult and requires advanced programming skills, we have taken up the challenge of devising an interactive authoring tool where dynamic web pages can be edited in a WYSIWYG environment similar to a standard HTML editor [15, 22].

In order to tackle such a difficult problem, we have considered the assumption that a domain model and a presentation model are available. We assume an ontology-based domain description [9], and a specification of presentation on a per-class basis. We believe these assumptions are congruent with important trends in current web technology. The emerging semantic web view [1] promotes the construction and widespread availability of explicit models of domain knowledge. Simultaneously, a major recurring motto in the development of the new web technologies is the separation of data (domain knowledge) and presentation (e.g. XML+XSLT, see also XMLC [23], Cuypers [19], PEGASUS [5], to name a few).

The DESK Authoring Tool

We have developed a tool, DESK [13, 14], where authorized users can customize web page generation procedures by editing specific HTML pages produced by a dynamic page generation system. DESK acts as a client-side complement of a dynamic web page generation system, PEGASUS [5, 12], which generates HTML pages from an ontology-based domain model and an abstract presentation model. The PEGASUS presentation model specifies which pieces of knowledge should be presented and how when a certain unit of information from the domain model is output to the user. Instead of using the PEGASUS modeling language, authorized users can modify the internal presentation model by editing in DESK the HTML pages generated by PEGASUS.

DESK uses the PEGASUS domain model to a) identify pieces of domain contents in the edited page, b) establish relations between them, c) select one (or more) of the involved knowledge items as the root domain object behind the web page, from which all other objects are referred to as relative to this one, d) detect iteration patterns when the user lays out data over structured displays (e.g. records in a table).

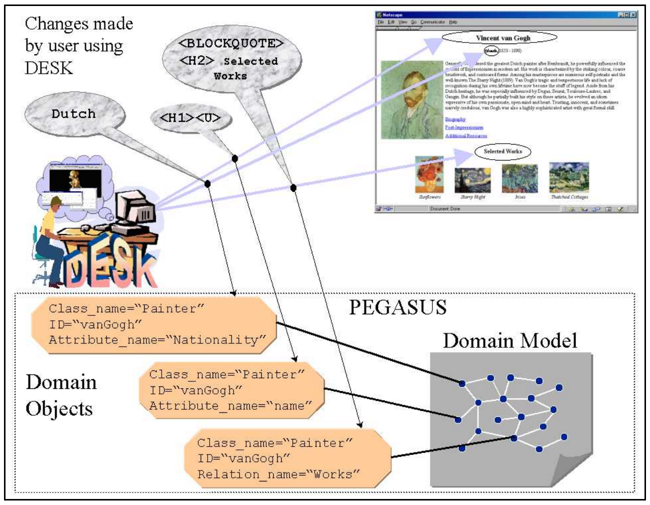

Changes made by user using DESK
<BLOCKQUOTE> <H2> Selected Works
Dutch
<H1><U>
Vincent van Gogh
Selected Works
DESK
PEGASUS
Domain Model
Domain Objects
Class_name="Painter" ID="vanGogh" Attribute_name="Nationality"
Class_name="Painter" ID="vanGogh" Attribute_name="name"
Class_name="Painter" ID="vanGogh" Relation_name="Works"

Figure 2. Detecting correspondences between page blocks and domain objects with DESK

For instance, consider a web page like the one shown on the upper-right corner of Figure 2, where information about Vincent van Gogh is presented. Assuming an ontology has been defined in PEGASUS for a virtual museum or a course on history of art, including classes like Painter, Painting, School, and so on, DESK is able to find that this page is displaying attributes (name, birth, short biography) and relations (works, school) of an instance of Painter. If the user adds text, changes the style or the position of a piece of the document (e.g. the thumbnail image on the lower-right corner of the web page in Figure 2), DESK finds a description of this piece that relates it to the main object (van Gogh) in terms of the vocabulary defined by the application domain ontology (e.g. "the *small-image* attribute of the last element in the *selected-works* relation of the object with ID *vangogh*”). This information is used by DESK to modify the presentation model for class Painter (or class Painting if appropriate), so that the change is permanent for all objects of class Painter. The van Gogh instance acts as an example for the user to see and change how a painter presentation (by PEGASUS) looks like, and DESK generalizes the modification to the whole instance class.

Reverse Engineering

PEGASUS generates web pages on the fly from a semantic network of ontology instances (the application data/knowledge) as the user implicitly requests viewing domain objects. These requests are internally generated from the navigational interaction of the user with an application supported by PEGASUS. To present an object, PEGASUS finds its class and applies the presentation model associated to the class to generate a web page where selected pieces of the object are displayed. DESK follows the inverse path: it parses the web page and locates the source of page fragments in the domain model, as well as the part of the presentation model that defines how the fragment was presented.

This backward transition from syntactic blocks to semantic blocks can be seen as a reverse engineering problem, and as such is a non-trivial task. Our current approach is based on a simple search of text and multimedia fragments in the domain knowledge base. Devising a smarter search is an open issue in our work. Other main difficulties are cutting out the right syntactic blocks in the page to be found in the KB, and removing the ambiguity when the search yields multiple results. To solve the latter, DESK uses heuristics such as requiring that found contents are connected to each other in the domain model, and priorizing the closeness of knowledge units in the semantic network. We carry out the former by looking for hints in user actions (e.g. selection of blocks), domain contents (e.g. readjust block boundaries when the search yields a partial match), and the syntactic structure of the display (e.g. paragraphs, table cells, etc.).

DESK uses an implicit display model based on different kinds of pre-programmed presentation widgets such as tables, selection lists, combo boxes, trees and so forth, as supported by HTML. Because the most flexible construct for structured layout in HTML are tables, an important part of our work in DESK is concerned with specific strategies for treating complex mappings from application data structures to nested tables. The considerable number of research works related to table analysis and interpretation that can be found in the literature [6, 7, 10] proves that table parsing is a difficult problem and an interesting object of study by itself. We believe that the introduction of models of domain knowledge in this frame brings about interesting views on the problem, that are particularly pertinent in the context of web applications and HTML as a standard for web user interface presentation. In particular, while other systems infer data structures from tables in HTML documents, in DESK the data structure description is taken from a dynamically built structured model of user’s actions related to domain information.

DISCUSSION

Building dynamic information visualization interfaces from examples requires elaborate data characterizations when the underlying domain knowledge has a complex structure, as is the case in many knowledge-based web applications and information systems. The usage of ontologies, i.e. explicit descriptions, to organize and share knowledge in such systems is becoming an increasingly popular approach. We propose to exploit these explicit models of domain knowledge, which are available for free (from the PBE system developer point of view), to improve the reach and precision of PBE techniques, and in particular as a highly valuable source of information for data characterization.

The use of a data model was already present in one of the earliest PBE systems, Peridot [16], in a very simple form. Peridot lets the user create a list of sample data to construct lists of user interface widgets. In Gold [17] and Sagebrush [20] the user can build custom charts and graphics by relating visual elements and properties to sets of data records. The data model in Peridot consists of lists of primitive data types. Gold and Sagebrush assume a relational data model. Our view in this regard is that it is interesting to lift these restrictions and support richer information structures, as proposed in our current and earlier research work.

One interesting issue when domain or data models are used in a PBE system is whether and to what extent the model should be visible for the user. There is a whole range between completely hiding the domain model and showing a full literal (abstract) view of it. Moving along this axis means trading simplicity for expressive power. For instance, HandsOn does show data, but in the form of specific examples, easier to have in mind and manipulate than an abstract model. The explicit manipulation of data in HandsOn has an additional advantage: keeping track of the relation from visual objects to data is not a problem, as all these links are defined by the user in the system, so that HandsOn can store and remember them. In DESK the data-presentation relation is known by the page generation system, PEGASUS, but this information is lost when DESK gets the generated page, and has to be recovered by the PBE system in a difficult and costly reverse engineering process.

DESK uses a more expressive model of application knowledge that HandsOn, but completely hides it from the user to stay within the strict WYSIWYG principle, thus requiring zero awareness from the user of the internal knowledge representation. In exchange, DESK has important expressive limitations: it is not possible to modify the way visual components are linked to domain objects, and it is not possible to add new object part presentations that are not already present in a page. This means, in particular, that it is not possible to build a presentation from scratch, and the user can only customize existing designs. We have followed this approach as an experiment to investigate how far one can go without giving up on the WYSIWYG requisite, but there is nothing that impedes augmenting DESK with views of the domain model to provide the expressive capabilities needed to remove these limitations, except a compromise in simplicity of use.

Other sources of information for PBE inferencing used by DESK, besides models of application domain knowledge, include spatial properties and relations in the edited display, knowledge about page design practice (page layout, table layout, standard spatial patterns), know-how about interactive design manipulation, and structure in user's actions and behavior (e.g. order and sequencing, iterative patterns). For the underlying representation of the visual constructs being created or modified by example, DESK does not use such a sophisticated model as HandsOn does because a) the editing functionalities and gestures needed to manipulate data examples and interface components in the HandsOn environment require a detailed description of the involved objects and imply frequent readjustments in data flow relations, whereas DESK provides more limited capabilities based on HTML editing, and b) the run-time implementation platform for target constructs in HandsOn is a full-fledged window-based toolkit (Amulet), while in DESK the (PEGASUS) interface model essentially builds upon the much simpler HTML user interface model.

ACKNOWLEDGMENTS

The work reported in this paper is being supported by the Spanish Ministry of Science and Technology (MCyT), project number TIC2002-1948.

REFERENCES

1. Berners-Lee, T., J. Hendler, O Lassila. The Semantic Web. Scientific American, May 2001.
2. Castells, P., Szekely, P., Salcher, E. Declarative Models of Presentation. Proceedings of the International Conference on Intelligent User Interfaces (IUI'97). January 6-9, 1997, Orlando, Florida, USA, pp. 137-144.
3. Castells, P., Szekely, P. Presentation Models by Example. In: Duke, D.J., Puerta A. (eds.): Design, Specification and Verification of Interactive Systems '99. Springer-Verlag, 1999, pp. 100-116.
4. Castells, P., Szekely, P. HandsOn: Dynamic Interface Presentation by Example. Proceedings of the HCI International'99. Munich, 1999, pp. 188-1292.
5. Castells, P., Macías, J.A. An Adaptive Hypermedia Presentation Modeling System for Custom Knowledge Representations. Proceedings of WebNet'01 - World Conference on the WWW and Internet. Orlando, Florida; October 23-27, 2001. Published by AACE, pp. 148-153.
6. Chen, H.; Tsai, S.; Tsai, J. Mining tables from large sale HTML texts. 18th International Conf. on Computational Linguistics (COLING), 2000, pp. 166-172.
7. Cohen, W., Hurst, M., Jensen, L. A flexible Learning System for Wrapping Tables and Lists in HTML Documents. In Proceedings of the WWW Conference. Honolulu, Hawaii, USA. May 7-11, 2002, pp. 232-241.
8. Cypher A. (ed.). Watch What I Do: Programming by Demonstration. The MIT Press, 1993.
9. Gruber, T. R. A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition, 5(2), pp. 199-220, 1993.
10. Hurst, M. F. The Interpretation of Tables in Texts. PhD. Thesis. University of Edinburgh, 2000.
11. Little, S., J. Geurts, and J. Hunter. Dynamic Generation of Intelligent Multimedia Presentations through Semantic Inferencing. 6th European Conference on Research and Advanced Technology for Digital Libraries (pages 158-189). Springer, 2002.
12. Macías, J.A., Castells, P. A Generic Presentation Modeling System for Adaptive Web-based Instructional Applications. ACM Conf. on Human Factors in Computing Systems (CHI'2001). Seattle, April 2001.
13. Macías, J.A., Castells, P. Dynamic Web Page Author-ing by Example Using Ontology-Based Domain Know-ledge. International Conference on Intelligent User Interfaces (IUI'2003). Miami, Florida. January 2003, pp. 133-140.
14. Macías, J.A., Castells, P. DESK-H: building meaningful histories in an editor of dynamic web pages. To appear in 11th International Conference on Human-Computer Interaction (HCII'2003). Creta, Grece, June 23-27, 2003.
15. Miller, R., Myers B. Creating Dynamic World Wide Web Pages By Demonstration. Carnegie Mellon University School of Computer Science, CMU-CS-97-131 and CMU-HCII-97-101, 1997.
16. Myers, B. A. Creating User Interfaces by Demonstration. Academic Press, San Diego, 1988.
17. Myers, B. A., J. Goldstein, M. Goldberg. Creating Charts by Demonstration. Proceedings of the CHI'94 Conference. ACM Press, Boston, April 1994.
18. Myers, B. A., et al. The Amulet 2.0 Reference Manual. Carnegie Mellon University Tech. Report, 1996.
19. Ossenbruggen et al. Towards 2nd and 3rd Gener-ation Web-Based Multimedia. 10th International World Wide Web Conference. ACM Press, 2001, pp. 479-488.
20. Roth, S. F. et al. Interactive Graphic Design Using Automatic Present-ation Knowledge. CHI'94 Con-ference. Boston, April 1994, pp. 112-117.
21. Spyns, P, R. Meersman, M. Jarrar. Data modelling versus Ontology engineering. SIGMOD Record 4, Dec. 2002.
22. Wolber, D., Su, Y., Chiang Yih. Designing Dynamic Web Pages and Persistence in the WYSIWYG Interface. Proceedings of the International Conference on Intelligent User Interfaces (IUI'2002). San Francisco, California, USA. January 13-16 2002, pp. 228-229.
23. Young D. H. Enhydra XMLC Java Presentation Development. Sams, 2002.

Short biography

José Antonio Macías Iglesias is a teaching assistant of the Escuela Politécnica Superior of the Universidad Autónoma de Madrid, where he is finishing his PhD thesis under the supervision of P. Castells. His PhD thesis is about the use of WYSIWYG authoring tools that use PBE techniques to create user interfaces for complex knowledge-based systems, with support for dynamic interface features (e.g. dynamic web pages). His general research interests fall in the areas of Human-Computer interaction, Computer-Assisted Learning, Ontology Engineering and the Semantic Web.

Pablo Castells is a staff member of the Escuela Politécnica Superior of the Universidad Autónoma de Madrid since October 1995, where he currently holds an associate professor position. He obtained a M.S. degree in Mathematics in 1989 and a PhD in Computer Science in 1994 at the Universidad Autónoma of Madrid. His PhD thesis was on automated theorem proving and knowledge-based problem solving in Physics and Mathematics. His current research interests include User Interface Development Tools, Progamming by Example, User Modeling, Ontologies, and Semantic Web technologies.

From April 1994 to September 1995 P. Castells worked under a post-doc fellowship at the Information Sciences Institute (ISI) of the University of Southern California (USC), where he collaborated with the Mastermind project, in which he contributed to the development of a model-based presentation support module for dynamic interface components that automatically adjust to application data, user models and platform characteristics. Back in Madrid, he worked on a PBE authoring tool for the Mastermind GUI development system. Since 1999 he has been working on an ontology-based presentation system for adaptive hypermedia. This research has progressed towards the wider scope addressing knowledge visualization and navigation on the Semantic Web. Simultaneously, J. A. Macías and P. Castells are investigating the development of interactive authoring techniques for adaptive hypermedia systems and ontology-based knowledge visualization systems. They are currently conducting this work in the context of wider projects in the area of the Semantic Web technologies.

End User Programming for Web Users

Robert C. Miller
MIT Lab for Computer Science
200 Technology Square, Cambridge, MA 02139 USA
rcm@lcs.mit.edu
http://graphics.lcs.mit.edu/~rcm

Introduction

The World Wide Web is increasingly a focus of business and entertainment. Applications which formerly would have been designed for the desktop — calendars, travel reservation systems, purchasing systems, library card catalogs, map viewers, even games like crossword puzzles and Tetris — have made the transition to the Web, largely successfully.

Applications have moved to the Web for a number of reasons. First, and probably most important, web applications need no installation. Just click on a link, and you can use the application immediately. Bugs can be fixed and new features can be rolled out without requiring users to install upgrades or apply patches. Multiple platforms are also easier to support. Platform-independent standards, such as XHTML, DOM, ECMAScript (aka JavaScript), and CSS, make it possible to target a web application to any standards-compliant web browser, regardless of operating system or windowing environment.

The migration of applications to the Web opens up a new vista of opportunity for end user programming. Applications that would have been closed and uncustomizable on the desktop suddenly sprout numerous hooks for customization when implemented in a web browser. Structured displays are represented by machine-readable HTML. Commands are invoked by generic HTTP requests. Graphical layouts can be tailored by stylesheet rules.

Unfortunately, although web browsers have a long history of built-in scripting languages, these languages are not designed for the *end user* of a web application. Instead, languages like JavaScript and Curl [8] are aimed at *designers* of web applications. Granted, many web designers lack a traditional programming background, so they may be considered end users in that respect. But the needs of a designer, building an application from whole cloth, differ greatly from the needs of a user looking to tailor or script an existing application. Current web scripting languages do not serve those needs.

In this paper, I consider the problem of end-user automation of web applications. After covering background and related work, I will present several motivating examples, and distill from those examples some essential requirements on a programming system for web users. Preliminary steps toward such a system have been taken, and the resulting research prototype (LAPIS) will be briefly described. Finally, I will mention some of the hard problems that arise.

Background

Closed, uncustomizable applications have long been a bugaboo for end-user scripting on the desktop. Despite the long existence of scripting frameworks like AppleScript, OLE Automation, and Visual Basic for Applications, and exhortations by platform vendors to support them, many desktop applications still do not provide the hooks required for scripting. In a software development environment that demands tight development cycles and short times to market, scripting and customization get short shrift compared to more pressing concerns like feature set, performance, reliability, and usability.

When a desktop application fails to provide an application programming interface (API), the end user must resort to automating the user interface — a technique often called, somewhat derogatively, *screen scraping*. Cross-application macro recorders support this technique by recording the user's mouse movements and keystrokes, then playing them back by inserting simulated mouse and keyboard events in the system queue. Macro recorders have a serious flaw in that they can only simulate input; they have no way to read an application's display to extract information or condition their behavior on the application's state. Triggers [14] and VisMap [15] address this problem by interpreting the screen contents at a pixel level, but this approach is challenging to program and has so far been applied only to simple tasks.

Interpreting desktop application output is hard. Web applications, however, display their output in structured, machine-readable HTML, making screen scraping much easier. As a result, web screen scraping abounds. Comparison-shopping sites, such as Priceline.com, use screen scraping behind the scenes to extract price information from online retailers. A Boston Red Sox fan used screen scraping to try to stuff the ballot box for baseball's 1999 All-Star Game ballot [2].

The component of a web screen scraper that interprets a web page and extracts information from it is called a *wrapper*. Many wrappers are written by hand in scripting languages like Perl or Python, but work has also been done on inducing wrappers from examples [7].

Most web screen scrapers are written in a scripting language that dwells *outside* the web browser, like Perl, Python, or WebL [4]. For an end-user, the distinction is significant. Cookies, authentication, session identifiers, plugins, user agents, client-side scripting, and proxies can all conspire to make the Web look different outside the web browser than inside. But perhaps the most telling difference, and the most intimidating one for an end user, is the simple fact that outside a web browser, a web page is just raw HTML. Even the most familiar web portal looks frighteningly complicated when viewed as HTML source.

Unfortunately, only a handful of systems have looked at putting end-user web automation into the browser, where it belongs. LiveAgent [5] is a macro recorder that can record and play back a sequence of browsing actions, using a local HTTP proxy to snoop on the user's actions. SPHINX [10] is a user-configurable web crawler that runs as a Java applet in the user's web browser, so that it would see the same pages seen by the user. TrIAS [1] constructs wrappers from examples given in a web browser.

Although JavaScript is primarily intended for site designers, end users can access it with *bookmarklets* [3]. A bookmarklet is a short piece of JavaScript code encoded as a URL and stored in a bookmark. When the user clicks on the bookmark, the JavaScript code runs on the current page. For example, here is a simple bookmarklet that changes the current page's background to white:

```
javascript:void(document.bgColor='white')
```

Bookmarklets can extract data from pages, change display properties, adjust window properties, and visit other sites. However, a bookmarklet must fit into a URL, strongly constraining its length and making it hard to read and modify.

Mozilla has brought some promising developments in browser-centric web automation [13]. All the "chrome" in Mozilla — the toolbars, panels, and dialog boxes that surround the browser itself — are specified in XUL, an XML-based user interface description language. A combination of XUL, JavaScript, and CSS is used to implement web screen scrapers directly in the browser. For example, when a Google search results page is displayed in the browser, Mozilla automatically parses it to present the results as a list of hyperlinks in the sidebar. Mozilla promises to be a powerful testbed for future research into end-user web automation.

Scenarios

For further motivation, let us consider some scenarios in which end-users of web applications would want scripting and customization. These scenarios offer concrete examples that guide the requirements to be discussed in the next section.

Scenario 1: Reviewing. Many conferences — including CHI — now use a web application to receive papers, distribute them to peer reviewers, and collect the reviews. A reviewer assigned 10 papers to read and review faces a lot of repetitive web browsing to download each paper, print it, and (later) upload a review for it. Some reviewing applications require the review to be submitted in a web form, so a review prepared off-line must be copied and pasted into the appropriate fields of the form. Tedious repetition is a strong argument for automation. Unfortunately, since the reviewing application is protected by authentication, a simple Perl script won't do the job.

Scenario 2: House hunting. Prospective home buyers in the US can use the Multiple Listings Service (MLS) to search for homes matching various criteria. A number of real estate companies now offer web interfaces that search the MLS (e.g., www.realtor.com). Interestingly, different MLS search interfaces provide different subsets of the available information, forcing a home buyer to search several sites to get a more complete picture. Furthermore, many location preferences that may be personally important to a home buyer cannot be specified in the search. If I buy this house, how far will I have to commute to my work? How far is the nearest grocery store, subway stop, or public park? How far is it from my mother's house? These questions can be answered by plugging the house address into an online map site (e.g., MapQuest).

Scenario 3: Book shopping. A voracious reader may frequently visit an online bookstore (e.g., Amazon.com) with a list of books to buy. A voracious reader on a budget, however, may want to check first whether any of the books are available in a local public or university library by searching its online catalog. This is a feature that Amazon is unlikely ever to offer.

Requirements

The scenarios above suggest a number of desirable criteria for an end-user web automation system.

Browser centricity. In these scenarios, the web browser is the center of the user's activity. Tasks interleave manual operations, such as logging in to a site, with automatable operations, such as downloading papers or searching for books. If the automation takes the user out of the browser, or digs below the familiar rendered world of the Web into raw HTML, the user's work flow is interrupted.

Data-parallel operations. Much of the repetitive activity in these scenarios revolves around sets of data items: papers to print, reviews to upload, houses to search, addresses to map, books to look up. The ability to apply an operation to multiple items at once would be extremely valuable in web browsing, just as it is in file managers, word processors, and drawing editors.

Cross-site scripting. The scenarios often require interacting with multiple web applications in the same task: e.g., multiple real estate sites, or a real estate site and a mapping site, or a bookstore and a library catalog. Instead of being confined to the environment of a single page, as JavaScript typically is, end-user automation must smoothly interact with multiple pages, extracting data from one page and using it in another.

Both manual and automatic invocation. Suppose the user creates a *distance-to-work* script that takes a house address as input and uses an online mapping site to compute how far the user would have to commute to work from that address. This script might be invoked in several ways. With manual invocation, the user selects a house (or list of houses) and triggers the script from a menu or toolbar. Bookmarklets support only manual invocation. With automatic invocation, on the other hand, the browser automatically runs the script on any page recognized as a list of houses. The resulting distances might be inserted in the house's description, or they might be used to filter the list of houses, hiding any that are farther than a given threshold. Mozilla's search sidebar uses automatic invocation; whenever it detects a Google search results page, it automatically parses the page and displays the results in the sidebar. Automatic invocation allows custom behavior to be injected into a web application in ways that were impossible with desktop applications.

Approach

The LAPIS research project at MIT is working toward this vision of end-user automation in the web browser. Our current prototype, LAPIS, is written entirely in Java. The LAPIS browser can display simple HTML, visit hyperlinks, and submit web forms, but it fails to support all the standards (such as cookies, JavaScript, CSS, and SSL) required by modern web applications. Work is underway to port some of the novel features of LAPIS into Mozilla, giving a much richer, standards-compliant testbed for web automation.

LAPIS is described in detail elsewhere [9]. Features that are most relevant to end user automation are highlighted below:

Pattern library. LAPIS includes an extensible library of patterns and parsers that can be referred to by simple names, such as Link, Paragraph, Button, or Table. An HTML parser is included in the library, naturally, but so are patterns for other common kinds of text structure, including dates, times, phone numbers, email addresses, URLs, etc. Wrappers for web sites, such as Google or Amazon, would naturally fall into the pattern library. A pattern library raises the abstraction level of data descriptions, so that when users think about identifying elements and extracting data from a web page, they can think in terms of *books* or *addresses* rather than low-level features of HTML. The LAPIS library is designed to be extended, and is language-independent in the sense that a library pattern can be implemented by an arbitrary kind of scanner — regular expression, context-free grammar, parser generator, neural network, or even a Turing-complete program.

Pattern language. Library patterns can be glued together with a pattern language called *text constraints*, which uses relational operators such as *before*, *after*, *in*, and *contains* to describe a set of regions in a page. The matches to a pattern are displayed as multiple selections, and editing commands can affect all selections at once [12]. LAPIS was designed with data-parallel operations in mind.

Command language. LAPIS has an embedded scripting language aimed at the end user, not the page designer. (Tcl was chosen as the scripting language, partly because of its syntactic simplicity and partly because a good pure-Java implementation was available. Tcl is also well-suited for interactive command execution.) Commands take patterns as arguments to indicate how to manipulate a web page. For example, the *keep* command extracts a set of regions matching a pattern; *delete* deletes the regions; *sort* sorts the regions in-place; and *replace* replaces each region with some replacement text, which may be a function of the original region. Other commands interact with the web page as a user would: *click* simulates a click on a hyperlink or form control matching a pattern, and *enter* places text in a form field. JavaScript can also access form controls, of course, but an important difference is that LAPIS patterns can be written without looking at the underlying HTML source, e.g.:

```
click {Link containing “Download this paper”}
click {Checkbox just after “Garage”}
```

Writing equivalent commands in JavaScript requires digging into the HTML source to find the names of the fields.

Browser shell. Instead of presenting the Tcl interpreter in a separate window, LAPIS integrates the interpreter directly into the browser window. Tcl commands may be typed into the Location box. The typed command is applied to the current page, and the command's output is displayed in the browser as a new page that is added to the browsing history. A command may also invoke an external program, passing the current page as standard input and displaying the program's standard output and error streams as a new page. This "browser shell" interface [11] allows legacy programs and scripts written in other languages to be integrated seamlessly into the browser environment.

Challenges

The primary challenge for end-user automation in the web browser can be simply stated: the user should never have to view the HTML source of a web site to customize or automate it. Web sites are becoming increasingly complicated. Even when a web interaction could be scripted outside the browser (with no trouble from cookies, authentication, or dynamically-generated content), the need to examine and understand the HTML source is a roadblock that discourages spur-of-the-moment innovation. Web automation must be done at the level of rendered pages.

This problem is far from trivial. What the user sees as "blue text" in a rendered page may be blue for many reasons: because it is a hyperlink; because it is contained in a FONT tag; because it has a CSS style attribute; because it matched by a CSS stylesheet rule; or because its color attribute was set by some JavaScript code. Worst of all, the "blue text" may be only a picture of text, embedded in a GIF or JPG image! The text pattern matching approaches used for web screen scraping outside the browser no longer work in general.

An automation system must deal smoothly with the proliferation of Web standards and syntaxes — XHTML, XML, CSS, MathML, SVG — while hiding the distinctions between them from the user. It must be integrated with a fully standards-compliant web browser, so that the user's web applications are functional and usable. Where previous approaches used a web proxy to extend the browser (e.g., LiveAgent), embedding automation into the browser is more likely to achieve the desired results.

Another challenge facing end-user web automation, like all web screen scrapers, is dealing with changes in web applications. One of the benefits of web applications (for their designers) is that changes can be rolled out without notice to users, but this turns out to be detrimental to end-user automation. Some steps toward solving this problem include regression tests that can detect when a wrapper is going wrong [6] and intelligent agents that relearn failed wrappers with the user's help [1]. Web services with well-specified XML APIs will also help, although considering how few desktop applications have scriptable APIs, it is hard to be optimistic about web applications.

Acknowledgements

This research was sponsored in part by the National Science Foundation, the Army Research Office, the Defense Advanced Research Project Agency, and the USENIX Association. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any sponsoring party or the U.S. Government.

References

[1] Mathias Bauer, Dietmar Dengler, Gabriele Paul, and Markus Meyer. Programming by demonstration for information agents. *Communications of the ACM*, 43(3):98–103, March 2000.

[2] Gordon Edes. This hack tried but couldn't connect. *Boston Globe*, July 1999.

[3] Steve Kangas. Bookmarklets. http://www.bookmarklets.com/, 1998.

[4] Thomas Kistler and Hannes Marais. WebL – a programming language for the Web. In *Proceedings of the 7th International World Wide Web Conference (WWW7)*, 1998.

[5] Bruce Krulwich. Automating the internet: Agents as user surrogates. *IEEE Internet Computing*, 1(4):34–38, 1997.

[6] Nicholas Kushmerick. Wrapper verification. *World Wide Web*, 3(2):79–94, 2000.

[7] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. Wrapper induction for information extraction. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[8] Friedger Müffke. The Curl programming environment. *Dr. Dobb's Journal*, September 2001.

[9] Robert C. Miller. *Lightweight Structure in Text*. PhD thesis, Carnegie Mellon University, May 2002.

[10] Robert C. Miller and Krishna Bharat. SPHINX: a framework for creating personal, site-specific web crawlers. *Computer Networks and ISDN Systems*, 30(1–7):119–130, 1998.

[11] Robert C. Miller and Brad A. Myers. Integrating a command shell into a web browser. In *USENIX 2000 Annual Technical Conference*, pages 171–182, June 2000.

[12] Robert C. Miller and Brad A. Myers. Multiple selections in smart text editing. In *Proceedings of the Sixth International Conference on Intelligent User Interfaces (IUI 2002)*, pages 103–110, 2002.

[13] Ian Oeschger, Eric Murphy, Brian King, Pete Collins, and David Boswell. *Creating Applications with Mozilla*. O'Reilly, 2002.

[14] Richard Potter. Triggers: Guiding automation with pixels to achieve data access. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 360–380. MIT Press, 1993.

[15] Luke Zettlemoyer and Robert St. Amant. A visual medium for programmatic control of interactive applications. In *Proceedings of ACM Conference on Human Factors in Computer Systems (CHI '99)*, pages 199–206, 1999.

Studying Development and Debugging To Help Create a Better Programming Environment:

For the: CHI 2003 Workshop on Perspectives in End User Development

Brad Myers
Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213 USA
+1 412 268 5150
bam+@cs.cmu.edu
http://www.cs.cmu.edu/~bam

Andrew Ko
Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
+1 412 412 0042
ajko@cmu.edu
http://www.andrew.cmu.edu/~ajko

INTRODUCTION

The event-based style is increasingly common in modern end-user programming languages. Visual Basic, Macromedia's Director, web scripting languages, as well as many recent novice-geared research prototypes such as Alice [9] and HANDS [8], provide event-based constructs and user interfaces, enabling programmers to create highly interactive environments. Yet very few of the user studies of programming environments and language usability investigate the event-based style. Since recent studies suggest that language paradigm is a predictor of program comprehension and programming strategies [2] [6], event-based programming environments and their user interfaces should be tailored to the event-based style.

As part of the Natural Programming Project (http://www.cs.cmu.edu/~NatProg/) we have begun to investigate how end users create, test, and debug event-based programs in an effort to guide the design of novel programming environment tools for end-user event-based languages. While our research has thus-far focused on Alice (see Figure 1), a system for novice programmers, we believe that the techniques we are proposing will be useful to novice and expert programmers using more general languages with event support such as Visual Basic, Java and C#.

STUDIES

We recently conducted a study of expert Alice users in order to get a glimpse at the general areas of difficulty

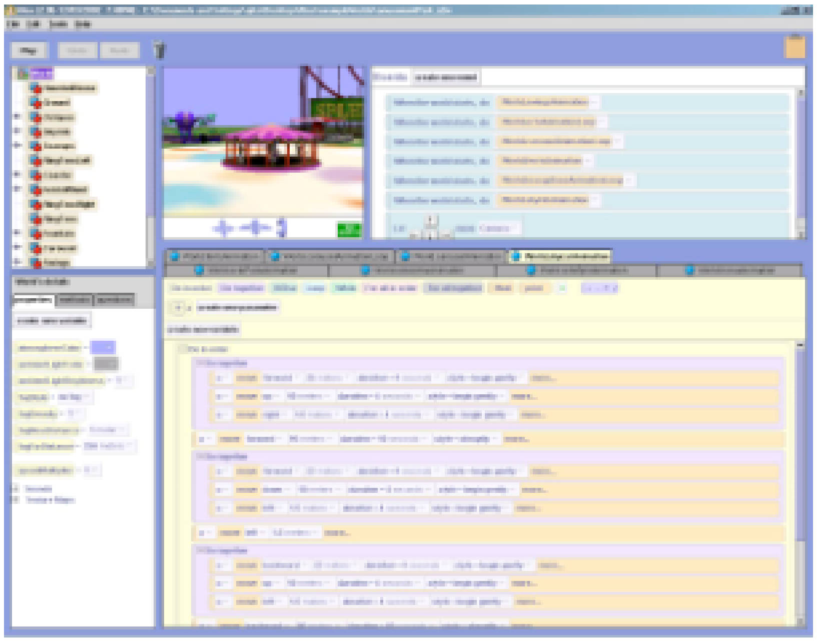


Figure 1. This is a typical view of Alice, with code and events at the lower and upper right, the worldview at the top center, the object list at the top left and the selected object's properties at the bottom left.

with using event-based languages and environments. The study used the method of contextual inquiry (CI) [1] in order (1) to identify problems that programmers encounter when creating highly interactive, event-based programs, and (2) to evaluate the utility of using CIs to extract design requirements for programming environments.

Participating programmers were enrolled in the "Building Virtual Worlds" course offered at Carnegie Mellon University. The course required collaborations among programmers, modelers, sound engineers, and painters to create a new interactive 3D world every two weeks using Alice. Alice provides a limited object model, global event handlers, and a structured editor that prevents all syntax errors.

Four programmers were recruited and observed during the second half of the semester, after the programmers were experienced with Alice. All had extensive experience with more than one programming language. As programmers worked on their programs, the experimenter recorded observations on paper and video. The experimenter formed hypotheses about the programmer's actions *in situ* and asked the programmers if the hypotheses were correct. For example, the experimenter would say, "It looks like you're trying to align these two objects." and the programmer would reply, "Basically. I want them to be aligned on this axis, but I don't care about the other two." Participants were paid $10 per hour for their participation.

Approximately 12 hours of observations were obtained from the four programmers over 12 sessions. Each of the sessions was reviewed for *breakdown scenarios*, in which a programmer's strategy was difficult to perform or unsuccessful. Breakdowns scenarios were consolidated into the problem types described below.

Programming Problems

In many breakdowns, code was reused to perform similar operations, such as animations or calculations, but the code was not properly adapted for its new location. These bugs were particularly difficult to isolate because they propagated through complex animations, which depended on events. These breakdowns highlight the need for more support for code reuse. We are designing a *smart copy and paste* mechanism that could automatically coerce parameters from method to method.

In other scenarios, programmers needed to create finely tuned sequences of animations and events by tweaking existing code. However, programmers often reverted to a previous version of code manually, to avoid undoing intermediary changes to unrelated code. One way to alleviate this difficulty would be to keep an extensive code history, which could support a *multi-level intelligent undo* and *checkpointing*.

Testing Problems

The programmers used visual cues extensively in order to aid testing tasks. For example, one programmer assigned the color of an object to the triggering of an event handler in order to verify the event occurred at the proper time. This suggests a need for a way of saying "*watch this variable by mapping it's value to **this***" where *this* could be a visual cue such as an object's color, size, or visibility.

A difficulty in testing code in isolation was the most prevalent breakdown. As one example, programmers were forced to wait for long animation sequences to complete in order to test the end of the sequence. To tweak an animation, programmers made a small modification, wrote an event to run the animation when a key was pressed during runtime, ran the program, viewed the animation, and repeated. Also, to test a program's response to an event in a specific world state, the programmer had to manually recreate the world state, and cause the event to occur. One possible solution to these problems would be a *timeline visualization* of events and methods in the world with the ability to click and zoom on objects, events, and time periods. Programmers could then recreate problems and directly associate a world state with specific code.

Debugging Problems

Debugging breakdowns occurred when programmers had difficulty answering debugging questions of the form "when," "why," and "why *not*." Questions such as "when was the last time this object moved?" were difficult to answer; and the timeline visualization discussed earlier could provide immediate access to this information. Answering other questions of the form "why" and "why not," were highly involved debugging tasks. The system could use simple heuristics to answer these. For example, in reply to "why did this object move?" the system could show the code that last moved the object.

FUTURE STUDIES

Future work will involve further analyses of the data obtained in these CI, as well as CIs with novice and expert programmers new to Alice. Though many of the breakdowns identified in our current study were not specific to event-based languages, we expect to find many new difficulties in our future studies.

FUTURE PROGRAMMING ENVIRONMENT

Using these observations as user interface design guidelines, as well as features described in previous work [4] [3] [10], we propose to create a set of new features and tools, and add these to the Alice environment. The features we currently envision include the following.

To help with the **construction** of code, we envision the system including:

- *Demonstrational techniques* [5], so that programmers can show the system the desired dynamic behavior using example objects, and have the appropriate code generated and inserted into the program. Demonstrational techniques will also help to automate repetitive tasks, such as creating a large number of objects that have similar properties.

- *Checkpointing and multi-level selective undo*, so that programmers can more easily try out different approaches and can back up or selectively undo some edits if they discover that the new code does not fix the problem at hand.
- *Smart cut-and-paste* to help with code re-use, so that when the programmer copies code, the system will detect differences between the old and new locations, and help the programmer edit the code appropriately based on the new context.
- *Agents to help with errors*, that will look for common error situations, as well as provide better support when there are run-time errors, by suggesting possible corrections.
- *Special-purpose editors*. Some parts of the code might be easier to write if the student could use a different format from the conventional code. For example, our prior research showed that a form-based presentation helped users construct more correct Boolean expressions [7] compared with conventional AND-OR-NOT expressions. We will also explore providing equation editors to help programmers convert math and science formulas into code.

To help with the **understanding, testing, and debugging** of code, we envision the system including:

- *Full visibility of data*, so programmers can see what is happening at run time. This will include simple ways to map values to properties of visible objects. For example, the user might want to temporarily use an object's color to visualize whether two objects are considered to be touching.
- The ability to *pause on any program event*, which includes conventional breakpoints, but also the ability to pause when there is user input, when an object or variable changes value, when an object of a particular type is created or deleted, etc.
- *Changing values at run-time* to see what would happen, with the ability to have these changes reflected into code.
- *Backing up a step and running the program backwards*, to find what happened before the current point.
- *A timeline visualization* to show important events during the running of the program. Collisions, input events, method invocations, parallel process activities, and many other events can be shown on the time-line. Users will be able to scroll the time line backward and forward in time, which will show the state of the code's variables and the 3D world at any point of the execution. It will also support zooming on events related to particular objects, code, and periods of time.
- *Checkpointing of run-time state* so users can repeatedly execute code that happens after a certain point. Often, users will need to debug behaviors that occur in the middle or end of a long execution run. Checkpoints will provide an easy way to jump directly to the point in the execution that needs to be tested.
- Support for "*why*" questions, so the programmer can determine a sequence of events that led to a variable or object having its current value, or why an event handler was called.
- Support for "*why not*" questions, which will use heuristics to propose reasons why some event did *not* happen. For example, if the programmer asks why an object is not visible, there are only a limited number of possible causes, which the system can test automatically (e.g., its position is outside of the window, it has zero size, it is behind the camera, it is occluded by another object, it is white on a white background, etc.).
- *Search capabilities*, such as searching for any variable with a particular value, or any object with certain properties.

The effectiveness of these tools will be tested empirically. The expected results from this work are new techniques and user interfaces applicable to all event-based programming environments.

SUMMARY

As the workshop call stated, "wide-spread penetration of interactive software systems has raised an increasing need for better environments for building applications." Since end users in particular are creating interactive applications using event-based languages, we need to better understand how to support event-based programming, testing, and debugging. Our current observational studies aim to help this understanding, in an effort to design novel environmental tools to support these tasks.

Both Brad Myers and Andrew Ko would like to participate in this workshop. Dr. Myers has been leading the Natural Programming Project and can provide a perspective on many years of work in the area. Andrew Ko is just starting to work on studying programmers, and has performed the studies of Alice described here. We look forward to participating in this workshop in order to discuss alternative methodologies for studying programming environments, novel event-based languages, as well as psychological studies of end users that may guide our research.

SHORT BIBLIOGRAPHY OF AUTHORS

Brad A. Myers is a Senior Research Scientist in the Human-Computer Interaction Institute in the School of Computer Science at Carnegie Mellon University, where he is the principal investigator for various research projects including: Natural Programming, Silver Multi-Media Authoring, the Pebbles Hand-Held Computer Project, User Interface Software, and Demonstrational

Interfaces. He is the author or editor of over 230 publications, including the books "Creating User Interfaces by Demonstration" and "Languages for Developing User Interfaces," and he is on the editorial board of five journals. He has been a consultant on user interface design and implementation to about 40 companies, and regularly teaches courses on user interface software. Myers received a PhD in computer science at the University of Toronto where he developed the Peridot UIMS. He received the MS and BSc degrees from the Massachusetts Institute of Technology during which time he was a research intern at Xerox PARC. From 1980 until 1983, he worked at PERQ Systems Corporation. His research interests include user interface development systems, user interfaces, hand-held computers, programming by example, programming languages for kids, visual programming, interaction techniques, window management, and programming environments. He belongs to SIGCHI, ACM, IEEE Computer Society, IEEE, and Computer Professionals for Social Responsibility.

Andrew Ko is a first year PhD student in the Human-Computer Interaction Institute at Carnegie Mellon University. As an undergraduate at Oregon State University, he worked with Margaret Burnett on the Forms/3 end-user visual spreadsheet environment, studying the use of cognitive walkthroughs for experiment design and conducting usability studies of novel interfaces for spreadsheet testing. For his undergraduate thesis, he investigated end-user and expert programmer problem solving strategies in a programmable statistics environment. Currently, he is emphasizing usability and design first by extracting design requirements for event-based programming environments by conducting contextual inquires with expert, intermediate and novice programmers.

REFERENCES

1. Beyer, H. and Holtzblatt, K., *Contextual Design: Defining Custom-Centered Systems.* 1998, San Francisco, CA: Morgan Kaufmann Publishers, Inc.
2. Corritore, C.L. and Wiedenbeck, S., "An Exploratory Study of Program Comprehension Strategies of Procedural and Object-Oriented Programmers." *International Journal of Human-Computer Studies*, 2001. (54): pp. 1-23.
3. Fry, C., "Programming on an Already Full Brain." *Communications of the ACM*, 1997. **40**(4): pp. 55-64.
4. Lieberman, H., "The Debugging Scandal and What to Do About It." *Communications of the ACM*, 1997. **40**(4): pp. 26-78.
5. Myers, B.A. "Demonstrational Interfaces: A Step Beyond Direct Manipulation," in *Watch What I Do: Programming by Demonstration.* 1993. Cambridge, MA: MIT Press. pp. 485-512.
6. Navarro-Prieto, R. and Canas, J.J., "Are Visual Programming Languages Better? The Role of Imagery in Program Comprehension." *International Journal of Human-Computer Studies*, 2001. (54): pp. 799-829.
7. Pane, J.F. and Myers, B.A. "Tabular and Textual Methods for Selecting Objects from a Group," in *Proceedings of VL 2000: IEEE International Symposium on Visual Languages.* 2000. Seattle, WA: IEEE Computer Society. pp. 157-164.
8. Pane, J.F. and Myers, B.A. "The Impact of Human-Centered Features on the Usability of a Programming System for Children," in *Extended Abstracts for CHI'2002: Human Factors in Computing Systems.* 2002. Minneapolis, MN: pp. 684-685.
9. Pausch, R., Burnette, T., Capehart, A.C., Conway, M., Cosgrove, D., DeLine, R., Durbin, J., Gossweiler, R., Koga, S., and White, J., "Alice: A Rapid Prototyping System for 3D Graphics." *IEEE Computer Graphics and Applications*, 1995. **15**(3): pp. 8-11.
10. Ungar, D., Lieberman, H., and Fry, C., "Debugging and the Experience of Immediacy." *Communications of the ACM*, 1997. **40**(4): pp. 39-43.

Towards a Research Agenda in End User Development

Fabio Paternò
ISTI-CNR
Via G.Moruzzi 1
56100 Pisa, Italy
+39 050 316 3066
fabio.paterno@cnuce.cnr.it

ABSTRACT
This paper proposes a research agenda in the area of End User Development. Its purpose is to stimulate discussion rather than to provide a definitive solution. It is based on the discussion carried out in the EUD-Net Network.

Keywords
End User Development, Research Agenda, End User Modelling and Programming.

INTRODUCTION
While some substantial progress has been made in improving the way users can access interactive software systems, developing applications that effectively support users' goals still requires considerable expertise in programming that cannot be expected from most citizens. Thus, one fundamental challenge for the coming years is to develop environments that allow people without particular background in programming to develop their own applications, with the ultimate aim of empowering people to flexibly employ advanced information and communication technologies within the future environments of ambient intelligence.

We think that over the next few years we will be moving from *easy-to-use* (which has yet to be completely achieved) to *easy-to-develop interactive software systems*. Some studies report that by 2005 there will be 55 million end-users, compared to 2.75 million professional users [BAB00].

End-user development in general means the active participation of end-users in the software development process. In this perspective, tasks that are traditionally performed by professional software developers are transferred to the users. They need to be specifically supported in performing these tasks. New environments able to seamlessly move between using and programming (or customizing) can be designed.

At the first EUD-Net workshop held in Pisa a definition of End User Development was identified: "*End User Development is a set of activities or techniques that allow people, who are non-professional developers, at some point to create or modify a software artefact*".

REQUIREMENTS
Software practices – including use, design, development and maintenance – seem to change character around adaptable systems. As tailoring interfaces allows the user to change the program, the border between use and design gets blurred. As use, tailoring, adaptation, maintenance and development projects get intertwined, they have to be co-ordinated in a different way. Traditional borders defining a project in this manner are often too rigid. Applications must be tailorable, adaptable by their users or by domain experts to meet the changing requirements. End User development also means the adaptation and further development of software in response to individual preferences, changing co-operative work practices as well as the developing business practices.

Model-based approaches can be useful for end-user development because they allow people to focus on the main concepts (the abstractions) without being confused by many low-level details. Through meaningful logical abstractions it is also possible to support participation of end-users already in the early stages of the development process. Optimally, model-based software development is to be combined with prototype-oriented development.

In traditional software engineering, the Unified Modelling Language (UML) [OMG] has become the de facto standard notation for software models. UML is a family of diagrammatic languages tailored to modelling all relevant aspects of a software system; and methods and pragmatics can define how these aspects can be consistently integrated. UML is a general-purpose modelling language that comes with built-in extension mechanisms and a so-called profiling mechanism to support tailoring, adaptation and extension for specific development processes and application domains. These fundamental concepts of the UML have started to be investigated for suggesting its extension to user interaction and user interface modelling. UML's extensibility may as well be deployed to design an end-user modelling profile containing user-oriented language elements and domain-specific (end-user) profiles.

Like programming, modelling requires the availability of suitable and usable languages and supporting tools to be effective. Visual modelling languages have been identified as promising candidates for defining models of the software systems to be produced. UML and related tools such as

Rationale Rose are the best-known examples. They inherently require notions of abstractions and should deploy concepts, metaphors, and intuitive notations that allow professional software developers, domain experts, and users to communicate ideas and concepts. This requirement is of prominent importance if models are not only to be understood, but also used and even produced by end users. Language usability needs to be empirically analysed.

Naturally, end-user development requires that the tasks intended to be performed by end users have to be addressed and solved beforehand on the more technical, underlying levels. For example, dynamic reconfiguration of system components (e.g. customisable interoperability) by end users must be supported by underlying component models and system architectures that technically enable this interoperability and component reuse. Only after these operations have been realized on a technical level, can such tasks be effectively performed by end users on a more abstract level.

The trade-off between expressiveness and usability is a general concern in the area of end-user development. For example, there are several direct manipulation environments that allow easy assembling of applications. Simple functionality can thus be easily constructed using the graphical interface. However, more sophisticated functionality and complex behaviour that go beyond simple applications or early prototypes require the use of a scripting language that is typically integrated with the visual elements of the graphical user interface, thus requiring specific programming skills.

End-user development has some important effects on other, more technical levels of software. Administration of customisable systems is far more complex than dealing with mostly standardized configurations and implementations. Component-based development raises issues of standardized interfaces, interoperability, etc. Adaptability in general requires ensuring program correctness and not invalidating other required properties. Therefore, adequate means for defining semantics and analysing system properties in the presence of adaptation are needed to restrict possible modifications.

The need for end user development is clearly emerging, some approaches in the area of end user development (such as programming-by-example), have long been considered from a research point of view [C93], and are now started to be used in widely used software such as MS-Excel.

THE RESEARCH AGENDA

In the first semester of the network life we have been able to develop a more detailed discussion of promising research lines in these area. In addition to these, we have to stress that traditional software engineering metrics do not seem suitable to evaluate the effectiveness of end user development environments. Metrics that more closely reflect value added to people and organizations should be identified.

Incrementally formalised development

Often the initial model is the result of brainstorming by either one single person or a group. Usually people start with some paper or whiteboard sketches. This seems an interesting application area for intelligent whiteboard systems [LM01] or augmented reality techniques able to detect and interpret the sketches and convert them into a format that can be edited and analysed by desktop tools.

Most end-user development will probably benefit by the combined use of multiple representations that can have various levels of formality. The possibility of developing through sketching can be highly appreciated in order to capture the results of early analysis or brainstorming discussions. Then, there is the issue of moving the content of such sketching into representations that can more precisely indicate what artefact should be developed or how it should be modified.

A similar approach is followed when people try to use informal descriptions in natural language such as scenario descriptions for obtaining more structure representations. An example is in [PM99] where starting from informal scenarios it is shown how to use the information that they contain to obtain more general task models.

Vocal interaction can play an important role in this respect as well. Support for vocal interaction is mature for the mass market. Its support for the Web is being standardised by W3C [A01]. The rationale for vocal interaction is that it enables the development of applications suitable for both Internet and wireless communication, it makes practical operations more natural and faster, and it makes possible multi-modal applications (graphic and/or vocal).

End User Modelling and Programming

In order to ease the development process people need high-level languages that highlight the important aspects to consider. To support them there is a need for multi-layers approaches able to map abstract functions and concepts onto low-level programming constructs. The starting point of a development activity can often vary. In some cases people start from scratch and have to develop something completely new, in other cases people start with an existing system (often developed by somebody else) and need to understand the underlying conceptual design in order to modify it or to extend it to new contexts of use. Thus, a general development environment should be able to support a mix of forward and reverse engineering processes. This calls for environments that can support various transformations able to move among various levels (code, specification, conceptual description) in both a top-down and bottom-up manner and to adapt to the foreseen interaction platforms (desktop, PDA, mobile phones, …) without duplication of the development process.

Another important aspect to consider is the psychology of programming that indicates what important psychological aspects can have an impact on this activity. Whatever the formalism or programming language used, the underlying paradigm has a great influence on what programmers can express and with what ease. However, there is little information available on the cognitive features of programming paradigms. As both users and designers of those paradigms, computer scientists are usually more interested in exploring the formal properties of paradigms and formalisms, and the related software engineering issues, than in understanding why they prefer such or such paradigm. However, it is a fact that paradigms are more or less easy to learn and apply depending on the task. A better understanding of the underlying cognitive issues would be as important to end-user programming as a better understanding of the properties of interaction styles is to user interface design. The graphical programming environment that was provided with the first versions of Lego's Robotic Invention System is a good example. Its visual language is clear and easy to manipulate for children.

Integration of Visual Modelling and Innovative Interaction Techniques

Recent years have seen a large adoption of visual modelling techniques in the software design process (example are Rationale Rose Together, Magic Draw, Enterprise Architect, Poseidon for UML), but there are also research environments publicly available such as CTTE [MPS02]). However, we are still far from visual representations that are easy to develop, analyse and modify, especially when realistic case studies are considered. The application and extension of innovative interaction techniques ([BMA01]), including those developed in information visualization (such as semantic feedback, fisheye, two-hand interactions, magic lens…), can noticeably improve their effectiveness.

Flexibilization of software development

There is an increasing tendency to remove the barrier between design and use. These activities tend to merge. Agile methods [C02] address such issues. The agile approach focuses on delivering business value early in the project lifetime and being able to incorporate late-breaking changes in requirements by accentuating the use of rich, informal communication channels and frequent delivery of running, tested systems, all while devoting due attention to the human component of software development. Proponents of the agile approach say that these practices lead to more satisfied customers and a superior success rate of delivering high quality software on time.

The concept of agility, referring to development methods that are more people oriented than process oriented, and emphasizing flexibility and adaptability over full description, can have a strong impact on software engineering.

Another important support in this respect is given by environments for component-based deployment. Software components and component-based design have received much attention in the software engineering and application development communities over the past years. Software components allow systems to be built by starting from high-level reusable building blocks instead of writing program statements in a general purpose programming languages. One of the great promises of composition is that it has the potential to be performed at runtime (i.e., when the system is in use). Connecting two components only requires 'glue code' (i.e., a high-level script) that records the connections between the components. However, the integration of software components by end-users to make new applications is far from trivial.

A critical bottleneck is that end users need to know what interface methods are defined on the various components and how they must be called to realise the integration of two components. Interestingly, a model for software component integration is Lego toy construction. Lego allows great flexibility in how two components can be coupled together. By keeping interfaces (connection points) general, each brick can connect to many other bricks (of different shapes). This generality is approached in software by method interfaces that cater to many combinational needs. However, the cost of generality (advantageous for component developers) is paid at the expense of end-user mastery because the connection points will often not have intuitive (domain-specific) names and may require parameters to be specified so that they can be used in many combinations.

Architectural concepts for flexible systems

The need for flexible environments has implications also at the architectural level. One example occurs when we consider adaptivity for ubiquitous computing. Adaptive environments help users to interact with their applications by dynamically modifying their behaviour and functionality while taking into account various aspects: user behaviour, external environment, tasks to perform, interaction device and so on. In this area it is of particular interest to design applications able to address the many possible use environments, on-the-fly dynamic configuration of interaction devices and the rapidly increasing availability of many types of devices (ranging from small phones to large flat displays, including embedded computers in cameras, cars, ..). This development will continue and computers will start to vanish into the environment, and computational power and networking capabilities will then become ubiquitous.

This engenders the need for context-dependent applications that can be supported by both adaptive and adaptable techniques. When the system is adaptable it can be tailored (manually) by the end users to fit their needs, work practices, business goals, etc. The results will enhance user competence and awareness of the system, allowing for personal adaptations, with the creation of new functionalities and user interface features. An important aspect is that adaptations should be as unobtrusive as possible (not interfering with the task itself). Thus, more work is needed on user modelling and how it can improve efficiency and effectiveness in end-user programming.

One of the main challenges for the success of ubiquitous computing is the design of personalised user interfaces and software that allows easy access to relevant information and that is flexible enough to handle changes in user context and availability of resources.

Application domains (home applications, …)

A number of application domains seem particularly suitable for end user development environments. An example is given by home applications. (Almost) everybody has a home. It is possible to electronically interact with many devices in a house. This means that the house potentially can become one of the most popular applications for information technology. Thus, it can be a domain where the need for end user development will be particularly important.

Cooperative end user programming.

Cooperative end user programming involves environments that help end users to support each other in programming, to share their programs and modified shared programs. Given the fact that users typically have very different skills and interests in tailoring or programming, there are many different divisions of labour with regard to these activities. Therefore, it is important to provide technical features which support cooperative end-user programming. An important aspect of this is to develop annotation and manipulation tools that act on partial designs, allowing users to customise software directly in individual or cooperative working environments. In some cases it will also be important to consider that the cooperation will occur across people with various levels of expertise.

Tailoring environments

The possibility of tailoring applications is particularly important in some end user development environments. In these cases users modify existing environments in order to tailor their functionality according their needs. This is important also in mobile applications.

Software engineering for end users

The definition of end user development is based on the differences between end- users and professional programmers and software engineering. There are differences in training, in the scale of problems to be solved, in the processes, etc. However, there are some similarities. Some of those similarities are to be found in the life cycle of the developed software artefacts. For instance, managing the successive versions of a piece of software will most probably become a problem for end users. Version management is already a problem with word processor documents. However, one cannot expect an end user to apply the techniques provided by the software engineering field. Software engineering methods and tools require knowledge of abstract models that end users do not have. They imply the use of methods and tools that require specific training. They probably consume more time than an end user is willing to afford, etc. In addition, not all problems from software engineering are equally important for end users: team development techniques are most probably beyond end users needs. Consequently, an interesting line of research consists in identifying new sets of techniques and tools that would be the counterpart of software engineering for end users: software crafting.

CONCLUSIONS

After a discussion of the motivations and more relevant requirements for end user development identified so far, there is an early discussion of promising research lines that if adequately supported can provide important results in obtaining effective end user development environments. We hope it can be useful to stimulate further discussion at the CHI workshop.

ACKNOWLEDGMENTS

This work has been supported by the European Network of Excellence EUD-Net (http://giove.cnuce.cnr.it/eud-net.html). Support from the European Commission is gratefully acknowledged. I wish to thank also all the members of the networks for their useful contribution to the preparation of this first research agenda.

REFERENCES

[A01] Ken Abbott, Voice Enabling Web Applications: VoiceXML and Beyond. ISBN: 1893115739, APress L. P., 2001

[BMA01] Beaudouin-Lafon M., Mackay E., Andersen P., at al., CPN/Tools: A Post-WIMP Interface for Editing and Simulating Coloured Petri Nets. Proceedings ICATPN 2001. pp.71-80, Springer Verlag LNCS N. 2075.

[BAB00] Barry W. Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece, Software Cost Esimation with COCOMO II, Prentice Hall PTR, Upper Saddle River, NJ, 2000.

[C93] Allen Cypher, ed. Watch What I Do MIT Press 1993

[C02] Alistair Cockburn. Agile Software Development. Addison Wesley. 2002

[LM01] Landay J. and Myers B., "Sketching Interfaces: Toward More Human Interface Design." In IEEE Computer, 34(3), March 2001, pp. 56-64

[MPS02] Mori G., Paternò F., Santoro C., CTTE: Support for Developing and Analysing Task Models for Interactive System Design, to appear in *IEEE Transactions in Software Engineering*, September 2002.

[OMG] OMG Unified Modeling Language Specification, Version 1.4, September 2001; available at http://www.omg.org/technology/documents/formal/uml.htm

[PM99] F.Paternò, C.Mancini, Developing Task Models from Informal Scenarios, Proceedings ACM CHI'99, Late Breaking Results, pp.228-229, ACM Press, Pittsburgh, May 1999.

Position paper to the workshop on End User Development, CHI 2003

Prototyping Interactivity before Programming

John Sören Pettersson, Information Systems, Karlstad University, 651 88 Karlstad, Sweden
john_soren.pettersson@kau.se

ABSTRACT

A system for testing interaction design without the need for programming is described. It is claimed that this tool will make end-user driven development possible by introducing laymen as designers and testers, not as programmers.

INTRODUCTION

In the call for the workshop it is pointed out that "The interactive richness of new devices has created the potential to overcome the traditional separation between end users and software developers." The invitation also states that "There are studies that indicate that the end-user programming population will be growing at more than 10 percent per year worldwide". A substantial growth seems indeed granted and we do in fact see more advanced programming tools being developed. However, the implication in the invitation to this workshop is that it is the programming environments which need to be refined. In this position paper I will argue for the pertinence of refining another part of systems development: the phase of conceptualization in which the interaction designed is preconceived.

My starting point is that, in general, end-user involvement in software development processes brings better, more usable software about. I will argue for an end-user driven development. However, I do not mean to imply that we can skip usability tests altogether by trashing the expert programmer, for instance replacing her with some any-user who programs by demonstrations. Whatever software produced by end users, such software might be used by others than the originator. This raises the same questions about usability as encountered in professional software development. Who is to decide (design) for whom?

I am not arguing against end user programming, but I want to direct the attention to issues which should come earlier in the development process. How do users conceptualize users' needs? Especially, how are users' needs concerning interaction automation conceived by people with scant insight in automatically generated responses?

At the workshop I would like to present the ideas behind a tool, Ozlab, which was demonstrated at the last NordiCHI conference in Denmark held in October 2002 (Pettersson & Siponen, 2002). Below, I borrow from that presentation (hence the format of this position paper!), but also add some information on end-user driven conceptual development of interfaces.

The Ozlab software has been developed at Karlstad University in order to make it easy to test already on a conceptual stage the interactivity of graphical user interfaces, GUIs. The term GUI as used here does not mean simply drop-down menus and dialogue boxes but more graphically and spatially oriented interaction.

BASIC TECHNIQUE: WIZARD OF OZ

There is an experimental technique often employed in language technology called 'Wizard of Oz' (Dahlbäck et al. 1993). In Wizard-of-Oz experiments a test person thinks he writes or speaks to the computer in front of him when in fact the test manager sits in the next room interpreting the user's commands and providing appropriate responses (see Figure 1).

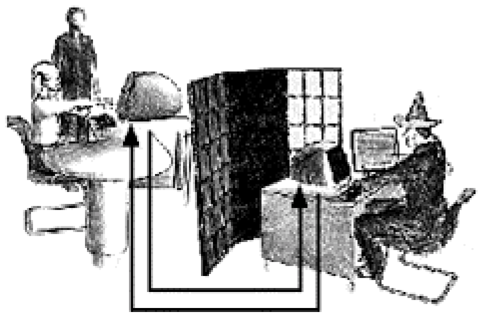


Figure 1. In Wizard-of-Oz experiments the test manager sits in the next room interpreting the user's commands and provides the system's responses. The system functionality is thus faked in a way that makes it possible to test system proposals without programming prototypes.

The reason why this deceptive technique has been popular in language technology is simply that natural language processing of either text or speech can be simulated even when there is no unit available that understands natural human language. Thereby dialogue structure can be tested before one decides on how clever the automatic interpretation has to be (e.g., whether to analyze individual words merely or also syntactic structures). Automatic interpretation of text or speech is difficult and the Wizard-of-Oz technique thus gives systems developers a chance to test systems before it is even possible to make them. This kind of manual prototype could act as a stand-in for working prototypes.

Extending the ordinary Wizard of Oz

It seems as if the mere deceptive property of this technique has not been fully exploited. Since the system looks real to the test user, one could use Wizard-of-Oz mock-ups to test design ideas when there are reasons to believe that simple tests by sketches and slides, as preferred and recommended by usability experts (inter alia Klee, 2000; Cato, 2001, pp. 78ff), will not provide the right responses. For instance, Molich (2002) notes that test subjects may behave less explorative when testing a paper mock-up because they do not want to bother the test leader. It is also a fact that the look-and-feel of a paper mock-up will differ very much from the final product. The latter point is especially important when testing design alternatives on children or when the person (company) who orders the system is inexperienced and need a way to see various variants of his own suggestions in working practice, i.e., he needs to see the system proposals in interaction with users.

End users developing and testing concepts

When it comes to end-user driven development, the importance of making the interactivity explicit must not be underestimated. Naturally, one could argue that simple programming tools would provide means for the non-professional programmer to try out ideas. However, such testing of ideas entails two drawbacks which are perhaps easily overlooked:

1) Programming an interaction in advance will not inform the designer of the user's needs in the same way as a real conversation does. In a Wizard-of-Oz experiment, the designer *is* the system and will get a better feeling for the computer responses needed by the user.

2) A programmed, autonomously functioning system is too easily seen as 'functional' also in a task-oriented sense, even if different design alternatives have not been tested. Every programmer (and every designer) has a tendency to refine rather than redesign. For instance, interaction designer Bill Verplank notes this in an interview in Preece & al. (1994, p. 467.):

"What happened was that they set out with some very fixed notions early on, and simply kept refining them, so there was no real comparison of alternatives. There was really only one idea and they refined it and refined it […] No-one was ever very satisfied with the design and what I attribute that to was that they had a working prototype even before they decided what the product was going to be."

From this perspective there would appear to be a problem with any simplified programming technique, because it will always be tempting to stick to the first shot because it 'works'. An Oz prototype never 'works'. On the other hand, it is easy to change the interaction design and every test-run informs the designer in a way which is simply not conceivable with working prototypes. When we were constructing Ozlab, we used three inexperienced designers (i.e. inexperienced as UI designers) to run tests on their own designs. They matured very quickly in their roles as wizards, and it was obvious that for them it was quite a thrill to interact with their clients through their designs (Pettersson, 2002b). In this sense Wizard-of-Oz prototypes *really* work.

I could add that the temptation to test-run one's prototypes on your own is very small when dealing with Oz prototyping. By being manual, any Oz test will always ask a reason for why it is performed. It will seem futile to test it on your own when the prototype is manual (this holds for single-person test by having two screens next to each other as well as for peer-testing within the design group). Furthermore, the thrill noted already on our first wizards when they were conducting a Wizard-of-Oz test is very satisfying from a methodological perspective: it indicates that user-involvement is preferred by naive designers. (Pettersson, 2002a)

MAKING GUI TESTING SIMPLE

A problem, however, with modern interactivity tests, is that they are not as easily performed as a Turing test. It is not enough with voice or text input followed by voice and text output from the computer. Graphics has to be included. Therefore, when we designed Ozlab, it was to make possible fluent tests with ordinary PC interfaces. The graphical details have to be there during a test – that requirement cannot be conjured away. But ordinary GUI behaviors like moving objects and disappearing / reappearing objects are already pre-programmed in Ozlab, so that the wizard only connects such functionality to the objects when putting the graphics in place. This functionality then allows for easy manipulation of the user interface during a test. (However, I have to admit that the building of the prototype file is done in Macromedia's Director, which contains a plethora of advanced functions, a fact which is bewildering for a naive designer even if they do not have to use them.)

The 'graphical' input channel

Some researchers and system developers have been conducting experiments described as 'multimodal', often implying that the test user has access to more than a single input channel. For our Ozlab we have focused very much on the ordinary PC set-up with mouse input. But experiments with keyboard input as well as simulated mobile phone output have also been conducted.

Primarily, Ozlab is intended for the *interaction* to be simulated. It does support showing videoclips on the test user's monitor, but a 'multimedia' piece like that is not really dependent on interaction with the user.

We have concentrated on how to let the wizard respond graphically by direct manipulation of the users GUI. (Voice response is of course also very easily produced in any Wizard-of-Oz test including Ozlab-tests as explained in the following section.)

Graphical interaction vs. linguistic interaction

The graphical output does not demand the same exactness as text output in ordinary Wizard-of-Oz prototyping where spelling mistakes will be very revealing. This fact makes it easier to test graphical 'dialogues' than to test linguistic dialogues.

However, it should be noted that pre-written sentences can easily be made visible in Ozlab like any other graphical object. Furthermore, Ozlab allows for free text output as well, and also free voice responses through a voice-disguising unit. The latter has been very much used by some of our wizards to test spoken comments to graphical demonstrations. But Ozlab is not primarily designed to support linguistic output in contrast to the set-up in the above-referred study by Dahlbäck et al. (1993). I felt the need for a WOZ tool for simulating the graphical interaction made possible by direct manipulations in GUIs.

Short time-to-test and testing adaptive UIs

It should be observed that because not much more than the graphics and a few wizard-supporting functions attached to each graphic object is needed before test trials can be run, the Ozlab system could be used for improvisation – explorative experiments – as well as for ordinary tests of a pre-defined response scheme. A combined form will allow for testing various parameter-settings in adaptive interfaces.

Some further data about Ozlab

The Ozlab system was operational in August 2001 and has since then gone through several revisions. It has been used to let inexperienced designers as well as persons of various levels of design expertise make interaction designs. Short descriptions on works so far conducted with the Ozlab system is found on the (more or less up-dated) project web site www.cs.kau.se/~jsp/ozlab.

It should perhaps be pointed out, that the Ozlab software could be run outside the laboratory on any pair of laptops or ordinary PCs, which makes it possible to make mock-up prototyping on site. Of course, some of the wizard functionality is affected of the locality. Most notable is this for any guiding voice from the faked system when such voice messages are not pre-recorded or delivered by text-to-speech systems, but instead made by the wizard herself. For such a true WOZ set-up to work, the test user has to be out of hearing distance of the wizard.

ETHICAL AND IMPLEMENTATIONAL PROBLEMS

Naturally, it should be recognized that faking a system as in the Wizard-of-Oz experiments may cause some ethical problems, especially if these experiments are to be performed by people who are not schooled in usability testing. Furthermore, the interaction taking place during a Wizard-of-Oz experiment may not always be easily implemented in a computer program. In particular, this may be the case if linguistic interaction takes place. Compared to linguistic responses it is easier to keep graphical responses within the limits of what may be implemented.

A further defense of Oz prototyping could be derived from authors stressing the need to keep design and implementation apart (Cooper, 1999; Löwgren 1995). End user design as described above could be followed by professional implementation. Communicating the Oz prototype to the computer experts could probably be done with video and screen recordings. Likewise, a multimedial form of design description makes it possible to communicate in a precise manner also to less professional developers, like local champions of IT.

REFERENCES

Dahlbäck, N., A. Jönsson & L. Ahrenberg (1993) Wizard of Oz Studies – Why and How. *Knowledge-Based Systems*, Vol. 6, No. 4, pp. 258-266.

Cato, J. (2001) *User-Centered Web Design*. Addison-Wesley, Harlow, England.

Cooper, A. (1999) *The Inmates are Running the Asylum*. SAMS publishing.

Klee, M. (2000) Five paper prototyping tips. UIE Newsletter *Eye for Design*, March/April. Also at http://world.std.com/~uieweb (inspected 2001-03-07).

Löwgren, J. (1995). Applying design methodology to software development. In S*ymposium on Designing Interactive Systems, DIS'95 Proceedings*, pages 87–95. New York: ACM Press.

Molich, R. (2002) *Webbdesign med fokus på användbarhet*. (trans. from Danish *Brugervenligt webdesign*, 2000). Studenlitteratur, Lund.

Pettersson, J.S. (2002a) "Naïve designers as concept developers and test managers". In *Promote IT2002, Part I. Skövde, April 22-24, 2002*, Eds. J. Bubenko & B. Wrangler. Skövde: University College of Skövde. Pp 136-146.

Pettersson, J.S. (2002b) "Visualising interactive graphics design for testing with users". *Digital Creativity* vol 13 (3): 143-155.

Pettersson, J.S. & J Siponen (2002c) Ozlab – a simple demonstration tool for prototyping interactivity. Demonstration at *NordiCHI 2002*, October 19-23, 2002, Aarhus, Denmark. Pp. 293-294

Preece (1994) *Human-Computer Interaction*. Addison.

Fostering End-User Participation with the Oregon Groupware Development Process

Till Schuemmer
FernUniversitaet Hagen
Computer Science VI - Distributed Systems
Universitaetsstrasse 1
55084 Hagen, Germany
+49-2331-987-4371
till.schuemmer@fernuni-hagen.de

Robert Slagter
Telematica Instituut
P.O. Box 589,
7500 AN, Enschede,
The Netherlands
+31-53-4850-488
robert.slagter@telin.nl

ABSTRACT

This position paper outlines the Oregon Groupware Development Process, which fosters user involvement in all stages of groupware development. It combines several – up to now often unrelated – design techniques such as iterative development, tailoring, or participatory design.

INTRODUCING SUMMARY

Groupware applications are becoming more and more important in the context of distributed collaboration and new enterprise models, such as virtual organizations. However, we still observe a lack of a proper design methodology for groupware applications that heavily involves end-users and fosters reuse of existing knowledge from the area of groupware research.

Thus, we strongly advocate participatory design of groupware, where social experts, technical experts and end users are involved throughout the design process. We propose the use of patterns to reuse existing design knowledge and facilitate communication between the various stakeholders. The process consists of several short design iterations (using prototypes and mock-ups) where various aspects of the system can be discussed, tested, and modified as soon as possible.

Patterns for groupware design play a central role in our approach. We apply these patterns both for developing and for tailoring groupware. Since the collaborating people themselves are the ones most affected if the groupware does not properly support their cooperation, they should be empowered to adapt their groupware systems. As tailoring operations are performed by end users, the groupware patterns need to be in a form that is understandable by these end users. To start such tailoring operations, end users have to *diagnose* (reflect on) their collaboration, discover indications of problems and apply a proven solution, as described in our groupware patterns.

Accompanying our approach, we have defined a new pattern structure that is appropriate for both types of stakeholders. This is of utmost importance, since in our approach (in contrast to other pattern approaches) patterns are used for programming, tailoring and to foster shared understanding throughout the participatory design trajectory. So, in contrast to existing pattern approaches, our patterns should also be suitable to help end-users tailor the behaviour of their groupware application.

DESIGN PATTERNS AND THE OREGON EXPERIMENT

Today, design patterns are widely accepted in the software development community. By means of design patterns, one can describe expert knowledge in the form of rules of thumb. These rules include a problem description, which highlights a set of conflicting forces and a proven solution, which helps to resolve the forces.

Initially, patterns were developed in architecture, used by non-experts in the context of a participatory design process. The foundation for this process lies in the philosophy of Alexander, which becomes clear in [3]:

> "The people can shape buildings for themselves, and have done it for centuries, by using languages which I call pattern languages. A pattern language gives each person who uses it, the power to create an infinite variety of new and unique buildings, just as his ordinary language gives him the power to create an infinite variety of sentences." (p. 167)

Every user of a building or a place should have the freedom to shape the environment, in which he acts. This basic idea was institutionalised in the planning process of the campus of the university of Oregon – the Oregon Experiment [2]. The process defines six basic principles: *organic order, participation, piecemeal growth, patterns, diagnosis,* and *coordination.* Some of these principles can shape a groupware design process. We will explain their application in the Oregon experiment, and investigate how to apply them to groupware development.

Participation ensures that the final users will be part of the planning process and therefore participate in shaping their environments. Alexander defines it as a

> "…process by which the users of an environment help to shape it. The most modest kind of participation is the kind where the user helps to shape a building by acting as a client for an architect. The fullest kind of participation is the kind where users actually build their buildings for themselves." ([2] p. 39)

In the Oregon Experiment, this principle led to a very successful campus design [11]. The University established a user group that contained students, faculty members, and staff. The user group then decided, which projects should be built in the next phases. In a refinement phase, special focus groups (users with specific interests) concentrated on one aspect of the building.

Piecemeal growth. By concentrating on one part at a time, one follows the principle of piecemeal growth. This includes identifying one concrete problem and finding a solution for this problem.

Patterns. To empower end users to find working solutions, they necessarily need a way of accessing previous proven solutions. Patterns fulfill this role. They are the essential language- the Lingua Franca as Erickson calls it [4] in the domain of HCI – that is used in the user group. In the formation phase of the community, the members agree on a common set of patterns (taken from a pattern language [1]). Although patterns are proven solutions, they are not static. Users are encouraged to enhance or correct the patterns. These new patterns will then be incorporated in the community pattern language, as long as the whole community agrees with the adaptation.

Diagnosis is the process of analysing the existing campus regarding aspects that work and aspects that do not work. This includes a phase of reflection: during the use of the environment, users are encouraged to step back and ask themselves, whether or not the environment serves their needs. If not, they are asked to mark the deficits and thus state change requests for the environment.

CURRENT GROUPWARE DEVELOPMENT APPROACHES

The process of groupware development is currently supported by different technologies and methods. These include iterative processes, participatory design, tailorable software design, and the use of software patterns to ease the design of the groupware application. All these technologies and methods have been part of the Oregon experiment for many years now. In this position paper, we concentrate on participatory design and tailorable software design and compare them to the Oregon experiment.

Participatory design is a design approach that “puts people first”. In participatory design (prospective) end users are involved in the design of new systems. The end users are in tight interaction with the designers, but the designers still do all design activities.

Since participatory design is based on the interaction between (prospective) users, social experts and technical experts, these people need a common language to discuss a design and implications of design decisions. Given the different backgrounds of social experts, technical experts and (prospective) users, reaching a shared language is not trivial.

The main difference between the user groups in the Oregon experiment and participatory design in software development is that in the Oregon experiment the users really acted as designers. This ensures that the users' wishes are really reflected in the resulting design. Using a shared pattern language helped reducing communication problems.

We see a possibility to improve groupware design practice by providing users with a means to express their ideas throughout the whole design process and come up with their own groupware design. As in the Oregon experiment, patterns play an important role in this phase.

Patterns have become very prominent in software design. But the design pattern approaches concentrated on the need of patterns for software developers. This is different to the intended way, as presented in the Oregon Experiment. The patterns of Alexander et al. concentrated on the user, who needs to solve a very personal problem. We therefore argue that groupware design should use patterns to inform the *users* about proven solutions in the field. This implies that groupware patterns need to be in a form that is understandable by end-users.

Tailorable software can be adapted within the context of its use [6]. In a collaborative setting the collaborating people themselves, the tasks they perform together, and the context in which they perform these tasks all change over time. This all contributes to changing requirements (or forces) on the technology support. One can observe that groupware systems must *evolve* because they cannot be completely designed prior to use. This evolution has to take place at the hand of the users, as they are the *owners* of the problems. The systems must therefore be designed for modifications, to suit evolution of use [10]; ([13]; [12]).

Tailorable groupware offers end-users the possibility to adapt system behaviour as well as the look and feel of the system in the context of collaboration, not as a separate design activity. The challenge here is to provide opportunities for tailoring that are appropriate for the people who need to make changes. The way tailoring options are presented to users should match their mental model, since the tailors have to understand when and how they should adapt their software application.

In the Oregon experiment, tailoring was described as process of diagnosis and repair. Repair meant that the users proposed the application of high-level patterns. Anyhow, end-users were not able to implement the patterns since changing buildings requires special builder's skills. Software components could be user-pluggable, if the process of applying or integrating them is intuitive to the user. The results of tailoring operations were reintegrated in the Oregon process by adapting patterns.

The principle of *diagnosis* is present in most iterative processes. It should also be present as end-user reflection: the collaborating people reflect at regular intervals whether the provided technical support still matches their tasks.

THE OREGON GROUPWARE DEVELOPMENT PROCESS

We propose a groupware development process that combines all four process principles, which we call the Oregon Groupware Development Process since it was inspired by Oregon experiment. It intends to foster end-user participation, pattern-oriented transfer of design knowledge, piecemeal growth in form of short iterations, and frequent diagnosis or reflection that leads to an improved groupware system.

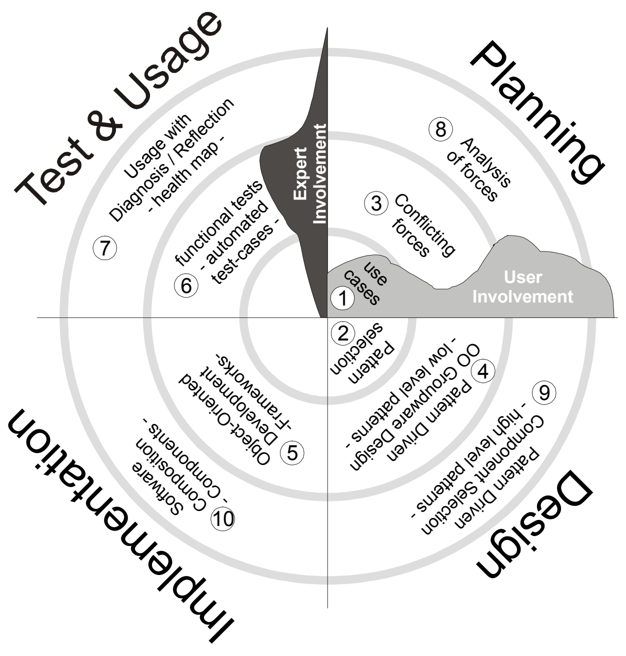

Test & Usage
Planning
Design
Implementation
Expert Involvement
User Involvement
1 use cases
2 Pattern selection
3 Conflicting forces
4 OO Groupware Pattern Driven Design - low level patterns -
5 Object-Oriented Development -Frameworks-
6 functional tests - automated test-cases -
7 Usage with Diagnosis / Reflection - health map -
8 Analysis of forces
9 Pattern Driven Component Selection - high level patterns -
10 Software Composition - Components -

Fig. 1. The Oregon Groupware Design Process

Figure 1 shows the different phases of the process. It suggests three different kinds of iterations. In the following paragraphs, we will explain these three kinds of iterations (numbers refer to the step number in figure 1). In the actual execution of the process, each kind will be executed many times (like in the spiral model).

Throughout all iterations, the users participate and work with a shared groupware pattern language (first patterns of this language are available at the groupware-pattern catalogue [9]). This language contains patterns on two levels: low-level and high-level patterns. Low-level patterns describe solutions, which the end-user will not be able to apply alone. One reason for this might be that the solution cannot be encapsulated in a high-level component, which means that the functionality crosscuts the existing component structure. High-level patterns can be implemented by adding groupware components (without the need of changing the component's code). If the process of adding the component is easy enough end users can execute these patterns. Macro-level patterns are special high-level patterns that describe the combination of several high-level patterns.

In contrast to traditional participatory software design, our groupware patterns approach aims to provide users with profound design knowledge. This knowledge empowers the end-user to act as a designer and solve some issues, without having to escalate these issues to a designer.

The innermost inceptive iterations comprise the activities of (1) use-case analysis and (2) the selection of appropriate patterns. First, the users make up their mind on the usage of the system by specifying simple use cases. These can be stories (for users, who are not familiar with formal use-cases) or success scenarios, which describe the use-case's story in a more formal way.

The use cases then drive the selection of initial patterns from the groupware-patterns catalogue, which serve as starting points for exploring the different forces in the area. During the inceptive iterations, the end-users will be highly involved. According to their knowledge in groupware patterns, they can perform the iterations without any expert guidance. But in most cases, there will be social experts and technology experts, who support the users in writing stories, discovering the social functions that need to be supported, and pointing the users to appropriate sets of patterns. One result of inceptive iterations is a common pattern language, which then eases the process of communication.

The second set of iterations is made up from (3) the detection of conflicting forces, (4) a pattern-driven object-oriented design, (5) the implementation of this design using object-oriented development technologies like frameworks or low-level components, and (6) functional tests. We call these iterations development iterations since they form the part of the process, where software engineers develop the application.

The user first identifies the conflicting forces. Developers will assist the user in this task, by structuring the discussion. Together with the user, the developer then seeks for low-level groupware patterns, which resolve the identified forces. Developers implement the pattern by means of application frameworks or developer-centred component frameworks. This normally involves the development of new software components. The components can be built using groupware frameworks or other base technologies. To ease the implementation, each groupware-pattern can have technology recipes that show how the pattern is implemented with a specific technology (using the cookbook style that was described in the *Assembly Cookbook* pattern [5]).

The result is tested using as much automated tests as possible (note that phases 5 and 6 are executed in reverse order, if the eXtreme Programming process is combined with our development process). Phases (3) to (6) require a groupware-expert and software developer to be involved. The user still makes an important contribution to the design because he participates in steps (3) and (4).

When the development of the groupware system is complete, the end-user starts using it for the desired purpose. While using the system, end-users with pattern-based groupware design knowledge are encouraged to (7) reflect on their activities. This *reflection in action* [8] reveals actions that complicate or hinder the work process. High-level groupware patterns (8) help in this process by describing frequently occurring issues, the various forces and a proven solution in a way that is appropriate for tailoring end users. Based on high-level patterns the user can now select (9) and combine (10) the groupware components that suit his needs.

It is important to note that the user will perform tailoring operations on his own. Typically, there will be no software developer available to assist him. So, the task of tailoring must be appropriate for the end user.

Groupware templates

Macro-level groupware patterns describe even more than just the need for the integration of a single component. They act as templates, and describe aspects such as a set of people, the tools they use to collaborate (communication means and shared information objects), possibly also coordination policies that apply during an online meeting and the roles of specific participants. Some of these aspects may not be fully specified in the pattern: if e.g., the pattern does not specify which people should be invited for a sales meeting, the groupware system can query the user for this information when the pattern is applied.

The high-level patterns and the macro-level patterns have to be written in a form that suitable for an end-user, who acts as tailor. The tailoring environment should supports him in the process of (10) composing the identified components. The collaborative nature of groupware provides many triggers for tailoring operations [7] (e.g., by observing each other's tailored systems) and offers opportunities for sharing tailored artefacts [14].

CONCLUSION

The Oregon Groupware Development Process presented in this paper is a design methodology for groupware applications that heavily involves end-users and fosters reuse of existing knowledge from the area of groupware research. We believe such a process is needed to empower people to flexibly employ groupware applications.

Examples of groupware patterns as mentioned in this paper can be found in the groupware-pattern catalogue [9]). We are in the process of validating our approach in real-life collaborative settings.

REFERENCES

1. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, S.: A pattern language. New York: Oxford University Press, 1977.
2. Alexander, C., Silverstein, M., Angel, S., Ishikawa, S. and Abrams, D.: The Oregon Experiment. New York: Oxford University Press, 1980.
3. Alexander, C.: The timeless way of building. New York: Oxford University Press, 1979.
4. Erickson, T.: Lingua Francas for Design: Sacred Places and Pattern Languages. *Proceedings of Designing Interactive Systems (DIS 2000)*, ACM Press: Brooklyn, NY,2000.
5. Eskelin, P.: Assembly Cookbook Pattern. http://c2.com/cgi/wiki?AssemblyCookbook: 1999.
6. Kahler, H., Mørch, A., Stiemerling, O. and Wulf, V.: Tailorable Systems and Cooperative Work (introduction). In *Special Issue of Computer Supported Cooperative Work*, 9 (1), 2000.
7. Mackay, W. E.: Triggers and barriers to customizing software. *Proceedings of ACM CHI '91 Human Factors in Computing Systems*, ACM/SIGCHI: New Orleans, Louisianna,1991.
8. Schön, D.: The Reflective Practitioner. How Professionals Think in Action.. New York: Basic Books, 1983.
9. Schuemmer, T., Fernandez, A. and Holmer, T. (Ed.): *The Catalog of Groupware-Patterns*, Available online at: http://www.groupware-patterns.org 2002.
10. Slagter, R., Biemans, M. and Ter Hofte, G. H.: Evolution in Use of Groupware: Facilitating Tailoring to the Extreme. *Proceedings of the CRIWG Seventh international Workshop on Groupware*, IEEE Computer Society Press: Darmstadt,2001, 68-73.
11. Snider, J. R.: User Participation and the Oregon Experiment as Impolemented with the Esslinger Hall Recreation and Fitness Center. http://darkwing.uoregon.edu/~jsnider/esslinger.html: 1999.
12. Stiemerling, O.: Component-based tailorability. Bonn, Germany: University of Bonn, 2000.
13. Teege, G.: Users as Composers: Parts and Features as a Basis for Tailorability in CSCW Systems. In *Computer Supported Cooperative Work (CSCW)*, 9 2000, pp. 101-122.
14. Wulf, V.: 'Let's see your Search-Tool!' - On the Collaborative use of Tailored Artifacts. *Proceedings of GROUP '99*, ACM-Press: New York,1999, 50-60.

Position paper for CHI workshop on EUD

Design Principles and Claims for End-User Development

Alistair Sutcliffe

Centre for HCI Design
Department of Computation, UMIST
PO Box 88, Manchester M60 1QD, UK
a.g.sutcliffe@co.umist.ac.uk

1. Introduction

EUD essentially out-sources development effort to the end user. Hence it imposes cost of additional design time and learning EUD tools. These costs are critical because end users are busy people for whom programming is not their primary task. They only tolerate development activity as a means towards the end that they wish to achieve; for instance, creating a simulation, experimenting with a design, building a prototype. User motivation is derived from perceived benefits and actual rewards from creating working systems. The key to success or failure EUD, I will argue in this paper, is to maintain a positive balance between user motivation and cost. Design principles should focus design toward that end. However, design principles can only provide high-level guidance that has to be interpreted in a specific design context. Consequently I will elaborate EUD principles as asset of claims (Carroll, 2002).

2. Principles for EUD

The aim for all design is to achieve an optimal fit between the product and the requirements of the customer population, with minimal cost (see Sutcliffe 2002 for more details). Generally, the better the fit between users' needs and application functionality, the greater the users' satisfaction; however, product fit is influenced by the application scope i.e. the generality/specialisation of the domain. This can be summarised in the principle of user satisfaction:

- *The user satisfaction supplied by an EUD environment will be inversely proportional to domain scope and variability in the user population.*

The consequences of this law are that more general EUD systems either have to have more motivated users, or motivate their users more. Furthermore a heterogeneous user population will be more difficult to satisfy, because getting the right fit for each sub-group of individuals becomes progressively more challenging and expensive. The second consequence is that general applications tend to be more complex; and people have a larger learning burden with complex products. General products may not motivate us to expend development effort because the utility they deliver is less than a perceived reward from satisfying our specific requirements:

- *The effort a user will devote to customising and learning software will be proportional to the perceived utility of a product in achieving a job of work or entertainment.*

User motivation will depend critically on perceived utility and then the actual utility payoff. For work-related applications we are likely to spend time customising and developing software only if we are confident that it will empower our work, save time on the job and raise productivity. Development effort can range from customisation of products by setting parameters, style sheets and user profiles, to designing customised reports and user interfaces with tools, to full development of functionality by programming or design by configuration of reusable component. Adaptable products provide users with these facilities but at the penalty of increasing effort. In contrast, adaptive products take the initiative to save users effort, but the downside is when the adaptation creates errors.

Adaptation is fine so long as it is accurate, but when the machine makes mistakes and adapts in the wrong direction it inflicts a double penalty because incorrect adaptation is perceived as a fault. This lowers our motivation to use it effectively, and leads to a third principle:

- *The acceptability of adaptation is inversely proportional to system errors in adaptation.*

Inappropriate adaptation inflicts a triple penalty on our motivation from the cost of diagnosing mistakes, cost of working around the problem, and negative emotions about systems usurping human roles). Design of EUD therefore has to concentrate on cost minimisation either by appropriate automation in adaptive systems, or by leaving the initiative with the user and thereby imposing more costs. Costs and rewards are going to be influenced by the type of user and their domain that can be expressed as a scenario. Design issues concern the modality and intuitiveness of communication, expressive power of the EUD language, and system initiative. In the following section I describe these issues as claims for EUD.

3. Claims for EUD.

Claims are a form of extended, psychologically motivated design rationale that express a design principle, with usability and utility trade offs, set in a context of use by a scenario, with a specific design that exemplifies the principle. The first claim focuses on a specific domain for EUD:

Claim ID: Domain specific EUD Language for Protein Biology.
Claim: A formal sub language is created with domain specific lexicon and syntax that allows users to create executable procedures and queries.
Upsides: The formal sub language is already part of the user's domain knowledge so it is easy to learn. The formal sub language restricts interpretation errors.
Downsides: The sub language will not be extensible to other domains. The sub language may be difficult to change and evolve.
Scenario: Simon is a molecular biologist who works with Proteases. He needs to simulate the structure and behaviour of new peptide structures. He composes a new peptide by picking amino acids from a palette, and then parameterises the simulation by identifying target proteins for the protease to react with a range of Ph and temperatures. The system creates a visualisation of the protease as it reacts with the protein over time according to the range of Ph and temperature conditions.
Example: Comp <NewProtease>::= Lys, Asp, Glut, …
React <NewProtease> + AclCoH, EndoPh, Serot With Ph<4.5 -> 6.7 inc 0.1>; C 10 → 20 inc 1.0
VisStruct Tint = 5 secs

Note that assumptions about the user's domain knowledge in the scenario are necessary for interpretation of the claim and its trade offs. A closely related claim describes a direct manipulation design in which the user picks amino acids from a palette and set the simulation parameters by a set of sliders and buttons. This GUI style design would have trade offs of being easier to learn for users not familiar with the sub language but the downside of less flexibility since the parameters and amino acids in the palette would have to be fixed by the design. Of course another EUD interface could be added to programme the interface, and that would be expressed in another claim. The second claim describes a more general application:

Claim ID: Robot Programme by example.

Claim: Instructions for programming a robot are given with a graphical simulation system that detect movements of the robot and automatically generates instruction for programming a real robot.
Upsides: Programming is easy for users who just have to demonstrate the behaviour they want.
Downsides: Complex instructions are difficult to express by demonstration; the conceptual model of instruction may not be clear to users.
Instructions can be misinterpreted if the demonstration is not clear.
Scenario: Katie wants to programme her robot for the computer football competition. She starts the robot instruction environment, which shows her the arena divided into squares, with icons for her robot and opposing team's robot. She has to give her robot instructions to manoeuvre past the opposition and score a goal. She sets the recognise button on and moves her robot so it collides with the opposition. Then she sets the react button on and moves her robot back one square, then to the left and then forward two squares. Katie then selects the test button and moves her robot to collide with the other one and observes the response generated by the system. When she is happy with the result she presses the download button and connects the physical robot she has designed.
Example: Legoland robot control system.

In this claim a semi automated programming by example is proposed. The downside draws attention to the limitations of this approach, for instance longer sequences become progressively more difficult to interpret, and Katie might have had problems in grasping the conceptual model so that it is better to give short declarative instructions rather than long procedural sequences. A related claim <Robot instruction sub language> that proposes a specific language for instructing the robot (i.e. If collide Then Move N/W/S/E; x spaces) is related to the first domain specific language claim but may have different upsides depending on the scenario context. For instance it may be desirable to make the language explicit so children learn control abstractions.

The third claim addresses a more general problem.

Claim ID: General Purpose EUD simulations.
Claim: The EUD environment combines editors for building graphical simulations with form-filling dialogues for specifying rules that control agents' behaviour.
Upsides: EUD environment is very powerful because it can be used for a wide range of applications.
Combination of the graphic simulation and declarative rule-based instruction is easy to learn.
Downsides: Solutions to complex applications may not be easy.
Building graphical simulations are time consuming.
Rules can have complex and unpredictable interactions in complex systems.
Scenario: Alex has been challenged to build a programme that can simulate a soccer game. He starts the AgentSheets system and builds icons for players and the graphics for the football pitch. Then he starts to enter rules for the player's behaviour when they have the ball and hear an opposing player, near the goal, etc. He runs the simulation but the player's behaviour doesn't look right, furthermore, he has a large number of rules for each type of player and the number of unexpected interactions is increasing. Then he has a brainwave and creates a data structure that maps the football field into zones of attractive or repulsive force. This simplifies his rules since his agents just have to maximise their attraction towards the opposite goal and repel opposing players. He re-runs the simulation and the player's behaviour follows real life patterns.
Example: AgentSheets EUD environment.

This claim is partially related to Robot Programming sub Language but it deals with the merits of a more general EUD environment. The scenario plays a dual role, first it provides a context for interpreting the general EUD environment claim, and secondly, problems described within motivate development of new claims. For instance the problem with interacting rules and solutions using a data structure representing force attraction could suggest a claim about creating a reuse library to share conceptual models for solving difficult problems. Furthermore, the long time taken to create the graphics, expressed in one of the downsides, could indicate a similar claim for sharable graphics libraries.

4. Conclusions

Claims provide a powerful way of representing design knowledge for End User Development because scenarios provide a rich context for interpreting the advantages and disadvantages of a particular design approach. Setting claims with a perspective of cost-benefit analysis should also help to check the reality of different design approaches and assess the competitive advantage of EUD over other development paradigms such as customisable COTS software. However, scenarios also beg the question of completeness. Creating a necessary and sufficient set of scenarios to brainstorm all the possibilities for EUD would be a marathon task, and is unlike to be completeable. On a more optimistic note, taking a claims oriented approach to investigating the EUD problem space does allow design to be developed form concrete description of need (in scenarios) and evolving a network of related claims could lay the foundation for a library of EUD design knowledge.

References

Carroll, J. M. (2000). *Making use: Scenario-based design of human-computer interactions*. Cambridge MA: MIT Press.
Sutcliffe, A. G. (2002). *The Domain Theory: Patterns for knowledge and software reuse*. Mahwah NJ: Lawrence Erlbaum Associates.

The Pragmatic Web: Customizable Web Applications

Alexander Repenning
Department of Computer Science
Center for LifeLong Learning & Design
University of Colorado, Boulder CO 80309-0430
ralex@cs.colorado.edu

INTRODUCTION

Information is no longer a scarce resource. However, this achievement is largely useless if information is not provided in a format tailored to the user. Most people in the United States now have access to online information thanks to inexpensive computers and information appliances [1], and public access to computing resources in libraries and other institutions. Web-based information can be accessed – through wires or wirelessly – from desktop computers, laptops, PDAs, cell phones, and specialized information appliances. However, *ubiquitous information access does not imply universal information access*. The representational formats chosen by information producers often do not match the needs of information consumers. For instance, a Web page may contain highly relevant information to a user, but this information cannot be accessed if the user cannot see or cannot read. Reasons for producer/consumer *information representation mismatch* include:

- **Wrong Modality**: Blind users cannot read textual descriptions. Automatic text-to-speech interfaces may be able to verbally convey the textual contents of a Web page to users, but if the Web page is formatted for visual access, the sequential presentation of information as speech may be unintelligible or inefficient.
- **Wrong Language**: Crucial explanatory text may be provided in the wrong language. Less than 15% of U.S. Web sites contain Spanish translations [2].
- **Wrong Nomenclature**: Information may be expressed in an unfamiliar measurement system. The translation of Celsius to Fahrenheit or kilometers to miles, while scientifically trivial, may present a serious impediment to many users.
- **Wrong Time**: Information may be correct, relevant and readable, but presented at the wrong time. Stock information, for instance, is most useful when presented in real time.
- **Wrong Format**: Information can look great on a large computer monitor, but be completely unsuitable for small information devices such as PDAs and cell phones.

A mismatch between information presentation and the formats required by an information consumer can be difficult to address with a traditional Web browser. For economic reasons, a producer may choose to use a single representation scheme that addresses only the needs of an anticipated majority of information consumers. Because creating and maintaining multilingual Web sites can be costly, information consumers who are not proficient English readers have few options with this model.

In the case of a person with a cognitive disability planning to use the public transportation system, there is a good chance that essential information exists on the Web but cannot be accessed meaningfully with conventional information technology. The goal of this proposal is to extend control to information consumers and let them use information in fundamentally new ways that might not be anticipated by information producers.

The ultimate questions involve *who* controls information representation and *how* the information is processed. The Control over Representation diagram (Figure 1) illustrates a continuum of control that ranges between two extreme positions. From left to right, it identifies conceptual as well as technical frameworks in order of increasing information consumer control: the Syntactic Web, the Semantic Web and the Pragmatic Web.

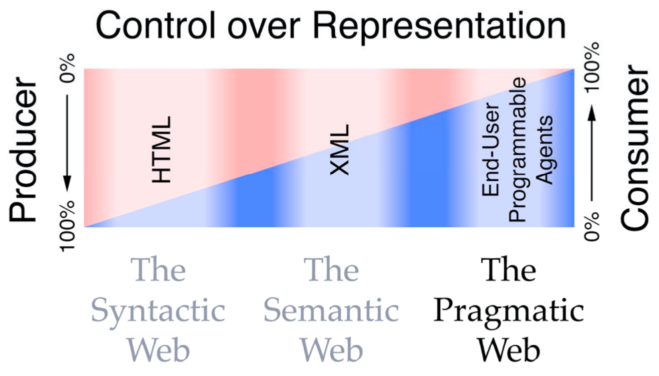


Figure 1: Distribution of Control over Representation shared by Information Producer and Information Consumer.

The Syntactic Web. In this first generation of Web technology, a simple markup language (HTML) is used to define content at a high level of detail at a syntactic level that controls the appearance of information. Information producers define content, font selection, layout, and colors. Information consumers have limited control over representations in their browser, including adjusting the size of fonts, and enabling/disabling animations and plug-ins.

The Semantic Web. According to Tim Berners-Lee, the Semantic Web [3-5] will "radically change the nature of the Web." [4] The formal nature of representation languages such as the eXtensible Markup Language (XML) and the Resource Description Framework (RDF) make Web-based information readable not only to humans, but also to computers. For instance, semantic-enabled search agents will be able to collect machine-readable data from diverse sources, process it and infer new facts. Unfortunately, the full benefits of the Semantic Web may be years away and will be reached only when a critical mass of semantic information is available. Critics of the Semantic Web [6] point out the enormous undertaking of creating the necessary standardized information ontologies to make information universally processable.

The Pragmatic Web. In contrast to the Syntactic and Semantic Web the Pragmatic Web is not about form or meaning of information but about **how information is used**.

The Pragmatic Web's mission is to provide information consumers with computational *agents* to transform *existing information* into *relevant information of practical consequences*. This transformation may be as simple as extracting a number out of a table from a single Web page or may be as complex as intelligently fusing the information from many different Web pages into new aggregated representations.

This agent-based transformation needs to be extremely flexible to deal with a variety of contexts and user requirements. An agent running on a desktop computer with a large display may utilize rich graphical representation versus an agent running on a cell phone with a small display may have to resort to synthesized text information to convey the same information.

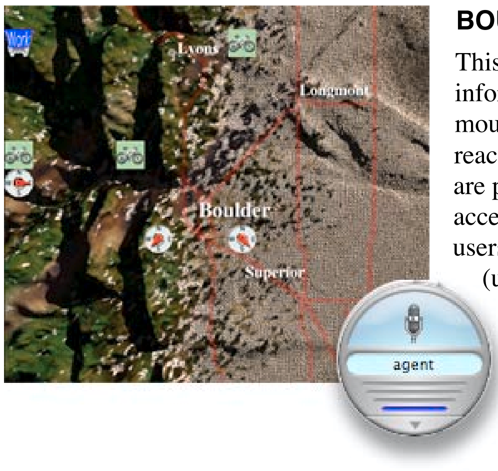
The Pragmatic Web research explores the practice of using information and the design of tools supporting this process.

Instead of the traditional "click the link" browser-based interfaces, agents capable of multimodal communication will provide access to Web-based information. Agent communication methods include facial animation, speech synthesis, and speech recognition and understanding. End-users or caretakers will instruct agents to transform information in highly customized ways. Agents will work together to combine information from multiple web pages, access information autonomously or triggered by voice commands, and represent synthesized information through multimodal channels.

End-User Customization [7, 8] will be an integral part of the Pragmatic Web. Successful development of end-user customization will make giant steps toward an Every Citizen Interface (ECI) [9] by letting minority groups of information consumers, possibly down to the single individual level, obtain ways to control information representations in accordance with their specific needs. This computer-supported *Information Processing* is a form of knowledge management [10, 11] that turns raw data into information. End-user customization will let users specify *where* information is accessed (e.g. part of an existing Web page), *how* it is accessed (e.g., voice activated), and how information is further *processed.* For instance, a Pragmatic Web application could run on a wireless PDA equipped with GPS to help a person with a cognitive disability navigate through town using the local public transportation system.

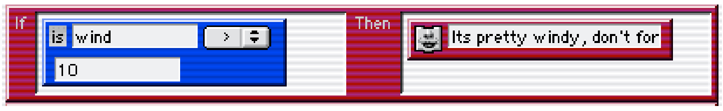
The Pragmatic Web does not intend to subsume the Syntactic Web or the Semantic Web. On the contrary, the Pragmatic Web will initially work with the Syntactic Web by letting end-user customizable agents extract information out of existing (HTML) Web pages [12]. When the Semantic Web reaches a minimal critical mass, the Pragmatic Web will then utilize the Semantic Web with agents that access ontologies and make inferences based on these ontologies.

.



BOULDER MOUNTAIN BIKE ADVISOR

This is an AgentSheets-based application that connects real-time Web information with speech recognition. A user asks “Where should I go mountain biking.” Several agents located on a map of Boulder County react to this voice command. These agents are representing locations that are possible candidates for biking and also feature real time, Web accessible weather information sensors. Rules previously defined by the users capture pragmatic interpretations. For instance, an agent may reply (using speech output): “It’s really nice up here at Betasso but you should bring a jacket because it’s a little windy”



BOULDER LIVE

Navigating through a city using public transportation can be a challenge. An effort sponsored by the Coleman foundation is using Added Dimension 3D to help persons with cognitive disabilities with navigation tasks. 27 Global Positioning System equipped busses in Boulder Colorado are tracked wirelessly by agents. A 3D visualization allows observers to see the current location of busses and GPS equipped bus users. Observers can watch in real time, play recorded data and assume different camera positions (e.g., birds eye, bus stop perspective, bus driver perspective). Bus users can locate relevant busses based on their current position and identification information. End-user development allows users, or their care providers, to specify rules that turn the general bus information space into personally relevant, pragmatic information communicated through cell phones.

bus
departing
bus

[1] A. Fox, B. Johanson, P. Hanrahan, and T. Winograd, "Integrating Information Appliances into an Interactive Workspace," *IEEE Computer Graphics and Applications*, vol. 30, pp. 54-65, 2000.

[2] E. Lamb, "Web content struggles to go worldwide," in *Red Herring*, vol. 91, 2001, pp. 38-39.

[3] W3C, "W3C Semantic Web Activity Statement," 2001.

[4] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, 2001.

[5] T. Berners-Lee, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. San Francisco, CA: Harper, 1999.

[6] M. Frauenfelder, "A Smarter Web," *Technology Review*, 2001.

[7] B. Nardi, *A Small Matter of Programming*. Cambridge, MA: MIT Press, 1993.

[8] C. Jones, "End-user programming," *IEEE Computer*, vol. 28, pp. 68-70, 1995.

[9] T. a. E.-C. I. t. t. N. s. I. I. S. Committee, C. S. a. T. Board, M. Commission on Physical Sciences, and Applications, and N. R. Council, *More Than Screen Deep: Toward Every-Citizen Interfaces to the Nation's Information Infrastructure*. Washington, D.C.: National Academy Press, 1997.

[10] M. Sumner, "Knowledge management: theory and practice," presented at Proceedings of the 1999 ACM SIGCPR conference on Computer personnel research, New Orleans, LA USA, 1999.

[11] A. J. Murray, "Knowledge management and consciousness," *Advances in Mind–Body Medicine*, vol. 16, pp. 233-237, 1999.

[12] W. H. Inmon, "The Data Warehouse and Data Mining," *Communications of the ACM*, vol. 39, pp. 49-50, 1996.

Supporting End-User Development of Component-Based Applications by Checking Semantic Integrity

Markus Won
Institute for Computer Science III, University of Bonn
Roemerstrasse 164, 53117 Bonn, Germany
+49 228 73 4506,
won@cs.uni-bonn.de

ABSTRACT

Component-based software can be used to build highly tailorable and therefore flexible software systems. To support end-users when tailoring or even developing their applications themselves different approaches were discussed. This papers describes an interactive integrity check as a support for end-user development or tailoring. It is based on the idea that developers can describe the “right” use of their components as well as they can describe properties which belong to specific groups of applications. Those information can be used to check the application composed by such components at tailoring time. Thus, the learning of tailoring activities will improve as well as a better understanding for the resulting software can be achieved.

Keywords
End-User Development, Tailoring, Component-Based Systems, Integrity Checking

INTRODUCTION

Most of the software sold nowadays are off-the-shelf products designed to meet the requirements of very different types of users. One way to meet these requirements is to design software that is flexible in such a way that it can be used in very different contexts. This flexibility can be achieved by tailorable design architectures. The idea behind this concept is that every user can tailor his software in the way that it meets his personal working contexts best or build an application “from the scratch” using existing components.

Component-based architectures – basically developed with the idea of higher reusability of parts of software – can be used to build those highly flexible software. The same operations that are used by developers (i.e. choosing components, parameterizing them, binding them together) then can be applied to the context of end-user development. The main difference is the granularity and semantics of the single components.

Especially if the components have a GUI representation it is quite easy to design a tailoring language including a visual representation. Although both – the idea of adapting an application seen as a composition of components and the use of visual tailoring language which supports only very few tailoring mechanisms – are well understood by users, there is a need for further support during tailoring time.

This paper describes a new approach which uses an integrity check not only to ensure correctness of an application but support users interactively during development time (as a guidance). So, an integrity checking mechanism not only has to control the validity of the composition but shows the source of error and also can give hints or corrects the composition itself. The idea behind that is not only to assist the end-user’s construction activities but also to stimulate the learning of the tailoring language, the proper use of the components and of the functionality of the resulting application (by looking behind the scenes).

In the following section the context of this work is described in more detail focusing on tailorable component-based software. After that we will concentrate on the idea of integrity and integrity checking which are both common in computer science. Here a short overview on the state of the art is given. The following sections then focus on the idea of integrity checking as a support for end-user tailoring and shows the current state of our prototype.

COMPONENT-BASED TAILORABILITY

In the last years component-based architectures [10] have become quite fashionable in the field of software engineering. A very important property of a component is its reusability and independent development of components. Thus, components can be seen as small programs which can exist and run alone but may also be combined with other components. An application normally consists of several components that are connected with each other. Applications based on component architectures are designed by selecting one or more components and connections between them. Furthermore, applications can be easily enhanced by adding new components to the existing set. Mainly there are three different operations which are used to design applications with component-based architectures.

- adding or removing components,
- changing the parameters of one component or
- linking two components according to their specified interfaces

They can be used during construction time as well as in case of tailoring an existing application. [9]. The end-user tailoring language then consists of those three simple operations. The operands within the tailoring language are the components. According to ease the learning of such a tailoring language it is due to the designer of an component set to choose the right granularity and a "natural" description of the components' functionality. Thus, beginners start to compose their own applications by using few components which provide for a great amount of functionality, whereas more experienced users can combine more components which are smaller. A second step to ease the learning therefore is to allow for a layered architectures [12] which means that several components can be stick together and saved as one larger component (which we call abstract component).

This idea was implemented with the FREEVOLVE platform [9] which is based on the FLEXIBEAN [9] component model. Several visual tailoring environments were developed and evaluated. Most of the users understood the concept of component-based architectures easily as there are several domains in real life which are very similar to that concept. One of the remaining problems is that it is not clear which component to choose or how the components have to be bound together. Seen from the perspective of the less-experienced developers the semantics of the components as well as their parameters and interfaces have to become more transparent. In the following several concepts that ease the learning of a tailoring environment as well as their integration into a search tool based on the ideas of the FREEVOLVE platform are described shortly.

First, Mackay [4] found that the lack of documentation of respective functions is a barrier to tailoring. Manuals and help texts are typical means to describe the functionality of applications. Thus, all simple components within FREEVOLVE were extended by *descriptions*. Furthermore, help texts can be added to abstract components by the composers. Such an abstract component and the belonging description then can be annotated by other users. So, discussions could support the deeper understanding of an tailoring artifact.

Mackay [4] and Oppermann and Simm [6] found that *experimentation* plays a major role in learning tailoring functions. "Undo function", "experimental data", "neutral mode", etc. are features which support users in carrying out experiments with a system's function. Those functions were integrated into the FREEVOLVE platform. Furthermore, an exploration mode was added which simulates a work space with experimental data. Users who want to explore the changed functionality of a tailored application can do so by looking at the impacts on the virtual work space when using the application.

Examples provided by other users are an important trigger to tailor [15]. While the FREEVOLVE tailoring environment supports experiments, the question how to support the exploration of compound components or elementary components remains. These artifacts cannot be executed in the environment by themselves. A solution is to exemplify the use of a component by a small characteristic example application.

Finally, automatically generated and visualized *integrity checks* can prevent from building pointless application (cf. [13]). Here integrity checks are used not only to prevent from failures but in the way of supporting users when developing their applications in the way that making failures and getting corrections on that can be seen as learning.[1]

Thus in the following we will concentrate on how integrity checks can be used to ease the learning of a component-based tailoring language.

STATE OF THE ART – INTEGRITY CONTROL

Integrity checking is widely common within different fields of computer science. In the following we will list the techniques (cf. Figure 1) which are used in our concept (see next section).

Database integrity: The domain of information systems is concerned with the consistent structuring and storing of information. During design time scheme transformations can help to create a appropriate data models [8] which prevents from inconsistent data. During runtime information systems maintain the consistent data basis by controlling new input or changes. That can be done by constraints or triggers [8].

Software engineering (design by contract): The design by contract-concept [5] concentrates on the idea that additional conditions can be added to methods when they are to be invoked. So, a pre-condition can be formulated which has to be fulfilled by the invoking object. On the other hand after the method has ended a post-condition is guaranteed. For Java there are special tools (i.e. [3]) which deal with this approach focusing on better error removal and therefore faster development

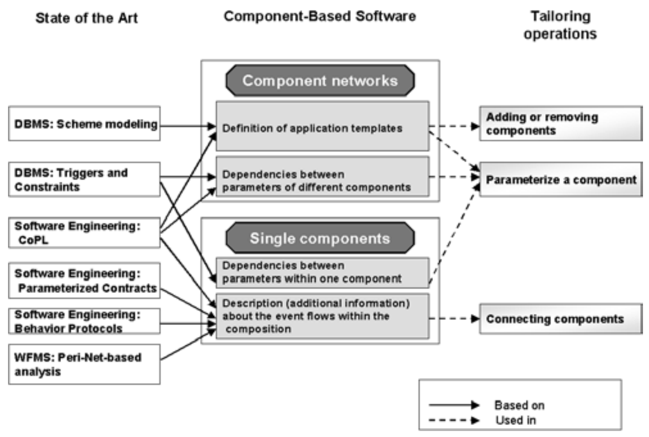

State of the Art
Component-Based Software
Tailoring operations
Component networks
DBMS: Scheme modeling
Definition of application templates
Adding or removing components
DBMS: Triggers and Constraints
Dependencies between parameters of different components
Parameterize a component
Software Engineering: CoPL
Single components
Software Engineering: Parameterized Contracts
Dependencies between parameters within one component
Software Engineering: Behavior Protocols
Description (additional information) about the event flows within the composition
Connecting components
WFMS: Peri-Net-based analysis
Based on
Used in

Figure 1: Integrity strategies (Overview)

Additional semantic interface descriptions: Behavior Protocols [7] add extending information to the interface definition by describing how a component can be used (which methods can be called in which order). If the component is used, the caller has to comply with this protocol.

[1] This technique often applied by developers who use the error message of the compiler to make corrections or improvements. The success depends to a great extent on the quality of the developing tools.

Application templates including integrity constraints: Birngruber and Hof [1] describe a group of applications by a plan. This plan consists of conditions on the use of special components, parameters which are dependent on others, etc. The idea here that the so called CoPL Generator analyses the plan and builds an application according to those conditions. The composition is done semi-automatically as there is user interaction where decisions can be made.

CONSTRAINT- AND EVENT-FLOW-BASED INTEGRITY

In the following we will describe our approach of extending the FREEVOLVE platform by an integrity check. This is done in two steps: First integrity strategies and a condition-describing language have to be designed. Those different integrity strategies are all based on the assumption that special meta information on the components and about domains or groups of applications is given or can be added to the component set. After that there an integrity checking mechanism which interacts with the tailoring end-users marks errors, gives hints etc. (see above) has to be integrated. The design of the interaction between the integrity check and the users should be done with the focus on easy understanding and helping to learn how to tailor by assemble components.

The integrity checks implemented in FREEVOLVE are based on two main techniques: Constraints and Event Flow Integrity.

Constraints – as they are used in database systems – monitor a system according to specific conditions. If a condition is not fulfilled anymore, the system outputs an error message. Some systems do further actions to correct the system's state according to the violated condition. In our system the components may have additional conditions (i.e. size, color etc.) which have to be controlled. If there are dependencies between different component they have to be described externally. But there can be mutual dependencies between different parameters within one component, too. They can be checked within the component's functionality. But an explicit description of those dependencies not only ensures the correct parameterization but also explains parts of the functionality, thus helps learning and understanding the component's semantics. External dependencies between parameters of different components have to be described explicitly. Furthermore, explicit information to components enhance the flexibility, especially if integrity conditions are bound to a special field of application or domain. Integrity conditions here an be compared to constraints in DBMS. According to this comparison automatic corrections can be seen as triggers where the action is the adjustment itself.

The Event Flow Integrity (EFI) [12] controls the data flow between components. A simple example illustrates the idea of this technique. In Figure 2 there are three components building a very simple search tool. A start button triggers a search engine which outputs the search result to another component (switch). To ensure that there are correct connections between these three components one could declare two constraints. In this special case we would have "the search engine has to be triggered (by a button)" and "the search result has to be passed to another component". The problem is that the switch (third component) is no real output component but only splits the search result. What we want to ensure is the data flow. Furthermore, we want to ensure that the search result is passed to an output component, or else: the produced search result has to be consumed.

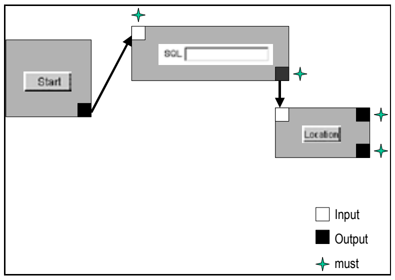


Figure 2: Event Flow Integrity

EFI now uses special information which belong to every component that describe how the ports are used and how the component behaves according to a data stream (i.e. producer, consumer, transmitter). Using those information EFI checks if every produced event or data (that has to be consumed) will be consumed. The algorithms used here are similar to the one which are used to analyze workflows [2] and base on Petri Nets.

Additionally, we have integrated so called application templates. To ease the end-user development we offer those templates for different kinds of application types. Here constraints not only for one component but for a set of components (composition, the application itself) are given. So, given the search tool example, a template "Search Tool" would ensure that there is a search engine and at least one output component (a arbitrary component which consumes search results). Furthermore the output port of the search engine (result) should be set to "essentially needed" to ensure that EFI checks whether there is a connection to at least one output component.

Both integrity strategies have to be integrated into the incremental process of developing an application. So in the following we will first describe how these ideas fit into the tailoring language. After that, we will describe how an tailoring or development GUI has to be designed to present those integrity information to the users.

SUPPORTING END-USER DEVELOPMENT BY INTEGRITY CHECKS

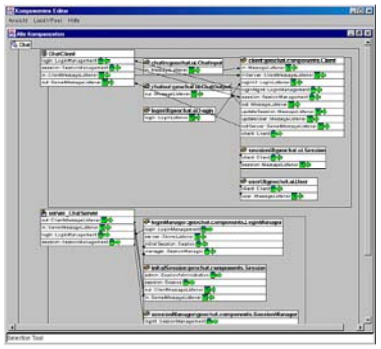
Both strategies have to be mapped to the tailoring operations which component architectures provide for. This is essential especially if the integrity checking should be done during the tailoring and so support the tailoring or developing act itself. Thus, the integrity constraints and conditions have to be associated with the three tailoring operations, that are:

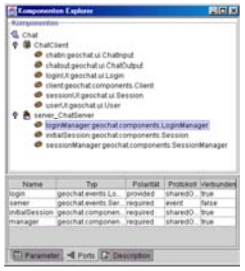
- **Parameterization of components**: Can be done by constraints. In some cases automatic corrections of parameters can be useful. Dependencies should be presented to the users. Eventually, hints can be given how corrections can be done.

- **Changing the connections**: Every time a connection is changed EFI can be check interactively. Using this mechanism interactive support can be given to the users during development time.
- **Adding/removing components**: Global constraints described in application templates (.i.e. for a the "class" of movie player applications) ensure the existence of some components. Furthermore here are described dependencies between component's existence (if component a is part of application X then there must be a component b or a component c). Additionally every time a component is added or removed EFI has to check if there are needed bindings.

INTEGRITY CHECKING TAILORING ENVIRONMENT

Basically, the FREEVOLVE system provides for an powerful API that allows easy integration of different visual tailoring environments. A new developed one supports different views at the composition (.i.e. WYSIWIG, components and their connections, tree view on components, etc.). All views are synchronized with each other. Currently, is designed to support experienced users or administrators (but not especially programmers) when tailoring. Figure 3 shows screen shots of the first prototype of the tailoring environment.





Chat
ChatClient
chatIn:geochat.ui.ChatInput
chatout:geochat.ui.ChatOutput
loginUI:geochat.ui.Login
client:geochat.components.Client
sessionUI:geochat.ui.Session
userUI:geochat.ui.User
server_ChatServer
loginManager:geochat.components.LoginManager
initialSession:geochat.components.Session
sessionManager:geochat.components.SessionManager

Name	Typ	Polarität	Protokoll	Verbunden
login	geochat.events.Lo...	provided	sharedO...	true
server	geochat.events.Ser...	required	event	false
initialSession	geochat.componen...	required	sharedO...	true
manager	geochat.componen...	required	sharedO...	true

Figure 3: Tailoring Environment

The tailoring environment itself provides for an integrity visualization API. It allows to highlight components or pop up messages. The current visualization of hints or messages of the integrity checker is based on the idea that the integrity check should mediate between the user's tailoring action and the tailoring environment. Thus every tailoring action and the current composition have to be checked. Messages are displayed by marking the component or connection which causes an error or displaying help texts (which should be clearly assigned to the defective part of the composition).

In some cases system generated recommendations can be shown how to improve the composition. If there is no direct way to do that examples can be generated which illustrate the "right" use of the components that are contained in the defective part of the composition.

First tests with users have been done so far.

CONCLUSIONS

This paper describes a new way how the learning and using of tailoring or end-user-oriented development languages can be eased. Here, this is done by adding an integrity checking mechanism which helps to interactively improving the developed application. This approach is still ongoing work.

REFERENCES

1. Birngruber, D.: "A Software Composition Language and Its Implementation", in *Perspectives of System Informatics (PSI 2001)*, vol. LNCS 2244, D. B. Bjorner, M.; Zamulin, A. V., Ed. Novosibirsk, Russland: Springer, 2001, pp. 519-529.
2. Blom, M.: "Semantic Integrity in Program Development", in *Department of Computer Science*: Karlstad University, 1997.
3. *Griffiths, A.*: "Introducing JUnit", 2001, http://www.octopull.demon.co.uk/java/Introducing_JUnit.html.
4. Mackay, W. E.: "Users and customizable Software: A Co-Adaptive Phenomenon", Boston (MA): MIT, 1990.
5. Meyer, B.: "Eifel: A Language and Environment for Software Engeneering", *Journal of Systems and Software*, 1988.
6. Oppermann, R. and Simm, H.: "Adaptability: User-Initiated Individualization", in *Adaptive User Support - Ergonomic Design of Manually and Automatically Adaptable Software*, R. Oppermann, Ed. Hillsdale, New Jersey: Lawrence Erlbaum Ass, 1994.
7. Plásil, F. V., S.; Besta, M.: "Behavior Protocols", Dep. of SW Engineering, Charles University, Prague, Technical Report, No: 2000/7, August 2000
8. Silberschatz, A., Korth, H., and Sudarshan, S.: *Database System Concepts*, Osborne McGraw-Hill, 2001.
9. Stiemerling, O.: "The Evolve Project.", in *Institute for Computer Science III*, University of Bonn, 2000.
10. Szyperski, C.: *Component Software - Beyond object-orientated programming*. New York: ACM Press, 1998.
11. Won, M.: "Komponentenbasierte Anpassbarkeit - Anwendung auf ein Suchtool für Groupware", in *Institute for Computer Science III*, University of Bonn, 1998.
12. Won, Markus, Cremers, Armin B.: "Supporting End-User Tailoring of Component-Based Software - Checking Integrity of Composition", in: Proceedings of Colognet 2002 (Conjuction with LOPSTR 2002), Madrid, Spain, 19.-20.09.2002
13. Wulf, V.: "Let's see your Search-Tool! - Collaborative use of Tailored Artifacts in Groupware", in: Proceedings of GROUP '99, ACM-Press, pp. 50-60, 1999