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

D3.4 Proceedings Session on End-User Development held at HCI International 2003 Conference

Summary

This document contains the proceedings of the session on End-User Development held at HCI International (Greece, June 2003).

October 2003

Table of Contents

Contributions, Costs and Prospects for End-User Development

Alistair Sutcliffe, Darren Lee & Nik Mehandjiev

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

Abstract

End-user development (EUD) has been a Holy Grail of software tool developers since James Martin launched 4th generation computing environments in the early eighties. Even though there has been considerable success in adaptable and programmable applications, EUD has yet to become a mainstream competitor in the software development marketplace. This paper presents a framework that critically evaluates the contributions of EUD environments in terms of the domains they can address, the modality and media of user-system communication, and degree of automation in the development process. The second part of the paper describes a socio-economic model of EUD costs and motivations.

1 Introduction

The functionality of office-style products such as database management systems, spreadsheets and word processors has been extended with macros, scripts, style sheets and other types of programmed instructions. However, in spite of some advances in end-user development (EUD) since the concept was launched in the early 1980s (Martin, 1984), EUD products are not commonplace; instead, they are an add-on to standard COTS products. This paper investigates the psychological and technological issues behind end-user development in an attempt to understand where the future research challenges lie.

2 Definitions and Concepts

First it is necessary to expand a little on the definition of end-user development and the narrower sense of end-user programming. Programming seems to be obvious: we write out a set of instructions which the computer interprets resulting in some behaviour. Unfortunately, the act of giving instructions is sometimes an implicit act. Take programming a VCR as an example. The user completes a form-filling dialogue using buttons and possibly numbers on a remote control. Programmes (on the TV) are selected to record in a sequence. The sequence, represented as a data structure hidden in the VCR, constitutes a set of instructions that will be executed at the appropriate time. End-user programming may therefore be defined as "Creating a data structure that represents a set of instructions either by explicit coding or by interaction with a device. The instructions are executed by a machine to produce the desired outputs or behaviour". End-user development widens the definition to include designed artefacts rather than instructions *per se*. Development may involve design by composition using ready-made components, or design of artefacts by powerful tools. Indeed, end-user development is differentiated from any design activity simply because non-domain experts carry it out. There are two general approaches to

helping users to design. In one case the computer is an intelligent design assistant that tracks the user's actions and infers what might be required. This approach is based on programming-by-example (Lieberman, 2001) and extends adaptable user interfaces that automatically change to fit a user profile or react to the user's behaviour. In the other approach, initiative is left with the user and the system provides powerful tools to support design activity (e.g. Agentsheets: Repenning, 1993). End-user development is a complex field which includes different approaches to helping users instruct machines and design artefacts. To investigate the research issues and properties of EUD products we propose a set of dimensions to classify designs.

2.1 Dimensions of EUD

The first dimension (see Figure 1) describes the scope of the EUD environment. Some systems are developed with the intention of supporting users in a narrow domain of expertise. Programmable queries on protein structures is an example in BioInformatics (Stevens et al., 2000). Such applications are task- and domain-specific. On the other hand, many EUD environments are intended to be general-purpose tools that can be applied to a wide variety of problems. In a similar manner to expert programming languages, EUD systems vary in scope from specific to general.

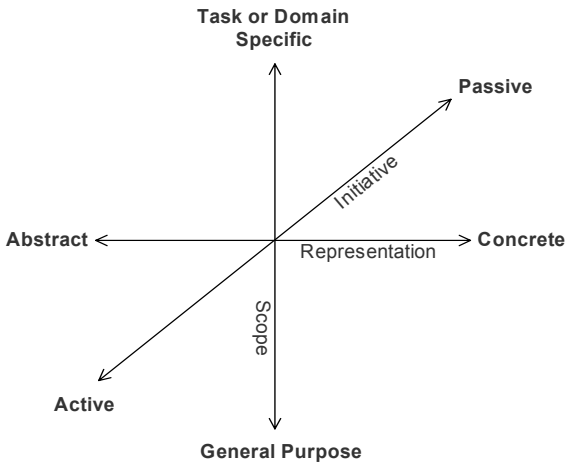

Figure 1: Dimensions of end-user development

The second dimension concerns the means of communicating with the user. Communication may use natural language and natural user actions; alternatively, a more formal language may be used which the end user has to learn. The modality of communication is also involved; for instance, instructions might be given by drawing intuitively understood marks and gestures (e.g. Palm Pilot), or manipulating a set of physical objects (e.g. turning a set of dials to program a washing machine). More formal communication can be achieved by symbolic text or diagrammatic languages; the formal syntax might be spoken, although this is unlikely. This dimension may also be described as a range from abstract (non-natural) to concrete (natural) representations. Representations could be assessed for naturalness and other properties such as changeability, comprehensibility, etc., using cognitive dimensions (Green & Petre, 1996). The key psychological trade-off for the naturalness dimension is the learning burden imposed on the end user by any artificial language versus the errors in interpretation that may arise from ambiguities inherent in less formal means of communication.

System initiative forms the third dimension for EUD. Systems might leave initiative completely with the user and just provide a means of instructing the machine. At the other extreme, intelligent systems infer the user's wishes from demonstrated actions or tracking user behaviour, and then take the initiative to create appropriate instructions or behaviour. In between are systems that provide users with development tools but constrain their actions so that only intelligible or appropriate instructions are given. System initiative may also be mixed, so in Domain Oriented Design Environments (Fischer, 1994), the system is mainly passive but it does embed critics which take the initiative when the system spots the user making a mistake. The dimensions are not completely orthogonal. For instance, natural communication in English implies some system initiatives in interpreting and disambiguating users' instructions, either automatically or by a clarification dialogue. Further dimensions may be added in the future; for instance, the concepts or subject matter represented by an EUD environment. The dimensions can be used to assess the psychological implications of different EUD approaches; however, we also need to consider the effort of development and user motivation. To investigate these issues we turn to a cost-benefit model.

2.2 Cost-benefit modelling EUD

EUD essentially out-sources development effort to the end user. Hence one element of the cost is the additional design time expended. Another cost is learning. This is a critical cost in EUD 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. Learning to use an EUD environment is an up-front cost that has to be motivated with a perceived reward in improved efficiency or empowered work practice. Cost of errors is a significant penalty for EUD users both in operation and learning. Furthermore, errors have a demotivating effect. Cost of EUD to the user can be assessed in terms of the time taken to learn to use the EUD product and possibly its language, the requirements or specification effort entailed in refining general ideas into specific instructions, the programming effort, followed by time for testing and correcting from errors. The trade-offs between effort and reward can be summarised as a set of motivating 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. Generally, the better the fit between users' needs and application functionality, the greater the users' satisfaction; however, product fit will be a function of the generality/specialisation dimension of an application. This can be summarised in the principle of user satisfaction:

- *The user satisfaction supplied by a general application will be inversely proportional to product complexity and variability in the user population.*

Complexity may be measured by counts of functional requirements or function points in an implemented system. More complex products will satisfy people less because they impose a larger learning burden; furthermore, 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, hence:

- *User effort in customising and learning software is proportional to the perceived utility of a product in achieving a job of work or entertainment.*

Our 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 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, with the corollary that inappropriate adaptation inflicts a penalty on our motivation (cost of diagnosing mistakes, cost of working around, and negative emotions).*

Hence adaptation cannot afford to be wrong, but adaptation is one of the most difficult problems for machines to model. The cost-benefit profiles for different EUD approaches is illustrated in Figure 2.

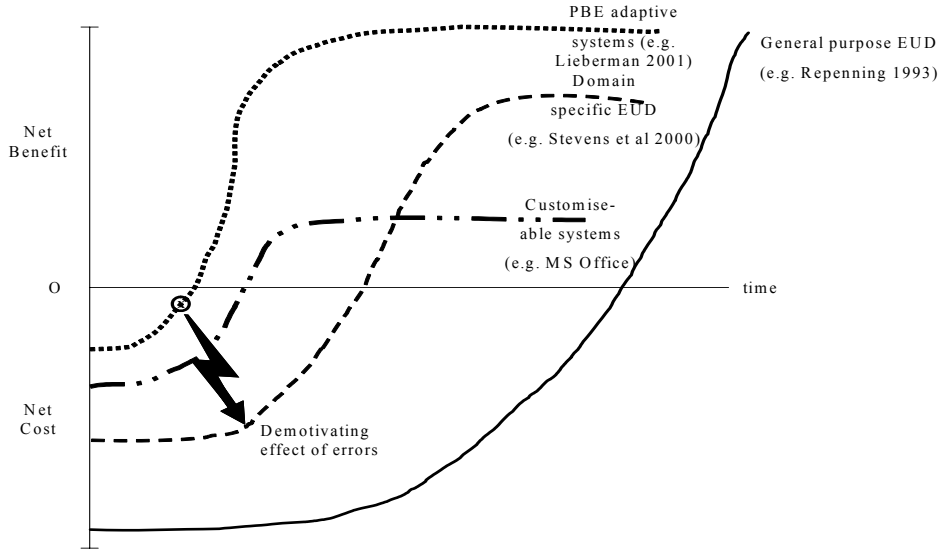

Figure 2: Cost-benefit profiles for EUD approaches

General purpose EUD tools have a longer learning curve, so the cost-benefit balance is negative for a long time period. Users have to be well motivated initially and their motivation maintained during the training period. This can be helped by ready-made EUD applications that can be tailored, by shareable repositories of components, and by models. Domain-specific EUD should have a more rapid learning time since the complexities of general syntax and vocabulary are avoided; furthermore, once competence is attained the rewards rapidly accrue. However, domain-specific EUD is by definition limited to one domain, so there may be a plateau effect on reward. This may not be important for users with a single domain that does not evolve; however, most applications face changing requirements and domain-specific languages can become limited in short time scales. Customisable applications imply less effort and more immediate reward, although the level of reward is lower because the ability to change the product to the user's wishes in inevitably limited to functions already programmed into the product. The level of effort also depends on complexity; as more customisable features are provided, complexity increases. Most

users don't use customisation facilities in office products, so the effort-reward trade-off does not appear to have been solved for this approach. Finally, adaptable products and programming-by-example (PBE) lower costs considerably so rewards are perceived quickly; however, this will only be realised in the absence of error. Early errors are critical. Users' motivation can be destroyed by annoying errors in the early stages of reuse, but if rewards are achieved without mistakes, then the user motivation may enable later errors to be tolerated. This suggests a gradual unfolding of PBE and adaptive products or simple applications to build up user motivation.

3 Conclusions

EUD is related to HCI fields of intelligent user interfaces, programming-by-demonstration, adaptive user interfaces and development tools. While considerable technical progress has been made, few attempts have been made to assess the acceptability of these technologies. The framework presented in this paper is a first step in this direction. Our analysis indicates the importance of connecting user motivation to the perceived reward of using EUD tools. User motivation requires considerable research since it will vary by the domain, and by how it is delivered through promotion, training, or functionality embedded in the tool (e.g. wizards, tutors, reuse faculties). The balance between cost and benefit suggests a graded exposure to complexity. Following Carroll's minimal manual and training wheels approach (Carroll, 1990) exposes the user to simple examples and a limited functionality, first to establish confidence and reduce errors. Early reinforcement of motivation will enable users to climb over the hump of effort into benefit. For system initiative approaches, less initial motivation may be required since the user has less to learn, but the critical success factor will be avoiding early errors.

Acknowledgements

This work was partially supported by the EU 5th Framework programme, Network of Excellence EUD (End-User Development) Net.

References

Carroll, J. M. (1990). *The Nurnberg Funnel: Designing minimalist instruction for practical computer skill*. Cambridge, MA: MIT Press.

Fischer, G. (1994). Domain-Oriented Design Environments. *Automated Software Engineering*, *1*(2), 177-203.

Green, T. R. G., & Petre, M. (1996). Usability analysis of visual programming environments: A cognitive dimensions framework. *Journal of Visual Languages and Computing*, *7*, 131-174.

Lieberman, H. (Ed.) (2001). *Your wish is my command: Programming by example*. San Francisco: Morgan Kaufmann.

Martin, J. (1984). *An information systems manifesto*. London: Prentice-Hall, 1984.

Repenning, A. (1993). *Agentsheets: A tool for building domain oriented-dynamic visual environments. Technical report, Dept of Computer Science, CU/CS/693/93*. Boulder, CO: University of Colorado.

Stevens, R., Baker, P., Bechhofer, S., Ng, G., Jacoby, A., Paton, N. W., Goble, C. A., & Brass, A. (2000). TAMBIS: Transparent Access to Multiple Bioinformatics Information Sources. *BioInformatics*, *16*(2), 184-186.

Domain-Expert Users and their Needs of Software Development[1]

M.F. Costabile°, D. Fogli, C. Letondal+, P. Mussio*, A. Piccinno°*

° DI - Università di Bari
Via Orabona 4
Bari, Italy
[costabile, piccinno]@di.uniba.it

* DEA - Università di Brescia
Via Branze 38
Brescia, Italy
[fogli, mussio]@ing.unibs.it

+ Pasteur Institute
25, rue du Dr Roux
Paris, France
letondal@pasteur.fr

Abstract

There are several categories of end-users of computer systems, depending on their culture, skills, and types of tasks they perform. This paper recognizes the problem of 'user diversity' even among people of the same technical or scientific tradition, and focuses on the study of a specific category of end-users, that we call domain-expert users: they are professionals in some domain different from computer science, who need to use computers in their daily work. We analyse the activities they usually perform or are willing to perform with computers and we identify their real needs of carrying out activities that result in the creation or modification of software artefacts.

1 Introduction

The ever increasing spread of computer environments in the information society is determining a continuous growth of the end-users population as broad as possible. Such end-users have different needs and backgrounds, and operate in different contexts. The following definition of end-users is given in (Cypher, 1993): "A user of an application program. Typically, the term means that the person is not a computer programmer. A person who uses a computer as part of daily life or daily work, but is not interested in computers per se." It is evident that several categories of end-users can be defined, for instance depending on whether the computer system is used for work, for personal use, for pleasure, for overcoming possible disabilities, etc.

Brancheau and Brown analyse the status of what they call *end-user computing* and define it as "… the adoption and use of information technology by people outside the information system department, to develop software applications in support of organizational tasks" (Brancheau & Brown, 1993). In this survey, they primarily analysed the needs of users that are experts in a specific discipline, but not in computer science. In our experience, we have often worked with end-users that are experts in their field, that need to use computer systems for performing their work tasks, but that are not and do not want to become computer scientists. This has motivated the definition of a particular class of end-users, that we call *domain-expert users* (*d-expert* in the following): they are experts in a specific domain, not necessarily experts in computer science, who use computer environments to perform their daily tasks. They have also the responsibility for induced errors and mistakes. In this paper, we focus on such users and analyse the activities they usually perform or are willing to perform with computers. This analysis shows that d-experts have real needs of performing some programming activities that result in the creation or modification of software artefacts.

In the rest of this paper, Section 2 describes some features of d-experts. Section 3 reports about activities d-experts have the need to perform. Section 4 describes real situations in which programming needs occur and Section 5 concludes the paper.

[1] The support of EUD-Net Thematic Network, sponsored by EC, project IST-2001-37470, is acknowledged.

2 Characterizing domain-expert users

In scientific and technological domains, d-experts communicate with each other through documents, expressed in some notations, which represent abstract or concrete concepts, prescriptions, and results of activities. Recognizing users as d-experts means recognizing the importance of their notations and dialects as reasoning and communication tools. As designers, we are challenged to create virtual environments, in which users interact using a formal representation of their traditional languages and with virtual tools that recall the real ones with which users are familiar. This is a hard challenge, because d-expert communities develop in time from the experience different notations, which reflect the different technical methods, languages, goals, tasks, ways of thinking, and documentation styles.

Often, dialects arise in a community, because the notation is applied in different practical situations and environments. For example, mechanical drawings are organized according to standard rules, which are different in Europe and in USA. Explicative annotations are written in different national languages. Often the whole document (drawing and text) is organized according to guidelines developed in each single company. The correct and complete understanding of a technical drawing depends on the recognition of the original standard as well as on the understanding of the national (and also company developed) dialects. Similar cases are quite common: d-experts of a same community constitute different sub-communities depending not only on user skill, culture, knowledge, but also on specific abilities (physical/cognitive), tasks, and context. Recognizing the diversity of users calls for the ability to represent a meaning of a concept with different materialization, e.g. text, images or sound, and to associate to a same materialization a different meaning according, e.g., to the context of interaction.

An important phenomenon, often observed in Human-Computer Interaction (HCI), is that "using the system changes the users, and as they change they will use the system in new ways" (Nielsen, 1993). In turn, the designer must evolve the system to adapt it to its new usages; we called this phenomenon *co-evolution of users and systems* (Arondi et al., 2002). In (Bourguin et al., 2001) it is observed that these new uses of the system determine the evolution of the user culture and of her/his models and procedures of task evolution, while the requests from users force the evolution of the whole technology supporting interaction.

Co-evolution stems from two main sources: a) user creativity: the users may devise novel ways to exploit the system in order to satisfy some needs not considered in the specification and design phase; and b) user acquired habits: users may follow some interaction strategy to which they are (or become) accustomed; this strategy must be facilitated with respect to the initial design.

3 Activities of domain-expert users

When working with a software application, d-experts feel the need to perform various activities that may even lead to the creation or modification of software artefacts, in order to get a better support to their specific tasks, thus being considered activities of End-User Development (EUD) in accordance with the following definition: "EUD is a set of activities or techniques that allow people, who are not professional developers, at any stage to create or modify a software artefacts for their own or shared use" (EUD-Net, 2002). The need of EUD is a consequence of user diversity and user evolution discussed in the previous section. Within EUD, we may include various tailoring activities described in the literature, and reported in the following.

3.1 Tailoring activities

Tailoring activities are defined in different ways in the literature; they include adaptation, customization, end-user modification, extension, personalization, etc. These definitions partly

overlap with respect to the phenomena they refer to, while often the same concepts are used to refer to different phenomena.

In (Wulf, 1999), tailorability is defined as the possibility of changing aspects of an application's functionality, during the use of an application, in a persistent way, by means of tailored artefacts; the changes may be performed by users that are local experts. Tailorability is very much related to adaptability. In (Trigg et al., 1987), a system is adaptable if an end-user "produces new system behaviour without help from programmers or designers". There are four levels for being adaptable: 1) *flexible* - objects and behaviours can be interpreted and used differently; 2) *parameterizable* - alternative behaviours can be chosen by the user; 3) *integrable* - the system can be integrated with other components, internal or external, 4) *tailorable* - users are allowed to change the system itself by building accelerators, specializing behaviour, or adding new functionality. Thus, tailoring involves the creation of new functionalities by end-users.

In (Mackay, 1991) and in (Nardi, 1993) empirical studies are reported on activities performed by end-users, and generally defined as tailoring activities. Mackay analyses how users of a UNIX software environment try to customise the system, intending as customisation the possibility of modifying software to make persistent changes. She finds that many users do not customise their applications as much as they could. This also depends on the fact that it takes too much time and deviates from other activities. Nardi conducted empirical studies on users of spreadsheets and CAD software. She found out that those users actually perform activities of end user programming, thus generating new software artefacts; these users are even able to master the formal languages embedded in these applications when they have a real motivation for doing so.

Mørch specifies three main categories of tailoring: customisation, integration, and extension (Mørch, 1997). *Customisation* usually consists of a set of preferences configurable by the user through a preference form, in which a user can set parameters for the various configuration options the application supports. *Integration* goes beyond customization by allowing users to add new functionality to an application, without accessing the underlying implementation code. Instead, users tailor an application by linking together predefined components within or across the application. *Extension* refers to the case in which the application doesn't provide, by itself or by its components, any functionality that accomplishes a specific user need, thus adding a new functionality generates a radical change in the software. In the above categorization, there are instances that cut across categorical boundaries.

3.2 Two classes of domain-expert activities

The brief overview given in the previous section shows that different meanings are associated to tailorability and adaptability. To avoid ambiguity, we propose two classes of d-expert activities:

Class 1. It includes activities that allow users, by setting some parameters, to choose among alternative behaviours (or presentations or interaction mechanisms) already available in the application; such activities are usually called parameterisation or customization or personalization.

Class 2. It includes all activities that imply some programming in any programming paradigm, thus creating or modifying a software artefact. Since we want to be as close as possible to the human, we will usually consider novel programming paradigms, such as programming by demonstration, programming with examples, visual programming, macro generation.

In the following, we provide examples of activities of both classes from experiences of participatory design workshops in two domains, biology and earth science.

Activities belonging to Class 1 are:

- *Parameterization*. It is intended as specification of unanticipated constraints in data analysis. In this situation, observed very often, the d-expert wishes to guide a computer program by indicating how to handle several parts of the data in a different way; the difference can just lay in

associating specific computation parameters to specific parts of the data, or in using different models of computations available in the program. In biology, this is related to protocol design.

- *Annotation*. D-experts often write comments next to data and result files in order to remember what they did, how they obtained their results, and how they could reproduce them.

The following activities belong to Class 2:

- *Modelling from the data*. The system supporting the d-expert derives some (formal) models from observing data, e.g. in (Blackwell, 2000) a kind of regular expression is inferred from selected parts of aligned sequences, or in (Arondi et al., 2002) patterns of interactions are derived.
- *Programming by demonstration*. D-experts show examples of properties occurrences in the data and the system infers from them a (visual) function.
- *Formula languages*. This is available in spreadsheets and could be extended to other environments, such as Biok (Biology Interactive Object Kit) that is a programmable application for biologists (Letondal, 2001). The purpose of Biok is twofold: to analyze biological data such as DNA, protein sequences or multiple alignments, and to support tailorability and extensions by the end-user through an integrated programming environment.
- *Indirect interaction with application objects*. As opposed to direct manipulation, a command language can be provided to script objects.
- *Incremental programming*. It is close to traditional programming, but limited to changing a small part of a program, such as a method in a class. It is easier than programming from scratch.
- *Extended Annotation*. A new functionality is associated with the annotated data. This functionality can be defined by any technique previously described.

4 Examples of EUD applications

In this section we describe some situations that show the real need of environments with EUD capabilities, as emerged in our work with biologists and earth scientists.

Experience acquired at the Pasteur Institute during several years indicates that in the field of biology software for academic research there are two types of software development: 1) large scale projects, developed in important bioinformatics centres, such as the European Bioinformatics Institute; 2) local development, by biologists who know some programming language, in order to deal with daily tasks, such as managing data, analysing results, or testing scientific ideas. We are here interested in the second type of development, since it can be considered end-user development. Moreover, it is worth mentioning that many biologists do not know anything about programming, and yet feel the need of modifying the application they use to better fit their needs. Below is a list of real programming situation examples, drawn from interviews with biologists, news forum, or technical support at the Pasteur Institute. These situations occurred when working with molecular sequences, i.e., either DNA or protein sequences: *scripting*, i.e. search for a sequence pattern, then retrieve all the corresponding secondary structures in a database; *parsing*, i.e. search for the best match in a database similarity search report but relative to each subsection; *formatting*, i.e. renumber one's sequence positions from -3000 to +500 instead of 0 to 3500; *variation*, i.e. search for patterns in a sequence, except repeated ones; *finer control on the computation*, i.e. control in what order multiple sequences are compared and aligned (sequences are called aligned when, after being compared, putative corresponding bases or amino-acid letters are put together); *simple operations*, i.e. search in a DNA sequence for some characters.

Considering the domain of earth science, we worked with scientists and technicians who analyse satellite images and produce documents such as thematic maps and reports, which include photographs, graphs, etc., and textual or numeric data related to the environmental phenomena of interest. Two sub-communities of d-experts are: 1) photo-interpreters who classify, interpret, and annotate remote sensed data of glaciers; 2) service oriented clerks, who organize the interpreted images into documents to be delivered to different communities of clients. Photo-interpreters and

clerks share environmental data archives, some models for their interpretation, some notations for their presentation, but also have to achieve different tasks, documented through different sub-notations and tools. Therefore, their notations can be considered two dialects of the Earth Scientist & Technologist general notation.

5 Conclusions

In this paper, we have focused on a specific category of end-users, called domain-expert users, and have analysed the activities they usually perform with computers as well as the activities they would like to perform in order to get a better support from computer systems to their daily work. The two examples in the previous section make the needs of d-experts emerge. In the case of biologists unpredictable needs may arise at any time, which require some kind of programming, even if such programming is rather simple most of the time. However, existing software does not usually provide any programming capability. Thus, the biologists have often to program everything from scratch, which is usually very difficult for them. In the case of earth science, photo-interpreters need software tools for interactive image processing and extended annotation. They interactively identify and classify parts of glacier images and associate to them an extended annotation, constituted by a textual part and a program created by selecting some computations on the basis of the observed data. In this way, photo-interpreters create new software artefacts, which are managed through user-defined widgets. Clerks will use these widgets to interact with the programs made available by the photo-interpreters to produce the required documents.
Thus, it is a challenge for us, as designers of computer systems more accessible to their end-users, to develop programming paradigms and software environments that are adequate to the needs of end-users.

6 References

Arondi, S., Baroni, P., Fogli, D., Mussio, P. (2002). Supporting co-evolution of users and systems by the recognition of Interaction Patterns. *Proc. of AVI 2002*, Trento (I), May 2002, 177-189.

Blackwell, A. (2000). See What You Need: Toward a visual Perl for end users. *Proc. of Workshop on visual languages for end-user and domain-specific programming*, September 10, 2000, Seattle, WA, USA.

Bourguin, G., Derycke, A., & Tarby, J.C. (2001). Beyond the Interface: Co-evolution inside Interactive Systems - A Proposal Founded on Activity Theory. *Proc. of IHM-HCI*.

Brancheau & Brown. (1993). The Management of End-User Computing: Status and Directions. *ACM Computing Surveys*, 25 (4), 437-482.

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

EUD-Net. (2002). http://giove.cnuce.cnr.it/EUD-NET/pisa.htm

Letondal, C. (2001). Programmation et interaction. PhD thesis, Université de Paris XI, Orsay.

Mackay, W.E. (1991). Triggers and Barriers to Customizing Software. *Proc. CHI'90 Human Factors in Computing Systems*, New Orleans, Apr. 27 – May 2, 153-160. ACM Press.

Mørch, A. (1997). Three Levels of End-User Tailoring: Customization, Integration, and Extension. In M. Kyng & L. Mathiassen (eds.), *Computers and Design in Context*, (51-76). The MIT Press, Cambridge.

Nardi,B.(1993). A small matter of programming: perspectives on end user computing.MIT Press, Cambridge.

J. Nielsen. (1993). Usability Engineering. Academic Press, San Diego.

Trigg, R.H., Moran, T.P., & Halasz, F.G. (1987). Adaptibility and Tailorability in NoteCards. *Proc. of INTERACT '87*, Stuttgart, 723-728. Elsevier Science Publishers.

Wulf, V. (1999). "Let's see your Search-Tool!" - Collaborative use of Tailored Artifacts in Groupware. *Proc. of GROUP '99*, Phoenix, USA, Nov.14-17, 50-60. ACM-Press, New York.

Challenges for End-User Development in CE devices

Boris de Ruyter
Philips Research
boris.de.Ruyter@philips.com

ABSTRACT

To provide an answer to the potential challenges of technology trends with regard to user-system interaction, the vision of Ambient Intelligence is introduced. By positioning human needs in the center of technology developments, Ambient Intelligence requires interactive systems to be personalized, context-aware, adaptive and anticipatory. Two examples of such systems and their need for end-user development are discussed.

Keywords

Consumer Electronics (CE), Ambient Intelligence, Context-awareness, user experience

INTRODUCTION

From the area of technology development (see Figure 1) we learn that, amongst other performance indicators, the **storage** capacity and **connectivity** bandwidth increase rapidly. By having more storage and high bandwidth it becomes possible to deliver large amounts of Audio / Video content to CE devices.

This could bring forward the situation of content overload to consumers. In terms of storage capacity for example, we see the emergence of high capacity optical storage media (today up to 22 Giga Byte) small enough to be integrated in many devices including portable systems. Connectivity is being supported by many different standards going from short-range wireless (low power) to full in-home networks for streaming high quality multimedia content.

In order to cope with this potential content overload, more functionality (such as different query and content management methods) will be introduced. The danger of this approach is that consumers will spend more time on operating devices than actually enjoying the content they want.

By positioning the human needs in the center of technology development, **Ambient Intelligence** aims at providing an answer for these scenarios.

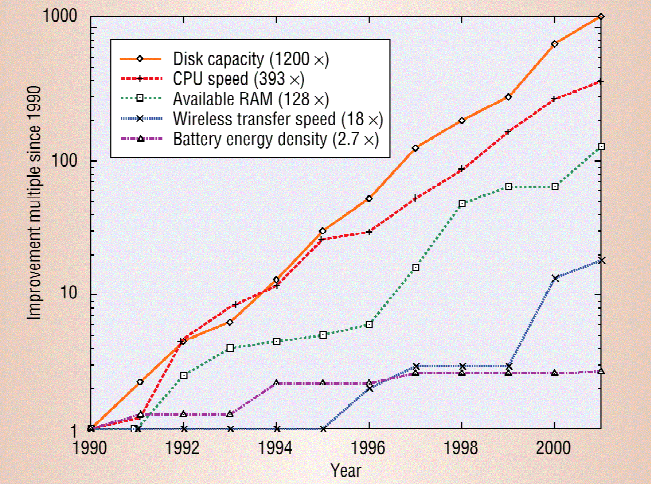

Figure 1: Moore's law for trends in storage, CPU, memory, wireless connectivity and battery technology

Ambient Intelligence is a vision of electronic environments that are sensitive and responsive to the presence of people [1]. Realizing user experiences and serving human needs (rather than pushing technology forward) are the main objective of this vision. These user experiences are not linked to one particular device but are realized by a network of intelligent devices present in the environment.

The different aspects of Ambient Intelligence are summarized in **Figure 2**.

Embedded	**Many invisible distributed devices in the environment**
Context aware	**that know about their situational state**
Personalized	**that can be tailored towards your needs**
Adaptive	**that can change in response to you and your environment,**
Anticipatory	**that anticipate your desires without conscious mediation**

Figure 2: aspects of Ambient Intelligence

Central in the realization of these aspects is that interactive systems should have some form of intelligence. However, given the fundamental need of users to be in control, end-user development becomes important to give users the ability to modify or program the behavior of their intelligent environment.

In the next section two scenarios requiring **end-user development** are discussed. While the first example describes a context-aware system that uses its situational state to change its behavior, the second example describes an interactive system by which users can personalize the experience of waking up.

The context aware remote control

One important property of intelligent systems is their awareness of the context in which they are being used. By adding some sensor and reasoning technology, a device can be made adaptive and exhibit adequate behavior for a given context [3].

As an example of a context-aware device, a universal remote control (based on the Philips PRONTO) with the ability to control different devices (such as TV, Audio set, etc.) is augmented with several context sensors that capture information with regard to the presence of people and devices in the environment. In addition to device control, the device is able to present an Electronic Program Guide (EPG) and give reminders for upcoming programs that match the preference profile of the user.

Figure 3: the concept of a context-aware remote control implemented on the PRONTO

By reasoning about the information obtained by these sensors, the device can (a) display an adaptive user interface to access the functionality relevant for the context of use[1] and (b) modify the way of reminding the user of upcoming programs that match the preference profile of this user[2].

The behavioral rules of the device that use the sensor information are not fixed in the software of the device but are represented by means of production rules that can be processed by an inference engine running on the context-aware remote control. To provide users with the ability to modify these behavioral rules, adequate programming tools need to be developed. Today, users of the **Philips PRONTO** can use the **ProntoEdit**[3] tool to modify the look-and-feel of

[1] Depending on the devices detected in the environment, the remote control can adapt the functionality offered through the user interface

[2] Depending on the usage context (noisy environment, multi-user situation, etc.) the remote control can adapt the mechanism of providing user feedback

[3] The **ProntoEdit** tool can be retrieved freely from the **Philips PRONTO** Internet site. End-users can publish their designs and share these with other Pronto users.

their universal remote control (see Figure 4). This tool allows end-users to design their own user interface for controlling CE devices with the **Philips PRONTO**. By selecting user interface elements (e.g. Button widgets) and associating these with Remote Control Infra Red codes, users can design complete interfaces.

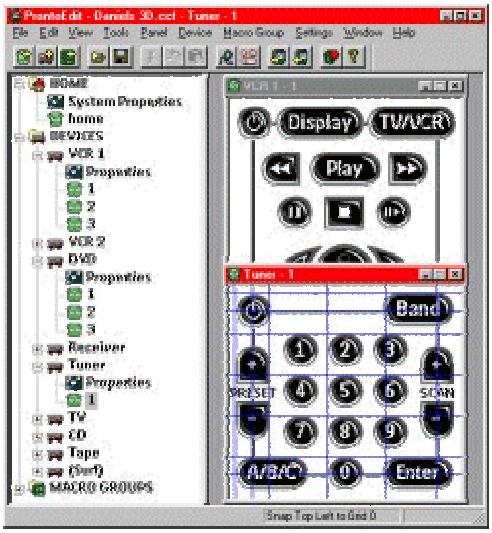

Figure 4: the end-user tool for programming the look-and-feel of the Philips PRONTO

To enable end users to modify the behavioral rules of context-aware devices, different programming metaphors need to be developed. This is one of the challenges for End User Development.

The wake-up experience

Survey studies into the characteristics of the home of the future have indicated the user need for customizing the wake-up experience, as many people are dissatisfied with their current wake up experience. By means of a questionnaire study with 120 respondents, [2] gathered user requirements for conceptualizing the optimal wake-up experience by asking respondents to describe their ideal wake-up experience. Although respondent differ greatly in the definition of their ideal wake-up experience, a common set of requirements was established. Examples of these include the requirement for a system to generate several stimuli simultaneously, the ease-of-use to create and modify a personal wake up experience and the ability to set the intensity of the different stimuli that compose a wake-up experience.

One of the major challenges is how to support people in designing their wake-up experiences. To avoid problems such as those known from VCR programming, the need for a simple but creative programming means was investigated via a workshop. Different concepts for creating a wake-up experience were collected and weighted (in terms of their feasibility and novelty). The selected concept is based on the analogy of making a painting.

Using a pen on a pressure-sensitive display, users can ‘paint’ their desired wake up experience. The display can be positioned on the bedside table where it can act as any normal alarm clock, just showing the time. However, when the pen approaches the display, the clock changes into a painting canvas. Here, users can select a certain time interval, for instance from 7.00 to 7.30 AM, for which they can start painting their desired wake up experience. A timeline for the interval is shown at the bottom of the canvas. People can choose a color from a palette of predefined wake-up stimuli, such as sounds of nature, lighting, coffee and music. The position of a stroke determines the time of ‘activation' of the stimulus, whereas the thickness of a stroke, controlled by the pressure on the pen, represents the intensity of the stimulus. At the moment of ‘painting' there is immediate feedback on the type and intensity of the stimulus that is set (except for the coffee maker for practical reasons). For instance, while making a green stroke, sounds from nature are played with the volume adjusted to the current stroke thickness.

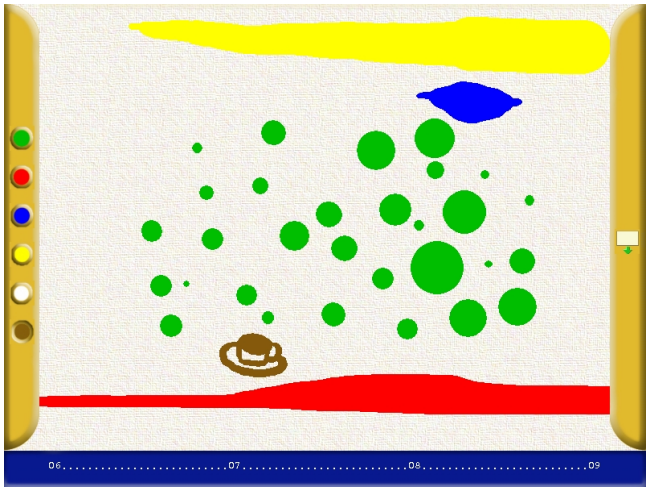

Figure 5: the interface for painting a wake-up experience

In the morning at the adjusted time interval the system generates the created 'wake up experience' by controlling the devices in the networked environment (such as lighting, coffeemaker, music, fan etc.). Figure 5 shows an example of a 'painted wake up experience'. In this example the system would start to raise the room temperature (red), then activate soft lights (yellow) and soft sounds of nature (green). These stimuli will gradually increase in intensity. The coffee maker will be switched on after some time (brown) and somewhat later music will be played for a few minutes (blue).

Conclusions

Technology trends can lead to future usage scenarios of consumer electronics that require users to interact more with system functionality than actually consuming Audio/Video content. The vision of Ambient Intelligence provides a framework in which embedded technology adapts to the needs of these users by being personalized, context-aware, adaptive and anticipatory to the needs of users. However, by adding intelligence to interactive systems, we emphasize the importance of end-user development given the need for end-users to be in control. Two applications of Consumer Electronics that require end-user development are presented. These applications emphasize the need for suitable models of end-user development in the area of consumer electronics.

REFERENCES

1. Aarts, E., Harwig, R., & Schuurmans, M. (2001). Ambient Intelligence. In: P.J. Denning (ed.), *The Invisible Future: The Seamless Integration Of Technology Into Everyday Life*, McGraw-Hill Professional, pp.235-250.
2. Eggen, B., Hollemans, G., & van de Sluis, S. (in press). *Cognition, Technology and Work*, Springer Verlag.
3. Lashina, T., & Vignoli, F. (2002). Decreasing the annoyance of your mobile device: a case study in context awareness, *the Third Philips User Interface Conference*, Philips Research, The Netherlands.

Shared initiative: Cross-fertilisation between system adaptivity and adaptability

Markus Klann, Markus Eisenhauer, Reinhard Oppermann, Volker Wulf

Fraunhofer Institute for Applied Information Technology FIT
Fraunhofer FIT, Schloss Birlinghoven, 53754 Sankt Augustin
{markus.klann, markus.eisenhauer, reinhard.oppermann, volker.wulf}@fit.fraunhofer.de

Abstract

In the present article we investigate a new way of how computer systems can better meet their users' requirements. We start from the well-known notions of situation-aware adaptivity, automatically carried out by the system, and adaptations, consciously carried out by the users. We indicate the shortcomings of both of these approaches and show how they can be compensated for, at least partially, by the respective other approach. We argue that such a shared initiative of both system and user adaptations, mutually supporting each other, provides a considerable advantage in keeping a computer system in line with dynamically changing user-requirements.

1 Introduction: the concept of shared initiative between system and user control

Today, software systems are faced with the problem of catering to the diverse and changing requirements of heterogeneous groups of users. Cutting down the requirements to a supposedly fitting least common denominator for *all* users means not taking up the challenge. The solution must be to master the heterogeneity of requirements and provide means to handle it. In this article, we investigate how software systems can reach a better fit with the diverse requirements of their users, by considering the joint benefits of two different approaches[1]: First, situation-aware adaptivity, meaning that the system adapts automatically to its users according to the situational context. Second, adaptability, meaning that the users themselves can substantially customize the system through tailoring activities.

Both of these approaches keep the system flexible during usage. Such a flexible system, which adapts to its users and which the users can adapt according to their needs and preferences, should be easier to handle and should enhance the users' productivity, optimize work-loads, and increase user satisfaction.

As it is impossible to anticipate the requirements of all users, a single best or optimal system configuration is impossible. Therefore, the task is to find a suitable trade-off between automatic system adaptivity and user controlled adaptability, resulting in a flexible system through shared initiative. We expect that system flexibility can be enhanced by exploiting situation-aware adaptivity for the system's tailorability, and vice versa.

[1] Of course, there are a large number of other approaches to capturing user requirements, notably during the design phase of computer systems (e.g. participatory and scenario-based design, visual and domain-specific languages for users to express their requirements).

2 State of the art: adaptivity and adaptability

Both adaptivity and adaptability of computer systems have received much attention in research during recent years (see (Oppermann, 1994) and (Krogsæter, 1994)). But research has addressed these two aspects largely independently of one another. The question of how these two aspects might benefit from each other has not yet been thoroughly investigated.

2.1 Adaptivity

The aim of adaptivity is to have systems that adapt themselves to the context of use with respect to their functionality, content selection, content presentation and user interactions. Systems displaying such adaptive behaviour with respect to the context of use are called situation-aware.

One aspect of situation-awareness is related to properties of the user itself, like the level of qualification, current task or previous behaviour. Traditionally, these properties have been captured in user models, which have been processed to generate appropriate adaptive behaviour. Currently, situation-awareness is continuously augmented by taking more and more situational properties into account. In particular, various sensors are used to gather information on properties relating to the physical environment of the context of use, like the time of day, location, line of sight, level of noise, etc. Other situational properties of the context of use of a particular user relate to what may be called the social environment, being composed of other users, communicative and cooperative interactions, shared artefacts and common tasks. One example of such an adaptive system would be a tourist information system on a mobile computer that presents specific informations based on its user's location, movement, profile of interests etc.

The basis for a successful and effective information and communication system is providing information and functionality that is relevant and at the right level of complexity with respect to the users' changing needs. As these changing needs are largely related to the situational properties, relevance and appropriate complexity can be supported by system adaptivity, which is to say by automatic proactive selection and context-sensitive presentation of functionalities and contents.

The objective is to assist the users by proactively supplying what they actually need. This way, users are not distracted from their primary task by searching and selecting. A good quality of such adaptivity clearly depends on complete and accurate user- and context-models, as well as on correct conclusions derived from them.

2.2 Adaptability

The aim of adaptability is to empower end-users without or with limited programming skills to customize or tailor computer systems according to their individual, context- or situation-specific requirements. Approaches to adaptability include:

- End-user-friendly programming languages, see e.g. (Repenning, 2000)
- 'Programming by example'[2] respectively 'Programming by demonstration', see (Cypher, 1993) and (Lieberman, 2001)
- Component-based tailoring, see e.g. (Stiemerling et al., 1999)

By avoiding costly and time-consuming development cycles with software engineers whenever possible, such approaches allow for fast adaptations to dynamically changing requirements by letting the end-users put their domain specific expertise to the task of system customization. While currently

[2] Taken as a conscious activity and not as an accidental side-effect to usage, ‚Programming by example' constitutes a case of adaptability. But as it also requires system activity, namely deducing some function from the user's behaviour, it may also be considered a case of adaptivity.

still being in its infancy with simple adaptations like macros for word processors or e-mail filters, more sophisticated forms of adaptability should enable end-users to become the initiators of a co-evolution between the systems they are using and their own requirements as defined by their tasks, level of expertise and current working context.

3 How to enhance adaptivity through adaptability

Automatic system adaptations are only possible to the extent that the system can gather the required information for performing some adaptation function. And they are only reasonable to the extent that the system can assess which adaptation functions are suitable for the current context and in case of multiple adaptation options which one to carry out with sufficient benefit.

In some cases, sufficient information can be gathered through sensors or other sources of information, like user preferences or interaction histories to derive useful adaptations and carry them out. But in other cases the required information for some adaptation is not available or it is not possible to select an appropriate one among a set of potential candidates.

In both cases, adding adaptability can help to still benefit from adaptive system functionality. Obviously, this is done by either letting users provide missing information in order for the system to automatically carry out some adaptation function or by having the user make the selection if multiple options exist. In the example of the mobile tourist information system this could mean that the system has tracked previous user activities as a basis for making recommendations that match the user's interests. The user might adapt this functionality by either adding or correcting information or by modifying the strategy by which the system acquires information and arrives at its conclusions.

These user-adaptations can of course be very simple. They might be nothing more than providing missing parameters or actually choosing among a list of adaptation options. But they might as well be rather complex. They might consist of defining what situational properties are to be taken into account by some adaptivity function as relevant for the current context . Or they might consist of defining an actual compound function that implements an adaptivity strategy suitable for the current situation. In both cases, users can exploit their superior awareness of their situation and their domain expertise to enable system adaptivity that would otherwise not be possible.

4 How to enhance adaptability through adaptivity

As any other system functionality, adaptability can benefit from adaptivity. As seen above, user adaptations can be very complex and correspondingly difficult and cognitively demanding. System adaptivity can help reduce the cognitive load on the customizing end-user by hiding those adaptation functions that are not pertinent in the current context.

Moreover, a system might provide adaptability at different levels of complexity, geared towards adaptations of varying degrees of difficulty. An adaptive system could choose an appropriate level of tailoring complexity based on the current task and level of tailoring expertise of its user.

Alternatively, a situation-aware system might be able to identify recurring tailoring situations based on the current task, pursuit goal, involved people etc. and might be able to suggest the reuse of existing tailored artefacts to its users or to suggest getting assistance from other users who have successfully tailored in similar situations. Thus, situation-aware systems might be very important to foster collaborative tailoring and the sharing of tailored artefacts (Wulf, 1999). As an example, a groupware system might present only those configuration options to its users that are appropriate for their level of expertise and their mutual task.

Finally, system adaptivity is actually of particular importance to adaptability for a specific reason. This is because adaptability functions should be as unobtrusive as possible on the user interface dur-

ing normal use (Wulf, 2001): if the user feels no need to tailor, as little as possible of his attention should be deviated by tailoring functionality. This can be achieved by adaptively displaying only those tailoring functions on the user interface that are likely to be of use, based on such properties as current task, level of expertise, available time etc. Thus, adaptivity can play a crucial role in reducing the cognitive load of tailoring functionality and consequently raising the users' inclination to carry out tailoring activities.

5 Discussion of shared initiative: the joint benefit of adaptivity and adaptability

As explained in the preceding sections, automatic situation-aware adaptivity and manual adaptability are important features of computer systems that are supposed to sustainably cater to their users' needs. However, it was also shown that both of these features have their specific limitiations and disadvantages.

However, it was also shown that both means, the automatic situation aware adaptivity and the manual adaptability, have their specific limitations and disadvantages. In particular, the first is incomplete and imprecise and reduces the users to a passive receptionist of automatic mechanisms. The latter is costly, requiring additional effort from the users for the meta-task of tailoring the system. Nonetheless, we believe that end-user tailoring is necessary, as today's high degree of situation-aware system adaptivity does not allow for software designers to anticipate all possible ways of system behaviour. On the other hand, today's higher degree of situation-awareness may allow for better proposals for tailoring activities.

A combination of both adaptivity and adaptability may help to overcome their individual problems by preparing a spectrum of increasingly conservative best guess proposals. The best guess proposals consider the current users, task and environment and they are presented as a zoomable spectrum with appropriate content and suitable interaction primitives. Such best guess solutions empower the user itself to comfortably browse from the most specific best guess option to alternative options requiring an increasing amount of user adaptations.

6 Conclusion

In this article we have shown how the shared initiative of adaptation activities, automatically carried out by an adaptive situation-aware computer system and consciously carried out by its user may lead to a synergetic advantage with respect to a continuous close fit between the system and its user in the presence of dynamically changing requirements.

The envisioned interplay of adaptivity and adaptability obviously constitutes a new kind of system behaviour. For users to fully exploit the potential advantages of such a shared initiative, that is, to rely on system adaptivity, to confidently carry out adaptations and to benefit from their interplay, users will probably have to change their expectations on how such computer systems operate.

How this change of expectations can be facilitated and supported, as for example by visualizing dependencies and the consequences of operations or by pedagogical efforts is a question for future research.

References

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

Krogsæter, M., Oppermann R., et al. (1994). A User Interface Integrating Adaptability and Adaptivity. In R. Oppermann (Ed.), *Adaptive User Support. Ergonomic Design of Manually and Automatically Adaptable Software* (pp. 97-125). Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Lieberman, H. (2001). *Your Wish is My Command: Programming By Example*. San Francisco: Morgan Kaufmann

Oppermann, R. (1994). *Adaptive User Support. Ergonomic Design of Manually and Automatically Adaptable Software*. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Repenning, A., Ioannidou, A., & Zola, J. (2000). AgentSheets: End-User Programmable Simulations. *Journal of Artificial Societies and Social Simulation*, 3 (3).

Stiemerling, O., Hinken, R., & Cremers, A. B. (1999). The EVOLVE tailoring platform: supporting the evolution of component-based groupware. In IEEE Communications Society (Ed.), *Proceedings of the Third International Enterprise Distributed Object Computing Conference, EDOC '99 : 27 - 30 September 1999, University of Mannheim, Germany . Mannheim* (pp. 106-115). Los Alamitos, Calif.: IEEE Press.

Wulf, V. (1999). "Let's See Your Search-Tool!" - On the Collaborative Use of Tailored Artifacts. In S. C. Hayne (Ed.), *Proceedings of the international ACM SIGGROUP conference on Supporting group work, GROUP '99* (pp. 50-60). New York: ACM Press.

Wulf, V., & Golombek B. (2001). Direct Activation: A Concept to Encourage Tailoring Activities. *Behaviour and Information Technology*, 20 (4), 249-263.

User-Centered Point of View to End-User Development

Philipe Palanque

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

Rémi Bastide

LIIHS-IRIT, Université Toulouse 1
place Anatole France,
31042 Toulouse Cedex, France
bastide@irit.fr

Abstract

This paper proposes a user centered point of view for the definition and the construction of end user development environments. Taking, as input, the seven stages of actions of Norman's action theory we propose a set of guidelines for easing end user development. The main criteria for the language are "types of application covered" and "closeness of the language with respect to the application domain". The main criterion for end user development environment is the "continuous and permanent feedback" proposed by the environment. These criteria are then exemplified on PetShop environment that aims at building highly interactive applications providing using the Interactive Cooperative Objects language (an object-oriented, distributed and concurrent programming language).

1 Introduction

Designing computer-based applications belongs to the type of human activities that is higly demanding on the user's side. In this paper we propose to use Norman's activity theory as a tool for investigating where the main difficulties can occur, and thus to provide design rules for notations and environments to support user's activities and reduce difficulties.
Bringing a user centered point of view to programming environment has as already been studied (Green 1990, Blackwell 1996) but we believe that new development in the field of software engineering can enhance previous results.
The paper is structured as follows. Section 2 introduces briefly Norman's model and presents how development activities relates to it. Section 3 is devoted to ways of reducing gaps both during execution and interpretation. Section 4 presents how the ICO visual language and its CASE tool PetShop provide mechanisms to reduce such gaps.

2 Norman's Action Theory and EUD

Figure 1 (left-hand side) presents the seven stages of action and is extracted from (Norman 1998). Right-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. Left-hand side represents the set of activities that have to be carried out by the user in order to interpret the changes resulting from his/her actions in the real world.

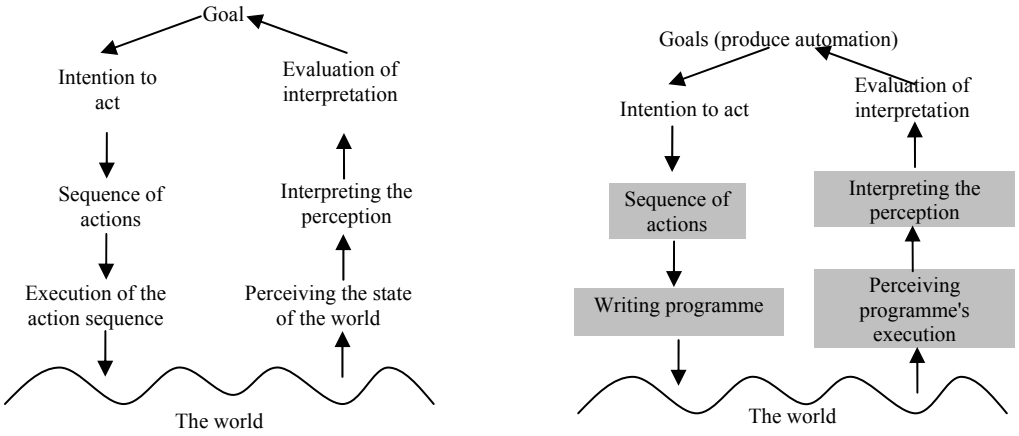

Figure 1: Seven stages of action (Norman 2001) (left)
Customization of Norman's model to EUD (right)

Figure 1 (right hand side) shows the refinement of specific activities from previous model to programming activities. The two main stages we are focusing on are:

- On the execution side, the activity of going form an intention to its actual transcription into some program,
- On the perception side, the activity of perceiving program execution and interpreting this perception.

3 Reducing Gulfs

There are several ways of reducing gulfs with respect to the models above. We investigate here two different and separated ways of reducing the gaps. The first one is related to the execution gulf and the second one is related to the evaluation gulf. Both of them are presented in next sections.

3.1 Reducing Execution Gulf

While designing a programming language, the designer must bear in mind that there is a tradeoff between the expressive power of the language and its "usability" [1] by the programmer. Due to space reasons, this very important aspect of usability of the language is not discussed in the paper.
One of the ways of making execution easier is to make description language closer to the application domain, thus making the concepts in the language close to the objects in the real word. This is of course much more important when end user programming is concerned as the end user is by definition highly connected to the real word. In fact having the end user writing the program directly avoids some classical potential usability problems when the vocabulary (used by the users in their everyday work) is different from the one used on the user interface by the computer scientists.
However, making the language closer to the application domain reduces its applicability to other domains. This has clearly been a tendency both in classical programming languages and in end

[1] For a concrete and measurable definition of usability of EUP languages please refer to (de Souza et al. 2001)

user development languages. Computer science has always been trying to provide generic programming languages that could be applied to a wide range of systems. This is a basic requirement for a programming language dedicated to professional programmers, as, for sake of efficiency, programmer's skills and knowledge must cover several potential application domains. On the other side, a lot of end user programming environments have been dedicated to specific domains such as simulation (Cypher & Smith 1995), geographic information systems (Traynor C. & Williams 1997) or air traffic control applications (Esteban et al. 1995), among others.
As a conclusion to this section, it is possible to reduce the execution gulf by providing programming languages in which constructs are close to the application domain. This is only true as the end users, by definition, only deals with applications that are heavily related to their activities and do not consider building applications for other users.

3.2 Reducing Evaluation Gulf

A way of reducing the evaluation gulf is to make easier the relationship between program edition and program execution. Usually these two activities are made available to the user in a modal way: first writing the program and then (usually in a different context) execute the program. Making these two activities separated and modal introduces difficulties in both perceiving the program execution and interpreting its behaviour (Butler et al. 1998).
In order to reduce this gulf we propose the support of these two activities in parallel within the programming environment. We have developed such a programming environment called PetShop. Its architecture as well as its use on simple case study is presented in section 4.

4 ICOs and PetShop

In previous work we have defined an object oriented distributed, concurrent and visual language called Interactive Cooperative Objects (ICO) (Bastide & Palanque 1999). This language is dedicated to the construction of highly interactive distributed applications. It is to be used by expert programmers with skills in: formal description techniques, object oriented approaches, distributed and interactive systems. Even though the programmers were not end users of the applications to be constructed, our goal was to increase usability of the language by providing ways of reducing evaluation gulf.

In order to show on a concrete example the concepts introduced above we have decided to use a simple case study in the field of components engineering for interactive applications.

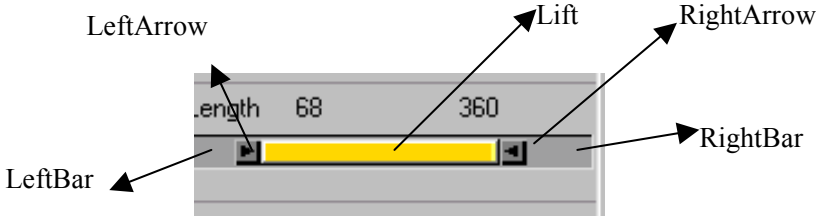

Figure 2: A simple application: a range slider (Ahlberg & Shneiderman 94)

A range slider is a basic interactive component that allows 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. This kind of compound quite complex interactors are more and more 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.

4.1 Language

Using the ICO language it is possible to describe the entire behaviour of such an interactor. We only present here a subset of this description i.e. its behaviour (see Petri net model in Figure 3). The complete description of the case study can be found in (Navarre et al. 2000).
This behavioural description models the set of events the range slider 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).
One of the problem of building such a concurrent program is, first to understand its behaviour and then to understand whether this behaviour is similar to the one that was expected.

4.2 PetShop Environment

By providing a way of having both program execution and program edition at a time PetShop allows for rapid prototyping, iterative and modeless construction of applications (Navarre et al. 2001). Figure 3 presents a snapshot of Petshop at runtime. The small window on top of the picture corresponds to the execution of the visual program (the Petri net) in the bigger window underneath.

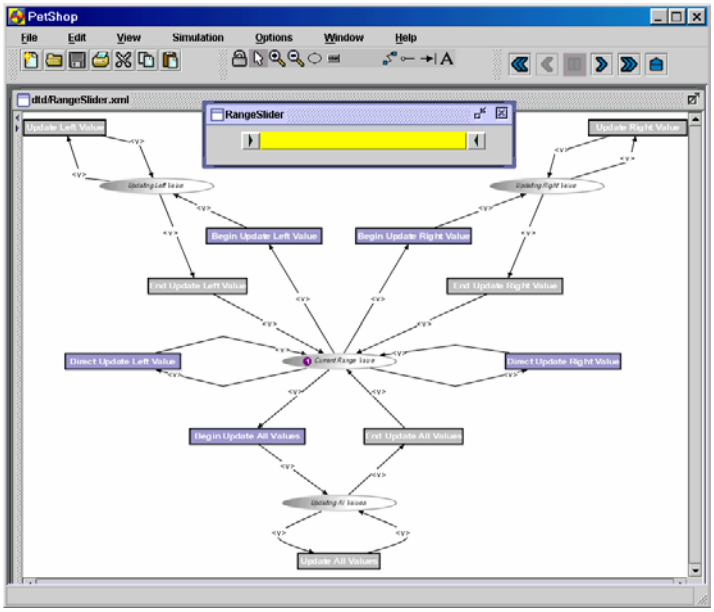

Figure 3: Integrated program edition and execution within PetShop

Thus, at a time, the programmer can interact with the application (use the range slider), see the impact of his/her action on the behaviour of the visual program. Another possibility is to modify the visual program and see immediately the impact on the program execution.
For instance, on Figure 3, 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 appeared as disabled (greyed out) and acting on it will have no effect.

5 Conclusions and Perspectives

This paper has proposed the use of Norman's model to understand the potential difficulties encountered by programmers. We have also shown how a programming environment (such as PetShop) that proposes program edition and execution in a modeless way could reduce those difficulties.
We have already conducted some early evaluation of ICO language and PetShop as part of the Mefisto LTR Esprit Project. However, in order to quantify and confirm the results of this early evaluation more usability tests should be conducted. This will be done in the framework of a military funded project starting in January 2003. This is a very important issue as pointed out in (De Souza et al. 2001).

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.

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.

Blackwell A. Metacognitive theory of visual programming: what do we think we are doing ? IEEE workshop on visual languages, pp. 240-246. IEEE computer society, 1996.

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, 1998.

Cypher A. & Smith D.C. Kidsim: end user programming of simulations. Proceedings of ACM SIGCHI CHI 95 conference, Denver, USA. pp. 27-34, 1995.

de Souza C., Barbosa S. & da Silva S. Semiotic engineering principles for evaluating end-user programming environments. Interacting with Computers, vol. 13, pp. 467-495. Elsevier. 2001

Esteban O., Chatty S. & Palanque P. Visual construction of Interactive Software. IFIP conference on Visual Database, Lausane, Switzerland. pp. 145-160, Chapman et Hall, 1995.

Green T. Cognitive dimensions of notations. People and computers V, Cambridge University Press, R. Winder & A Sutcliffe (Eds), 1990.

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. (1998). The design of everyday things. MIT press 1998.

Traynor C. & Williams M. A study of end-user programming for geographic information systems. Seventh workshop on empirical studies of programmers, pp. 140-156. 1997.

From Model-based to Natural Development

Fabio Paternò

ISTI-CNR
Via G.Moruzzi, 1 – 56100 Pisa - Italy
f.paterno@cnuce.cnr.it

Abstract
Model-based approaches aim to support development through the use of meaningful abstractions in order to avoid dealing with low-level details. Despite this potential benefit, their adoption has mainly been limited to professional designers. This paper discusses how they should be extended in order to obtain environments able to support real end-user development, by which designers can develop or modify interactive applications still using conceptual models but with a continuous support that facilitates their development, analysis, and use.

1 Introduction
One fundamental challenge for the coming years is to develop environments that allow people without particular background in programming to develop their own applications. The increasing interactive capabilities of new devices have created the potential to overcome the traditional separation between end users and software developers. 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 (Boehm et al., 2000).
End-user development in general means the active participation of end-users in the software development process. In this perspective, tasks that have traditionally been performed by professional software developers are transferred to the users, who need to be specifically supported in performing these tasks. New environments able to seamlessly move between using and programming (or customizing) can be designed. Some end-user development oriented techniques have already been added to software for the mass market, such as the adaptive menus in MS-Word or some programming-by-example techniques in MS-Excel. However, we are still quite far from their systematic adoption.
At the first EUD-Net workshop held in Pisa a definition of End User Development was established: "*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*". One of the goals of end-user development is to reach closeness of mapping: as Green and Petre put it (Green and Petre, 1996): "The closer the programming world is to the problem world, the easier the problem-solving ought to be.… Conventional textual languages are a long way from that goal". Even graphical languages often fail to furnish immediately understandable representations for developers. The work in Myers' group aims to obtain natural programming (Pane and Myers, 1996), meaning programming through languages that work in the way that people who do not have programming experience would expect. We intend to take a more comprehensive view of the development cycle, thus not limited only to programming, but also including requirements, designing, modifying, tailoring, ….. Natural development implies that people should be able to work through familiar and immediately understandable representations that allow them to easily express and manipulate relevant concepts, and thereby create or modify interactive software artefacts. On the other hand, since a software artefact needs to be precisely specified in order to be

implemented, there will still be the need for environments supporting transformations from intuitive and familiar representations into more precise, but more difficult to develop, descriptions.

2 Model-based Development

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.

In traditional software engineering, the Unified Modelling Language (UML) (OMG, 2001) 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. Like programming, in order to be effective, modelling requires the availability of suitable and usable languages and support tools. 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 or ArgoUML are the best-known examples. They inherently require 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.

The developers of UML did not pay a lot of attention to how to support the design of the interactive components of a software system. Thus, a number of specific approaches have been developed to address the model-based design of interactive systems. Since one of the basic usability principles is “focus on the users and their tasks”, it became important to consider task models. The basic idea is to focus on the tasks that need to be supported in order to understand their attributes and relations. Task models can be useful to provide an integrated description of system functionality and user interaction. This calls for identifying task allocation between the user and the system, and relating user and system views in an integrated design process, i.e., integrated modelling of user interface and system functionality. Then, the development of the corresponding artefact able to support the identified features should be obtainable through environments able to identify the most effective interaction and presentation techniques on the basis of a set of guidelines or design criteria.

Various solutions have been proposed for this purpose. They vary according to a number of dimensions. For example, the automation level can be different: a completely automatic solution can provide meaningful results only when the application domain is rather narrow and consequently the space of the possible solutions regarding the mapping of tasks to interaction techniques is limited. More general environments are based on the mixed initiative principle: the tool supporting the mapping provides suggestions that the designer can accept or modify. An example is the TERESA environment (Mori, Paternò and Santoro, 2003) that provides support for the design and development of nomadic applications, which can be accessed through different types of interaction platforms.
In order to move from model-based to natural development we have identified three key criteria that should be supported and will be discussed in the next section: integrated support of both informal and formal representations; effective representations highlighting the information of interest; and possibility of developing software artefacts from either scratch or an existing system.

3 Criteria for Natural Development Environments

In this section we discuss what design choices should be pursued to extend model-based approaches in order to obtain natural development environments.

3.1 Integrating informal and structured representations

Most end-user development would benefit by the combined use of multiple representations that can have various levels of formality. At the beginning of the design process many things are obscure and unclear. It is hard to develop precise specifications from scratch. In addition, there is the problem of clearly understanding what user requirements are. Thus, it can be useful to use the results of initial discussions to feed the more structured parts of the design process. In general, the main issue of end-user development is how to use personal intuition, familiar metaphors and concepts to obtain or modify a software artefact, whose features need to be precisely defined in order to obtain consistent support for the desired functionality and behaviour. In this process we should address all the available multimedia possibilities. For example, support for vocal interaction is mature for the mass market. Its support for the Web is being standardised by W3C (Abbott, 2001). The rationale for vocal interaction is that it makes practical operations quicker and more natural, and it also makes multi-modal (graphic and/or vocal) interactions possible. Vocal interaction can be considered both for the resulting interactive application and for supporting the development environment.

In CTTE (Mori, Paternò and Santoro, 2002), to support the initial modelling work we provide the possibility of loading an informal textual description of a scenario or a use case and interactively selecting the information of interest for the modelling work. To develop a task model from an informal textual description, designers first have to identify the different roles. Then, they can start to analyse the description of the scenario, trying to identify the main tasks that occur in the scenario's description and refer each task to a particular role. It is possible to specify the category of the task, in terms of performance allocation. In addition, a description of the task can be specified along with the logical objects used and handled. Reviewing the scenario description, the designer can identify the different tasks and then add them to the task list. This must be performed for each role in the application considered. Once designers have their list of activities to consider, they can start to create the hierarchical structure that describes the various levels of abstractions among tasks. The hierarchical structure obtained can then be further refined through the specification of the temporal relations among tasks and the tasks' attributes and objects. The use of these features is optional: designers can start to create the model directly using our editor, but such features can be useful to ease the modelling work. U-Tel (Tam, Maulsby, and Puerta, 1998) provides a different type of support: through automatic analysis of scenario content, nouns are automatically associated with the objects, and verbs with the tasks. This approach provides some useful results, but it is too simple to be generalised. For example, in some cases a task is defined by a verb followed by an object.

Another useful support can be obtained starting with the consideration that 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 (Landay and Myers, 2001) 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. 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.

3.2 Providing Effective Representations

Recent years have seen the widespread adoption of visual modelling techniques in the software design process. However, we are still far from visual representations that are easy to develop, analyse and modify, especially when large case studies are considered. As soon as the visual model increases in complexity, designers have to interact with many graphical symbols connected in various ways and have difficulties analysing the specification and understanding the relations among the various parts

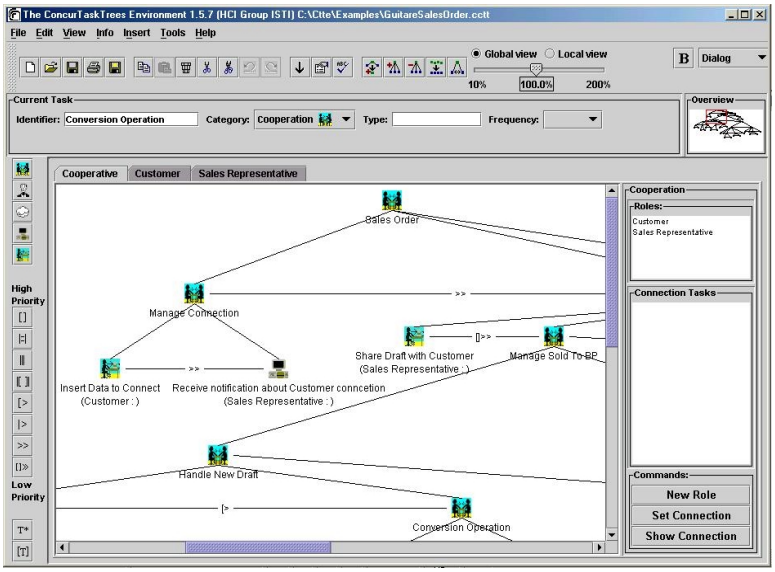

Figure 1: How CTTE provides focus+context representation.

CTTE provides support through some representations that ease the analysis of a model. It furnishes a focus+context representation of the model (see Figure 1). So, it is possible to have a detailed view of a part of the model in the main window (only a part can be displayed when large models are considered) but it still provides a small overview pane (in the right-top part) that shows the designer exactly where the part in the large window is located in the overall model. The representation in the overview pane is limited to show the hierarchical structure without indicating the task names and the icons that represent how the tasks are allocated. Identifying relations between the focus and the context window is facilitated by the hierarchical structure of the model.

In addition, an interactive system is a dynamic system so designers need to specify how the system can evolve: the sequential constraints, the possible choices, the interrupting activities and so on. When people analyse a specification, including the instances of the various temporal operators, they may have problems understanding the actual resulting dynamic behaviour. To this end, interactive simulators showing the enabled activities and how they change when any of them is performed can allow a better understanding of the actual behaviour specified.

In addition, the application and extension of innovative interaction techniques, including those developed in information visualization (such as semantic feedback, fisheye, two-hand interactions, magic lens…), can noticeably improve the effectiveness of the environments aiming to provide different interactive representations depending on the abstraction level of interest, or the aspects that designers want to analyse or the type of issues that they want to uncover.

3.3 Transformation-based environments

The starting point of 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 (from conceptual to concrete) and reverse (from concrete to conceptual) 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.

While TERESA is an example of forward engineering, WebRevenge (Paganelli and Paternò, 2002) is an example of reverse engineering: it is able to take the code of a Web site implemented in HTML and reconstruct the corresponding task model through an analysis of the interaction techniques, tags and links existing. The combination of TERESA and WebRevenge allows a process where a designer takes an existing Web site for desktop systems, reconstructs its logical model through Web Revenge, analyses and modifies it in order to redesign the application for a mobile device and generates a new version of the application for the different platform with the support of TERESA.

4 Conclusions and acknowledgments

This paper provides a discussion of how model-based design should be extended in order to obtain natural development environments. After a brief discussion of the motivations for end user development, we set forth the criteria that should be pursued in order to obtain effective natural development environments. This can be achieved by extending traditional model-based approaches. In order to make the discussion more concrete, a specific environment for model-based design (CTTE) has been considered. We gratefully acknowledge support from the European Commission through the EUD-Net Network of Excellence (http://giove.cnuce.cnr.it/eud.html).

5 References

Boehm, B. W., Abts, C., Winsor Brown A., Chulani, S., Clark, B. K., Horowitz, E., Madachy, R., Reifer, D. J., and Steece, B., (2000). Software Cost Estimation with COCOMO II, Prentice Hall PTR, Upper Saddle River, NJ.

Green, T.R.G., Petre M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework, in J. Visual Lang. and Computing, Vol.7, N.2, pp.131-174.

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

Mori G., Paternò F., Santoro C., (2002). CTTE: Support for Developing and Analysing Task Models for Interactive System Design, *IEEE Transactions in Software Engineering,* pp. 797-813, August 2002 (Vol. 28, No. 8), IEEE Press.

Mori, G., Paternò, F., Santoro, C., (2003) Tool Support for Designing Nomadic Applications, Proceedings Intelligent User Interfaces '03, pp.141-148, ACM Press, 2003.

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

Paganelli, L., Paternò, F., (2002) Automatic Reconstruction of the Underlying Interaction Design of Web Applications, Proceedings SEKE Conference, pp.439-445, ACM Press, Ischia.

Pane J. and Myers B. (1996), "Usability Issues in the Design of Novice Programming Systems" TR# CMU-CS-96-132. Aug, 1996. http://www.cs.cmu.edu/~pane/cmu-cs-96-132.html

Tam, R.C.-M., Maulsby, D., and Puerta, A., (1998). U-TEL: A Tool for Eliciting User Task Models from Domain Experts, *Proceedings IUI'98*, ACM Press, 1998.