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.

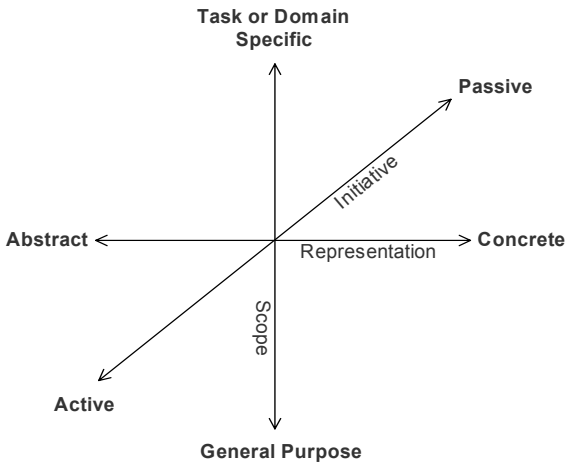

Figure 1: Dimensions of end-user development

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.

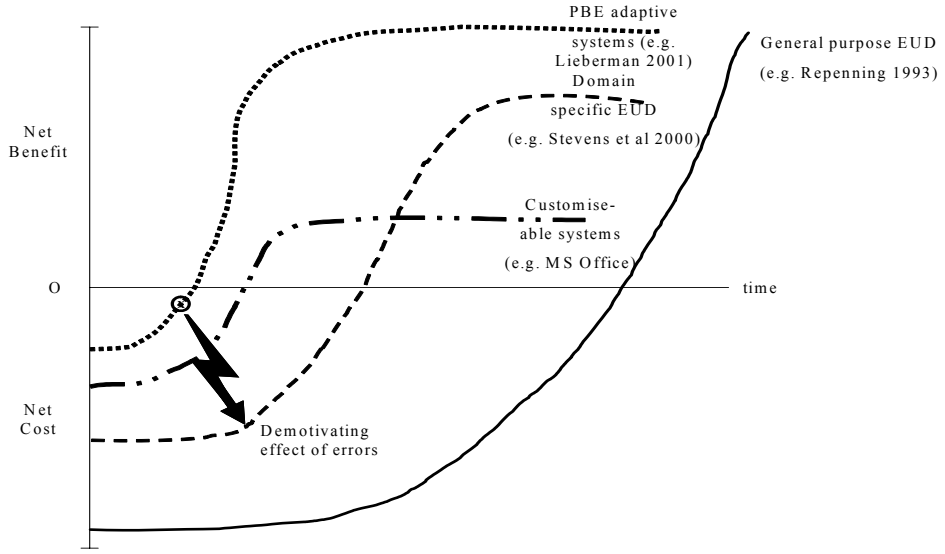

Figure 2: Cost-benefit profiles for EUD approaches

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.