2008

End User Development: Approaches Towards a Flexible Software Design

Michael Spahn
SAP Research, michael.spahn@sap.com

Christian Dorner
University of Siegen, christian.doerner@uni-siegen.de

Volker Wulf
Fraunhofer-Institut für Angewandte Informationstechnik FIT and Universität Siegen, volker.wulf@uni-siegen.de

END USER DEVELOPMENT: APPROACHES TOWARDS A FLEXIBLE SOFTWARE DESIGN

Michael Spahn, SAP Research, SAP AG, Bleichstr. 8, 64283 Darmstadt, Germany, michael.spahn@sap.com

Christian Dörner, University of Siegen, Hölderlinstraße 3, 57068 Siegen, Germany, christian.doerner@uni-siegen.de

Volker Wulf, University of Siegen, Hölderlinstraße 3, 57068 Siegen, Germany, volker.wulf@uni-siegen.de

Abstract

Acknowledging the competition in today's global markets demands enterprises to be resilient in order to survive. Therefore product life cycles shorten and new customer segments have to be addressed permanently. Obviously, such environments require flexible information systems, which can be adapted quickly to the enterprises' changing needs, without spending vast amounts of resources. Using End User Development (EUD) approaches can help to solve this dilemma by enabling software developers to create information systems that can even be adapted by technically inexperienced end users. This reduces time and costs needed for adaptations and increases their quality by avoiding potential misunderstandings between business users and IT experts. This paper presents a broad overview of existing EUD approaches. Based on this, it provides recommendations, how EUD design principles can be used conjointly, to develop embedded design environments for end users. We describe and classify EUD approaches taken from the literature, which are suitable approaches for different groups of end users. Implementing the right mixture of EUD approaches leads to embedded design environments, having a gentle slope of complexity. Such environments enable differently skilled end users to perform system adaptations on their own.

Keywords: Design Approaches, Methods and Methodologies, End User Development, Design Recommendations, Adaptable Software.

1 INTRODUCTION

The design and development of flexible standard software, which matches the needs of a maximum of customers, is still a hard issue for software engineers. On one hand it is impossible to identify all requirements of all stakeholders a priori of design time (Henderson & Kyng, 1991). On the other hand, even if this was possible, requirements of a system would not be static, leading to a growing misfit between altering customer needs and features offered by a system. Due to continuous changes, like market conditions, competitor behaviour, partner collaborations, and the legal environment, companies face considerable challenges resulting in changing workflows, processed data and information needs. As a result, one of the main requirements for software systems is their design for change. Many researchers dealt with this subject and most of them concluded that the adaptation of software is necessary during use time (Nardi, 1993; Repenning & Ioannidou, 2006). Continuous adaptations often lead to higher costs than the costs caused during their initial design, implementation and introduction phase (Wulf & Jarke, 2004). In many cases users are only able to adapt software systems indirectly, by communicating their change requirements to IT experts. This could result in communication problems between IT experts and users (Stiemerling et al., 1997), which have a negative impact on the adaptation quality. Offering users the possibility to adapt software systems by themselves, shortens the adaptation process, cuts down resulting costs, and increases the overall quality, while at the same time offering a high level of flexibility. By means of End User Development (EUD) approaches and the increasing popularity of the so-called Web 2.0 technologies (e.g. Mashups), system design will shift from "easy-to-use" to "easy-to-develop" and make the vision of designing "easy-to-use" user-customizable systems more tangible. However, technology by itself is not able to overcome the complexity inherent to software engineering (Brooks, 1987). Therefore, the research field of End User Development addresses the topic more holistic and explores how users can be enabled to adapt software systems at runtime. Lieberman et al. (2006) define End User Development to be *"a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create or modify a software artifact."* Such artefacts could be for example user interface models, workflow definitions or even source code.

This paper gives an overview of existing EUD approaches and provides recommendations, how those approaches can be used to develop embedded design environments for end users. The overview gives IS designers a guideline to select appropriate EUD approaches for the design of user-adaptable information systems. The paper is organised in five sections. In section two we describe the "gentle slope of complexity" as a meta-approach to realise user-adaptable information systems. Afterwards we give an introduction to EUD in section three and give an overview over EUD methods, tools and techniques. Based on this we provide recommendations for the design of embedded design environments in section four and conclude the paper in section five.

2 COMPLEXITY DISTRIBUTION

One reason, preventing users from adapting software systems on their own, is the high complexity of adaptation mechanisms. Analyzing the complexity of these mechanisms is a first step to identify the ability of a software system to be tailored by end users (MacLean et al., 1990). A coarse method of analysis is to identify the *complexity curve* of adaptation mechanisms, offered by a software system. To do this, it is necessary to rate the skill level needed to use an adaptation mechanism (complexity) and the power of adaptations achievable by its usage (adaptation power). If ratings for each mechanism are determined, it is possible to depict them in a diagram as shown in Figure 1 and 2. Steep slopes (indicated by dotted lines) imply barriers for users to understand, learn, and use a more powerful adaptation mechanism. Given the example of a system that only allows users to write source code to apply even the simplest adaptation of system functionality, will prevent most end users from doing any adaptations, due to the complexity of programming. A system offering only a sparse set of design

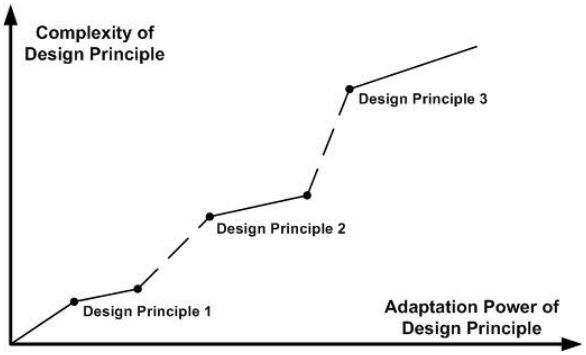

Figure 1. Steep slope of complexity

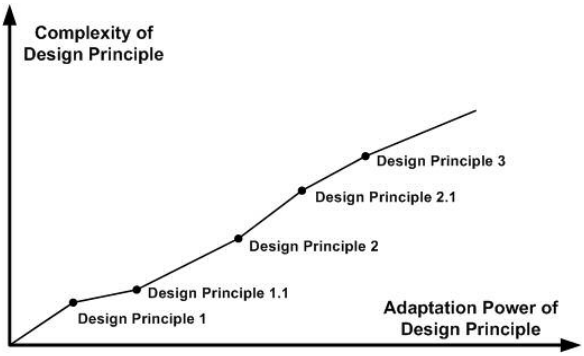

Figure 2. Gentle slope of complexity

principles for performing adaptations is depicted in Figure 1. In contrast to this, systems providing multiple adaptation mechanisms limit the additional complexity end users have to handle in order to use a more powerful adaptation mechanism (c.f. Figure 2). End users are able to start with simple adaptation mechanisms and gradually advance to more powerful adaptation mechanisms without facing insuperable barriers (MacLean et al., 1990). When examining how end users adapt software systems, this approach of using adaptation mechanisms with a stepwise increasing complexity can be observed. For instance, users of Microsoft Word start often with very simple adaptations like modifying the button bar. A first step to functional adaptations can be seen in the recording of recurring tasks as macros and assigning them to shortcuts or buttons. More advanced users start to have a look at the generated macrocode and adapt it, without understanding the code, just by replacing parameters. They gain a deeper understanding of how the code works and start modifying the code more freely. Even more advanced users start writing code from scratch.

In absence of adaptation mechanisms, usable by end users, adaptations have to be delegated to IT experts, often leading to a long and costly adaptation process – as already discussed before. If the relation of time, cost and quality is influenced in a negative manner, because end users are being prevented from adapting systems on their own, an improvement of this relation can be achieved by increasing the end user adaptability of these systems. Steep slopes of complexity in the design principles limit the system's adaptability. Such slopes can be smoothed and flattened through the introduction of additional adaptation mechanisms (c.f. Figure 2). Assuming that a better relation of time, cost and quality is desirable for all potential customers of a software system, the realization of a gentle slope of complexity can be seen as a crucial goal for the development of information systems. This leads to a design environment offering different levels of complexity for different types of users. As a consequence, users having the skills for using a certain adaptation mechanism can learn and understand an advanced adaptation mechanism more easily. The additional complexity end users face is reduced through the smooth slope of the complexity graph. A prerequisite for this effect is that the design principles used in the advanced adaptation mechanisms build on, or strongly relate to, principles introduced in preceding adaptation mechanisms.

3 END USER DEVELOPMENT APPROACHES

End User Development as a term has evolved over time and complements a lot of older research fields denoted by other terms, while having a stronger focus on the development of new artefacts. In the end of the 1970s, David Smith started to change the process of programming by the implementation of the *Pygmalion* system. Inspired by the system, the research field of end user programming (EUP) emerged (Cypher, 1993). During the 1980s, the term "end user computing" (EUC) became popular (Brancheau & Brown, 1993). Later on the term "(end user) tailoring" (EUT) emerged from those research efforts. The term refers to the change of stable aspects of an artefact (Henderson & Kyng, 1991). In the fol-

lowing we describe also approaches from those fields (EUP, EUC and EUT), as we consider them to be specific predecessors of EUD. Some surveys of EUD approaches can be found in the literature: (Blackwell, 2005; Brancheau & Brown, 1993; Germonprez et al., 2007; Nardi, 1993). In comparison to those surveys, we present an updated overview, focusing on the conjunction of the different approaches, while using our own classification schema. We propose this schema could be used in combination with the later presented recommendations as orientation for IS designers, to make informed design decisions, when building embedded design environments for adaptable information systems. Table 1 displays our classification schema. We classified the EUD approaches by using the dimensions *complexity of design principle* and *adaptation power of design principle*. The dimensions comply with the following definitions.

Complexity of design principle: The complexity of a design principle is defined in accordance to the technical knowledge needed by an end user in order to apply it in a sensible way. Despite the existence of many classifications of end users in the literature, even much more recent ones, the classification of Nardi and Miller (1990) is appropriate in the context of this paper, as it takes the user's technical knowledge into account. Although end users represent a continuum of people with different technical skill levels, Nardi and Miller classify end users by discrete user groups as follows: *Non-programmers* have no or only little programming education and lack an intrinsic interest in computers. *Local developers* are domain experts and have usually a good knowledge of particular programs. *Programmers* have a good education in computers and therefore a broader technical knowledge than the other groups. The term "EUD approach" focuses at approaches suitable for the user group with the least technical knowledge. But if we categorize an EUD approach as an appropriate design principle for non-programmers, it is obvious, that local developers and programmers can use this approach as well.

Adaptation power of design principle: The second dimension of our schema is used to classify how much power an adaptation method has, defining which level or extent of adaptations can be realised by its usage. We based our classification on the classification of Mørch (1997), who developed the schema for tailoring approaches. The first level, *customisation*, contains approaches that are used to tailor the look and feel of applications. The second level, *integration*, contains all approaches, which allow changing the internal design of applications by using some kind of model. The third level, *extension*, contains all approaches, which allow changing the program code.

<table>
<tr><th colspan="4">EUD Approaches</th><th rowspan="2">Supportive EUD Approaches</th></tr>
<tr><th>Complexity / Adaptation Power</th><th>Customisation</th><th>Integration</th><th>Extension</th></tr>
<tr><td>Programmers</td><td></td><td></td><td>Programming</td><td rowspan="3">Testing:
Question-based testing;
WYSIWYT;
Integrity checks;
Exploration environments

Community aspects:
Configuration files

Appropriation support</td></tr>
<tr><td>Local Developers</td><td></td><td>Component swapping at runtime;
Separated tailoring interfaces</td><td>Natural Programming;
Scripting</td></tr>
<tr><td>Non-Programmers</td><td>Interface customisation;
Parameterisation</td><td>Programming by demonstration (PBD);
Accountants paradigm;
Integrated tailoring interfaces</td><td></td></tr>
</table>

Table 1: Classification of EUD approaches

We further differentiate, if an approach is a primary, self-contained EUD approach, which can be used directly for adaptations, or if an approach is a secondary supportive EUD approach, which is targeted at supporting other (primary) EUD approaches. The complexity and adaptation power of supportive EUD approaches must correspond to the supported (primary) EUD approach in order to be useful.

3.1 General Considerations

Before discussing the EUD approaches of Table 1 in detail, we want to focus briefly on approaches, which are more on a meta-level, but should be considered for the design of end user adaptable IS as well. The whole design process should follow an appropriate model, like the SER model (Seeding, Evolutionary Growth, Reseeding Model) of Fischer et al. (1994). SER is a process model for the incremental development of collaborative design environments, enabling users to design modifications of the current realization of the domain itself. Costabile et al. (2006) propose Software Shaping Workshops (SSWs) for including end users in the design process. SSWs have the objective of designing software environments that enable domain experts as end users to become co-designers of tools. The commonly used language, signs, notations and metaphors of user domains are very important for the design of EUD mechanisms, as those domains may differ herein to a great extent (e.g. consider the usage of an application in an educational environment and in a business environment). Therefore Sieckenius et al. (2006) focus on semiotic aspects (signs and signification, including processes of representation) of designing technology that meets the needs of users.

3.2 EUD Approaches

In this section we will discuss the EUD approaches we have found in the literature, starting at the level of customization for non-programmers (according to Table 1). One widespread and well-known type of EUD approaches is **interface customisation**. Those customisation mechanisms can be found in most of today's software systems. As an example, many modern applications offer users the possibility to apply themes or skins to change the look and feel or to adapt the used toolbars to change the directly accessible functionality. One of the older systems, which allowed interface customisations is the tailoring architecture *Buttons* of MacLean et al. (1990). "Buttons" provided basic modifications, like changing the placement of a button on the desktop or changing the label and icon of a button. As mentioned before, MacLean et al. recognized the importance of a "gentle slope of complexity" and therefore provided not only such basic adaptation functions, but supplemented the "Buttons" system with more advanced concepts that are described later on. A second very popular EUD approach is the **parameterisation** of an application, for instance by providing an options menu, enabling users to adjust certain settings according to their needs (e.g. by setting individual security options in a web browser). As an ongoing example, the "Buttons" system also provided the possibility of influencing a button's functionality by setting parameters. The two approaches, interface customisation and parameterisation are comparatively easy to use and allow the customisation of applications in a way that has been anticipated by its developers.

Programming by demonstration (PBD), also called Programming by Example (PBE), enables end users to demonstrate algorithms to the computer, just by using the computer like they are used to. The computer tracks and records all actions of the user and allows re-executing them. The "Buttons" system provided a macro recording function, and allowed the assignment of recorded actions to a button placeable anywhere on the desktop. Today, PBD mechanisms are used in many applications, like Microsoft Excel, where users are able to record actions as macros to automate recurring tasks. The book *Watch What I Do: Programming by Demonstration* (Cypher, 1993) provides a good overview of PBD approaches. Smith et al. (2001) state, that a basic problem of PDB approaches *"[…] has always been how to represent a recorded program to users. It's no good allowing users to create a program easily and then require them to learn a difficult syntactic language to view and modify it, as with most PBD systems."* The book *Your Wish Is My Command: Programming by Example* (Lieberman, 2001) contains many different examples, how research tried to overcome such criticism of PBD. Lieberman presents more flexible PBD approaches, being able to adjust re-execution by providing different input parameters. A "Goal Oriented Web Browser" (Faaborg & Lieberman, 2006), being a PDB system for the web that allows users to create general-purpose procedures just by giving a single example, demonstrates further improvement of flexibility. Another recent approach in this field is called Sloppy Programming and is based on interpretation of pseudo-natural language instructions, instead of formal

syntactic statements (Little et al., 2007), thus increasing adaptability of code for non-programmers. A powerful but fuzzy EUD approach related to improving the handling and cognitive perception of information is what we call the **accountants paradigm**. This paradigm utilizes the fact that many people understand without any precedent explanation a tabular representation of data. Spreadsheet applications utilise this paradigm as the main interaction model of the user interface. The paradigm is well understood by many people – not only accountants – because it was already used for centuries for calculations. Therefore inexperienced end users can "write programs" by using formulas that establish relations between different data cells easily (Nardi & Miller, 1990). The last EUD paradigm, for non-programmers, we describe on the level of integration is the **integrated tailoring interface**. Integrated tailoring interfaces integrate the design-time and runtime view of an application seamlessly. Spreadsheet applications are a smart way of integrating both views into one, as no formal modelling is required prior to the usage of a spreadsheet. The program logic of the spreadsheet can be modified at any time by changing or creating formulas, providing instantly visible results to the user, without the need to switch between a design-time and a runtime. Spreadsheet systems are one of the most popular applications in EUD research, as they use many different EUD approaches simultaneously (Lieberman et al., 2005; Nardi & Miller, 1990).

Leaving the group of non-programmers, we now focus on local developers. On the integration level there are two EUD approaches mentionable for local developers. The first one is **component swapping at runtime**. This approach is based on the ability of users of a component-based information system to change the composition of its components at runtime or add and remove components from the composition. System functionality composed from components needs to be visualised at the user interface as an understandable model for local developers. An example for a system using such an approach is the *FreEvolve* platform (Won et al., 2006). It serves as a tailoring environment that allows users to adapt their system during use time. It is based on an enhanced JavaBeans component model called FlexiBeans model, which particularly improves easy adaptation and exchange of components during run-time. More recent implementations of this design principle are *Mashup* tools like *Yahoo! Pipes*. In this case users compose applications by connecting services, which are available on the web. The visualisation of the functionality is usually realized by a second EUD approach, which can be described as **separated tailoring interfaces**. Separated tailoring have in contrast to integrated tailoring interfaces a separated design-time and runtime. As an example, FreEvolve provides a tailoring interface, where users can switch easily to the tailoring mode, apply adaptations and switch back to the use mode, where they can use their modifications instantly. The tailoring mode looks only slightly different than the use mode of the system. All tailorable components of the user interface can be moved or deleted and additional components can be dragged to the user interface. The control flow and data flow of the system can be changed by simply connecting data input ports and output ports of components with wires. Such adaptation possibilities provide a powerful way of enabling advanced adaptations of the system, even for users with little technical knowledge.

Besides those approaches, local developers could also use two more powerful concepts, **natural programming** and **scripting**, that are located at the extension level. Natural programming aims at creating programming languages and environments that are more "natural", which means they are closer to the way users think about their tasks, ideally enabling them to formally express their ideas in the same way as they think about them (Myers et al., 2004). Programming can be seen as the process of transforming a mental plan described by familiar terms into one compatible with a computer. The closer a language is to the one in which the original plan is expressed in, the easier this transformation process will be. Myers et al. studied the language and structure, which are used for problem solving, before users have been exposed to programming, to analyse which fundamental paradigms of computing are the most natural ones for users. Results reveal, that users often use event-based or rule-based structures or aggregate operators (acting on a set of objects all at once instead of iterating through a set). Boolean expressions are rarely used and are likely to be defined incorrectly. According to their results, Myers et al. developed the *HANDS* system for highly interactive graphical programs. HANDS uses an event-based language that features a new model for computation, provides queries and aggregate operators, has high visibility of program data, and domain-specific features for the creation of interactive

animations and simulations. The HANDS system has been developed for children, as one example of a group of beginners. As user studies show, ten-year-olds are able to learn the HANDS system during a three-hour session, indicating that the effort of developing a more natural design environment pays off in the sense of achieved complexity reduction. More common than natural programming approaches is to offer users a scripting language within an application for adaptation purposes. **Scripting** languages are less complex than programming languages (e.g. they do not require strong typing) but therefore also limited in their adaptation power (e.g. they just offer the usage of predefined objects instead of creating new ones). Spreadsheet applications offer in many cases the possibility to use scripting languages to extend the functionality or to use the spreadsheet within other applications. A popular example is the utilization of Visual Basic for Applications (VBA) to program small applications that use parts of Microsoft Excel.

Leaving the group of local developers and remaining at the level of extension, there is only one approach left, which is more sophisticated: **Programming**. There is not much to say about that, except that programming could be a good supplement at the top of the complexity slope, as it was demonstrated in the "Buttons" systems, where programmers could put Lisp code inside a button.

3.3 Supportive EUD Approaches

In this section we want to focus on supportive EUD approaches, which are helpful to support users in using the previously described approaches. The supportive approaches are separated in testing support, community support and appropriation support, with a focus on testing. **Question-based testing** is an approach allowing users to ask questions in order to debug their code. User studies concerning debugging revealed that approximately one third of the questions asked during debugging are "why did" questions, which assume the occurrence of an unexpected runtime action. About two thirds are "why didn't" questions, which assume absence of an expected runtime action (Myers et al., 2004). With regard to this, Myers et al. developed the *WhyLine* debugging approach prototyped in the *Alice UI*, which directly allows users to ask "why did" and "why didn't" questions. The studies of Myers et al. showed a significantly decreased debugging time, using the WhyLine approach in their example of debugging a Pac Man game. The metaphor What You See Is What You Test (**WYSIWYT**) was utilised by Burnett et al. (2004) to construct a testing system under the same name. The WYSIWYT system was build for spreadsheet applications, seamlessly integrated in their already tightly integrated design-time and runtime. Users can test spreadsheets incrementally by simply validating cell values as correct (given the current inputs) by checking off the cell. As different input values in certain cells might be needed to cause other dependencies between formulas, the difficult generation of suitable input variables needed to increase test coverage is supported by a Help-Me-Test button. User studies revealed that the use of the described debugging methodology leads to a significantly increased effectiveness in debugging spreadsheet formulas for inexperienced users. The utilisation of **integrity checks** helps users to verify created compositions by exposing semantic errors and helps them to gain a better understanding of composition correctness. Won (2004) describes three interactive integrity check approaches, which were designed for the FreEvolve platform: *constraint integrity*, *restricted solution integrity* and *event flow integrity*. Constraints refer to local properties of a component and lead to a violation of the integrity, if they are not met. The restricted solution integrity is based on this, but provides recommendations, how the integrity can be re-established. The event flow integrity analyses message flows within a composition to ensure that generated messages are consumed as well. If important messages are ignored, the integrity of the composition is violated. The evaluation of the integrity system showed an increased usefulness of tailoring interfaces, because inexperienced users were enabled to validate their compositions. The testing approach **exploration environments,** is an in-between approach between testing and community support. Exploration environments allow users of a system to experiment securely with its functionality and do adaptations, without damaging the system or influencing the work of other users (Wulf, 2000). Powerful variants of such environments could furthermore allow users to switch their user interface to the user interface of a different user, enabling them to explore the system from different angles. The evaluation of an exploration environ-

ment approach in a groupware system presented by Wulf proved that exploration environments allow individual users to better experiment with groupware functions, but seem to be best suited for those users who already have a certain skill level in handling computers.

Configuration files as a mechanism of storing and exchanging adaptations can be used as a EUD approach, which supports user communities to perform their adaptations collaboratively. Different technical skills and different positions within an organisation lead to collaborations. Several empirical studies confirmed that adaptation activities are typically carried out collaboratively (Mackay, 1990; Nardi, 1993). Within such communities, the different user groups benefit from each other. For example, technically more experienced or motivated users can help technically inexperienced or less motivated users to do adaptations. User groups can be supported by different technical mechanisms, like **integrated repositories** and **integrated mailing functions**. Both mechanisms allow users to exchange their adaptation configurations, either by storing them in an integrated repository, or by sending them via email to other users. The FreEvolve platform allows users to store their individual adaptations to the system in configuration files and submit these to a central repository, accessible by all users and thus enabling exchange of adaptations.

The last category is called **appropriation support**, which *"[…] covers all measures to support appropriation activities as creative and collaborative processes of user-user interaction to fit a technology into an application field."* (Pipek, 2005). It covers for instance articulation support, decision support, observation support and explanation support. Concrete tools in this context could be for example a forum system or a help system. For a detailed discussion of the topic we refer to Pipek (2005).

4 RECOMMENDATIONS FOR THE UTILIZATION OF END USER DEVELOPMENT APPROACHES

After the introduction of the EUD approaches, we will provide recommendations, how the design principles can be used conjointly, to develop embedded design environments for end users. By this we will contribute to the knowledge pool and support designers in their design decisions. The recommendations can only be fuzzy, because they do not consider the domain or context of the system. As they are based on our literature review, they may sound naïve from a practical perspective, but should rather serve as a general guideline for developers and show them, how the combination of EUD approaches can lead to a higher flexibility of information systems. So far we cannot support this thesis by empirical evidence, but will work on it in the future. Figure 3 depicts the structure of an embedded design environment, which has an idealized gentle slope of complexity. The division of the X-axis (*Adaptation Power of Design Principle*) and of the Y-axis (*Complexity of Design Principle*) follow our categorisation schema. In the center of the figure are different EUD design principles that are building upon each other to generate the idealized gentle slope of complexity of the system. It was not possible to identify an example of a combination of the previously presented EUD approaches, which would lead to such an optimal slope, which indicates that more research is necessary. Therefore we just named the design principles 1 to 6. The slope shows how users of the system could advance over time, starting as non-programmers doing customisations and ending as programmers implementing extensions to the system. The additional complexity that has to be handled in order to be able to use a more sophisticated design principle is limited due to the gentle slope, thus reducing learning efforts and facilitating the usage of advanced tools. The two grey rectangles at the two sides of the figure represent the supportive EUD approaches, which could be seen as helpful "equipment to climb the slope". We have stated in the general considerations section that there are EUD approaches which describe a user-centered design process. Starting from this, we recommend the *design of a set of building blocks for solutions within a participatory design process together with domain experts.* Users should be allowed to use these domain-related building blocks or components later on to develop or refine their individual system. The goal of this approach is not to build one unique solution for a single problem, but to develop components, which can be combined to create a variety of solutions for a domain.

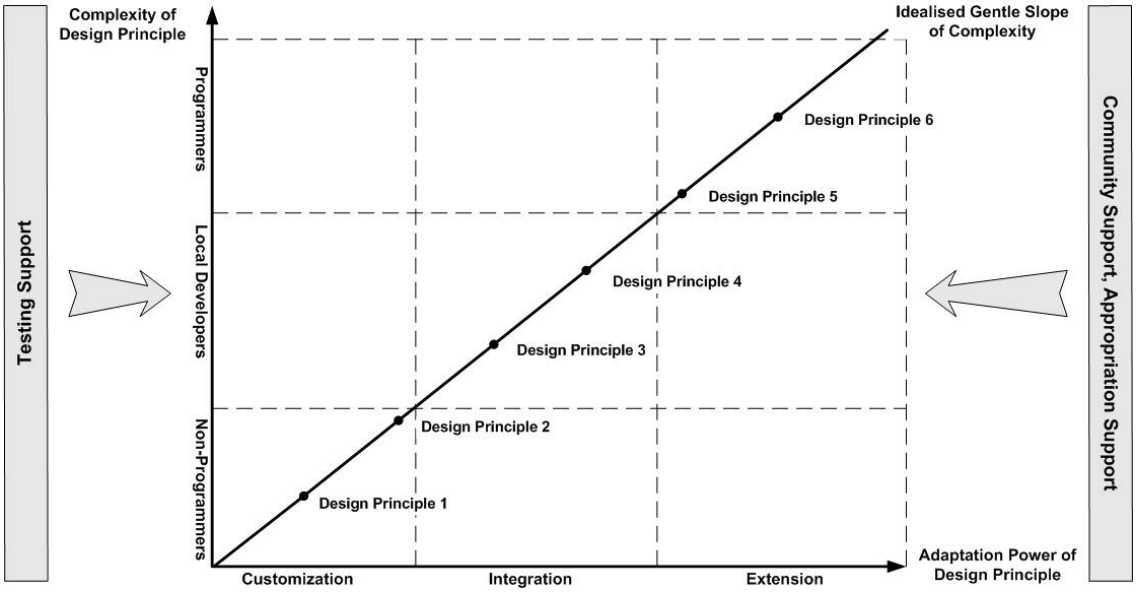

Figure 3: Embedded EUD Design Environment with an idealised gentle slope of complexity

The development of effective EUD mechanisms requires a high degree of understanding the end users and to know which concepts are familiar to them. We recommend *design environments to be as "natural" as possible for the addressed end user group and domain*, enabling users to formally express their ideas in the same way as they think about them. Design environments, using domain specific (visual) languages, which consider empirically identified semiotic aspects of the domain, are an example. The complexity reduction achievable by this design principle correlates with its relatedness to the domain and the "natural" problem solving strategies of users. The HANDS system developed by Myers et al. (2004) is an example of a design environment build upon the knowledge of what was "natural" for the addressed user group. As a result, using the system required a short training period and enabled users to achieve design results fast. As already stressed in the introduction, the development of EUD mechanisms following a gentle slope of complexity is a basic paradigm for the design of end user adaptable information systems. Avoiding steep slopes of complexity ensures that end users can learn the additional skills, needed to use an advanced design principle, with reasonable effort. Given the example of an IS offering reporting functionality, the IS should not only offer a design principle suitable for database experts. To flatten the complexity slope, a design principle for end users should be introduced as well. The system should enable users for example to create new reports, by selecting and combining predefined data building blocks of their domain in a simplified way. This could be complemented by an advanced design principle for developing building blocks. Continuing this chain of more and more advanced design principles may lead to a final design principle only suitable for experts, but users do not need to be experts to use the design principle at the beginning of the complexity slope. *Realizing a chain of design principles following a gentle slope of complexity can be seen as a crucial precondition for building end user adaptable information systems.* If users should be enabled to extend the functionality of a system, *we recommend programming by example as a powerful approach to avoid the need for formal modelling or traditional programming*. Recording the actions of users in the form of macro recording is a practical example for this. Allowing users not only to record and replay their actions, but also to modify them at a reasonable level of complexity, to extend or generalize their solution is an important option to unfold the real power of this approach. The use of programming by example can build an effective bridge to traditional programming or modelling as shown in the "Buttons" architecture (MacLean et al., 1990). More advanced approaches, helping users to generalize their recorded solutions and reuse them in other contexts by automatically changing relevant input parameters and addressed objects are in their infancy, but follow a promising idea and should be considered where applicable. Lieberman's goal-oriented web browser is a good example for such a system (Faaborg & Lieberman, 2006). Design principles, offered to end users, should avoid the

need to switch to a design environment, which follows a fundamentally different paradigm than the one at use time. If for example a user would modify a visual form, switching to the source code would not be an appropriate design option for an end user design environment. *We recommend designing the design time as similar to the runtime as possible*. In the form example, the design time could show the form exactly as in runtime, but allow modifications by interaction possibilities already known to the user, like drag and drop to add visual components from a component catalogue. *To be able to evaluate the success of design actions, results should be instantly visible and usable*. A good example for integrating design time and runtime are the previously stressed spreadsheet applications, which allow users to experience the results of their activity by directly manipulating and interacting with the artefact. If direct manipulation is not possible, the options to tailor a software artefact should be indicated consistently (Won et al., 2006). Spreadsheet applications are very successful in enabling even inexperienced users to get started with data processing. *We recommend the use of the "accountants paradigm" for design environments in a business context to get inexperienced users started quickly*. Basic reasons for that are the very simple sets of design rules a user has to follow as well as the long tradition of using spreadsheets for accounting tasks. Spreadsheets do not need to be modelled in advance in a formal way in a design process to be usable at the runtime. Although the spreadsheet paradigm is very simple, it cannot be easily transferred to fundamentally different design tasks than the one it was intentionally built for (calculation). But when thinking of the existing advanced solutions developed with spreadsheet applications or the basement of whole reporting solutions on spreadsheets, we encourage considering advanced usage scenarios for spreadsheets, to simplify the access for a broad user base.

Having a look at the supportive EUD approaches, we recommend designing the *debugging tools of the system as natural as possible*, making it easier for end users to understand how artefacts work. The WhyLine approach is a good example for this, because it allows users to formulate "Why" and "Why didn't" questions to find errors in the code. Furthermore the *verification or debugging mechanisms of the system should be integrated as close as possible into the runtime and design time environment* of the system, making it possible for users to find errors quickly, as realised in the WYSIWYT approach. This paradigm can also be found in professional software development environments, like the Eclipse platform, where programmers can already see error indicators in their source code editor (indicated as small red bars), without explicitly running a debugger. Sophisticated user support mechanisms could *provide end users with information on how detected errors could be corrected*, like in the restricted solution integrity approach. A helpful extension for testing is the *integration of an exploration environment into the system*. It allows end users to test and explore a modified application and allows users to learn how to use this system and what effects adaptations cause, without having to fear a damage of the system. Henderson and Kyng (1991) support this recommendation as they demand that systems should have an exploratory mode, in which users can experiment in a safe way. Adaptations are usually done by groups of different users as indicated by Nardi and Miller (1990). Therefore we recommend that *end user development tools support their community of users*. The tools should allow them to store and share their adaptations with other users, like in the case of the FreEvolve platform, where users are able to exchange component configurations that are stored in files. Additionally, the exchange of problem related information and help is an important support mechanism within a group of differently skilled users. Furthermore we recommend to *actively support the education of local developers*, like Gantt and Nardi proposed (Gantt & Nardi, 1992). Local developers often serve as translators between the users of a company and the programmers of an information system. This lowers potential misunderstandings between these groups, shortening the realization time and costs of adaptations, which can only be done by programmers.

5 CONCLUSIONS AND FURTHER RESEARCH

In this paper we provided an overview of the state-of-the-art in the field of End User Development, considering the different approaches from the perspective of information systems research. We classified the approaches by the two dimensions "*complexity of design principle"* and "*adaptation power of*

design principle" and derived general recommendations on how to make information systems more adaptable by end users. The recommendations are fuzzy and have to be selected and adapted in accordance to the demands of a specific domain and context. The combination of different recommendations allows the construction of information systems, which have a gentle slope of complexity. As users with different technical skills can adapt these systems with different mechanisms, the approach can lead to a higher flexibility of the whole system. Likewise, systems with a gentle slope of complexity allow users to advance over time and acquire more technical and adaptation knowledge, by providing an improved learning curve of available adaptation mechanisms. The construction of information systems, which are designed for a constant re-design by end users, have economical potential, because their overall costs (accumulated over the whole life-cycle) are lower, than the costs of comparable systems, which are not adaptable by end users. Re-designable systems are easier and faster to adapt to frequently changing requirements and can be used over a longer time period. Reduced efforts and costs coincide with increased effectiveness due to a better fit of the needed functionality and the functionality provided by the system. As organizational and technological development are closely correlated, the inability of users to adapt the technical systems used, is limiting the organizational development possibilities of organizations as well. Better adaptation mechanisms thus create better premises for being innovative and differentiate from competitors. Form a vendor's perspective the increased costs and effort in building such systems could pay off, as EUD approaches allow to offer less specialized products to a broader customer base by leaving the "last mile" of customization and development to a higher extent to the individual customers. For the development of constantly re-designable systems, it should be considered to integrate end user development as a part of the whole evolutionary, participative design cycle of a software system, as stated in (Wulf & Rohde, 1995). Although end user development approaches offer powerful methods for end users to adapt their information systems, there are some limitations of these approaches that should be kept in mind. One of the biggest problems of end user development was expressed by Repenning in an elegant way: *"End-user development environments cannot turn the intrinsically complex process of design into a simple one by employing clever interfaces no matter how intuitive they claim to be."* (Repenning & Ioannidou, 2006). Another mentionable issue is raised by Henderson and Kyng, stating that adapted systems are not uniform any more, making it hard for users to use the system of another person, because the system may look completely different (Henderson & Kyng, 1991). Other aspects comprise potentially harmful effects caused by errors in user-generated solutions, or a reduced execution speed/responsiveness of systems using a strongly layered architecture to include several adaptation mechanisms. As there are only few studies about such effects, more research is needed to investigate EUD approaches in real enterprise scenarios. To be able to support our recommendations empirically and investigate observable effects, we currently build an embedded design environment for enterprise systems, which will be evaluated in a real enterprise setup. Following the recommendations given in this paper, we are focusing on establishing a gentle slope of complexity and simplifying the adaptation processes for different types of users (mainly non-programmers and local developers).

6 ACKNOWLEDGEMENTS

We thank our reviewers for their valuable feedback and constructive suggestions. The research was funded by the German "Federal Ministry of Education and Research" (BMBF, project EUDISMES, number 01 IS E03 C).

References

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

Brooks, F. P. J. (1987). No silver bullet: essence and accidents of software engineering. Vol. 20 IEEE Computer Society Press, pp. 10-19.

Burnett, M., Cook, C. and Rothermel, G. (2004). End-user software engineering. Communications of the ACM, 47 (9), pp. 53-58.
Costabile, M. F., Fogli, D., Mussio, P. and Piccinno, A. (2006). End-User Development: The Software Shaping Workshop Approach. In End User Development, Springer, pp. 195- 217.
Cypher, A. (1993). Watch What I Do: Programming by Demonstration. MIT Press.
Faaborg, A. and Lieberman, H. (2006). A Goal-Oriented Web Browser. Proceedings of the CHI '06, ACM Press, pp. 51 - 760.
Fischer, G., McCall, R., Ostwald, J., Reeves, B. and Shipman, F. (1994). Seeding, evolutionary growth and reseeding: supporting the incremental development of design environments. Proceedings of the CHI '94, ACM Press, pp. 292 - 298.
Gantt, M. and Nardi, B. A. (1992). Gardeners and gurus: patterns of cooperation among CAD users. Proceedings of the CHI '92, ACM Press, pp. 107 - 117.
Henderson, A. and Kyng, M. (1991). There's no place like home: continuing design in use. In Design at Work: Cooperative Design of Computer Systems, Lawrence Erlbaum, pp. 219 - 240.
Lieberman, H. (2001). Your Wish Is My Command: Programming by Example. Morgan Kaufmann.
Lieberman, H., Paternò, F. and Wulf, V. (2006). End User Development. Springer, Dordrecht, The Netherlands.
Little, G., Lau, T. A., Cypher, A., Lin, J., Haber, E. M. and Kandogan, E. (2007). Koala: capture, share, automate, personalize business processes on the web. Proceedings of the CHI '07, ACM Press, pp. 943 - 946.
Mackay, W. E. (1990). Patterns of sharing customizable software. Proceedings of the CSCW '90, ACM Press, pp. 209 - 221.
MacLean, A., Carter, K., Lövstrand, L. and Moran, T. (1990). User-tailorable systems: pressing the issues with buttons. Proceedings of the CHI '90, ACM Press, pp. 175 - 182.
Mørch, A. (1997). Method and Tools for Tailoring of Object-oriented Applicatios: An Evolving Artifacts Approach. PhD Thesis, University of Oslo, Oslo.
Myers, B. A., Pane, J. F. and Ko, A. (2004). Natural programming languages and environments. Commun. ACM, 47 (9), pp. 47 - 52.
Nardi, B. A. (1993). A small matter of programming: perspectives on end user computing. MIT Press.
Nardi, B. A. and Miller, J. R. (1990). An ethnographic study of distributed problem solving in spreadsheet development. Proceedings of the CSCW '90, ACM Press, pp. 197 - 208.
Pipek, V. (2005). From Tailoring to Appropriation Support: Negotiating Groupware Usage. PhD thesis, Department of Information Processing Science, University of Oulu, Oulu.
Repenning, A. and Ioannidou, A. (2006). What makes End-User development Tick? 13 Design Guidelines. In End User Development, Springer.
Sieckenius, C., Souza, D. and Barbosa, S. D. J. (2006). A Semiotic Framing for End-User Development. In End User Development, Springer, pp. 405 - 431.
Smith, D. C., Cypher, A. and Tesler, L. (2001). Novice Programming Comes of Age. In Your Wish Is My Command: Programming By Example, Morgan Kaufmann.
Stiemerling, O., Kahler, H. and Wulf, V. (1997). How to make software softer - designing tailorable applications. Proceedings of the DIS '97, ACM Press.
Won, M. (2004). Interaktive Integritätsprüfung für komponentenbasierte Architekturen. PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät, Universität Bonn, Bonn, Germany, pp. 171.
Won, M., Stiemerling, O. and Wulf, V. (2006). Component-based Approaches to Tailorable Systems. In End User Development, Springer, pp. 127-153.
Wulf, V. (2000). Exploration Environments: Supportung Users to Learn Groupware Functions. Interacting with Computers, 13 (2), pp. 265-299.
Wulf, V. and Jarke, M. (2004). The economics of end-user development. Commun. ACM, 47 (9), pp. 41-42.
Wulf, V. and Rohde, M. (1995). Towards an integrated organization and technology development. Proceedings of the DIS '95, ACM Press, pp. 55 - 64.