# What can Model-Based UI Design offer to End-User Software Engineering?

**Anke Dittmar**[1], **Alfonso García Frey**[2], **Sophie Dupuy-Chessa**[3]
Dept. of Computer Science, University of Rostock[1], University of Grenoble[2], UJF[2], UPMF[3], LIG[2,3]
Germany[1], France[2,3]
Anke.Dittmar@uni-rostock.de, {Alfonso.Garcia-Frey, Sophie.Dupuy}@imag.fr

## ABSTRACT
End-User Programming enables end users to create their own programs. This can be accomplished in different ways, where one of them is by appropriation or reconfiguration of existing software. However, there is a trade-off between end users' 'situated design' and quality design which is addressed in End-User Software Engineering. This paper investigates how methods and techniques from Model-Based UI Design can contribute to End-User Software Engineering. Applying the concept of Extra-UI, the paper describes a Model-Based approach that allows to extend core applications in a way that some of the underlying models and assumptions become manipulable by end users. The approach is discussed through a running example.

## Author Keywords
End-User Software Engineering, End-User Programming, Model-Based UI design, Human-Computer Interaction.

## ACM Classification Keywords
H.5.2 Information Interfaces and Presentation: User Interfaces – theory and methods.: D.2.2.Software Engineering: Design Tools and Techniques – user interfaces.

## INTRODUCTION
The term 'End-User Programming' (EUP) has been developed in response to an existing gap between design and use of interactive applications. For a long time, 'designers' were seen as the experts in developing software artefacts. They acquired the right skills and followed the right processes to do so[1]. 'Users' were mainly seen as experts in their domains who have to learn to use the software that was built for them. Today, the number of people who do programming at work or in their leisure time is many times higher than the number of professional programmers as pointed out by [9] and others. Designing for end-user programming has supported this tendency and opened the design space for (end) users. However, "the problem with end-user programming is that end users' programs are all too often turning out to be of too low quality for the purposes for which they were created" [1]. There is a trade-off between 'situated design' and quality design. End-User Software Engineering addresses this problem [1].

[1]Obviously, this is an oversimplification. Software Engineering is still in its infancy.

In this paper, we investigate how methods and techniques that have been developed in Model-Based UI Design (MBD) can contribute to End-User Software Engineering (EUSE). MBD is an engineering approach that applies conceptual knowledge about the users' tasks and domains as well as ergonomic and technical knowledge about human-computer interaction to design user interfaces in a systematic way. In particular, we show how meta-models and the concept of Extra-UIs can help to put meta-design [6] into practice. In our approach, UI-designers become 'meta-designers' by adding Extra-UIs to their application under design. This offers design spaces for end users according to their main interests. They can explore possible uses of the user interface that are not necessarily intended by the designer.

The paper is organized as follows. Basic ideas and trends in End-User Software Engineering are first examined to position our own approach. The next two sections introduce MBD in more detail and illustrate one specific method and corresponding techniques through an example. This example application does not support end-user programming yet, but has to be enriched by an Extra-UI. How this can be done is shown in the following section to illustrate the overall idea of our approach that is explained at the beginning of that section. The paper concludes with a discussion of possible implications and future work.

## BACKGROUND I: END-USER PROGRAMMERS AND EUSE
We first need to understand the concept 'end-user programmer' in order to understand what EUSE is about. An indirect description may be given in the definition of End-User Development (EUD)[2] in [11] as 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, modify or extend a software artefact". According to [11], there are two types of EUP activities. Parameterization and customization allow the end user to choose between predefined options. Program creation and modification (e.g. by programming by example) allow them to change a program beyond the intentions of the 'professional developer'. In this paper, we are interested in supporting activities of the second type.

[2]EUSE and EUD are not distinguished in this paper.

A more precise view on the end-user programmer concept is given in [9]. Basically, it is seen as "a role and a state of mind". A person acts in this role if the manipulated program is not the primary outcome of their work but serves as a (often temporary) means to an end. "Traditional" software engineering assumes that developers are responsible for supplying quality products in terms of reliability, maintainability, usability etc., but that they often are not the users of their products. Personas and scenarios are well-known means for developers to understand and imagine contexts of use. In contrast, EUSE has to assume that developers modify a program for 'situated use', but may have less interest in or understanding of quality criteria and the role of abstraction in the application. Additionally, requirements and design activities are more intertwined in EUSE because end users tend to explore possible uses (appropriation). Two approaches to cope with these challenges are mentioned in [9]: dictating proper design practices, and injecting good design decisions into existing user practices. A technique that supports the latter approach, and that is also applied in this paper, is the combination of constraints and generation mechanisms. The end user can design or modify certain aspects only and the final code is generated automatically.

Above explanation reveals the weak boundaries between 'professional' programmers and end users and between related processes. Fischer et al. show how ideas of End-User Development extends the traditional notion of system development. In their meta-design approach they suggest to include users as active co-developers throughout the entire existence of the system. This includes periods of unplanned evolution and periods of deliberate (re)structuring and enhancement [6].

## BACKGROUND II: MODEL-BASED UI DESIGN

MBD is a software engineering approach that uses models capturing knowledge about different aspects of human-computer interaction as a basis for producing code of user interfaces (UI) in a structured way [12]. Typically, domain-dependent Concepts and Task models serve as starting point for producing Abstract UI models (AUI). From AUIs models Final UIs (FUI) are derived via an intermediate step - the Concrete UI models (CUI) [2]. Lower-level models are produced from higher-level descriptions by transformations. They can be performed by the application of transformation rules in an automated or semi-automated way or by the designer making explicit design decisions. Just to mention two examples: enabled task sets and heuristics are used in TERESA to transform task models automatically into AUI models [13]. A manual but tool-supported transformation of task and domain knowledge into dialog models is preferred in [3]. Transformations help to consider usability criteria during the design process [14, 8]. If they are applied on lower-level models to get higher-level ones they support re-engineering activities. Mappings are used to link elements from different models and, in some approaches, to keep track of the transformations from source to target elements. For example, each task of a task model and the concepts involved to achieve the task are mapped to a set of interactors in the CUI model in [14].

In MBD approaches, models can be re-used across different

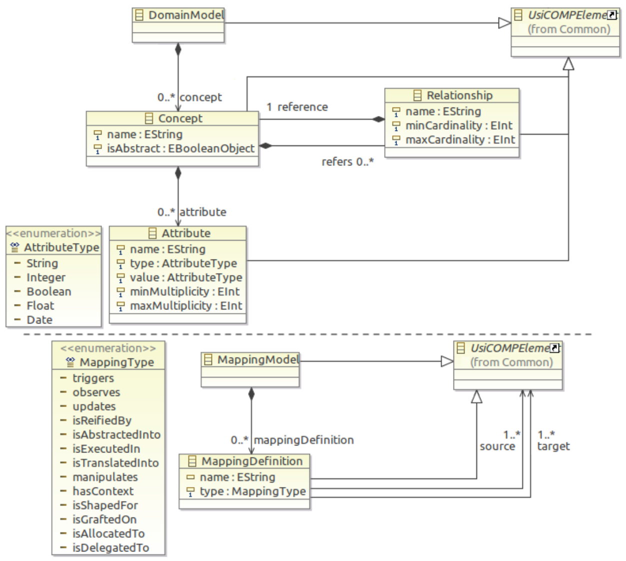


**Figure 1. Ecore representations of the Domain meta-model (top) and Mapping meta-model (bottom) that are applied in the case study.**

platforms and programming languages. Design decisions are made and described at conceptual levels. MBD also supports a more consistent development of UIs for different contexts [2]. However, there are still open questions concerning the link between UIs and functional cores in MBD. Limitations that arise from assumptions made in task descriptions are discussed e.g. in [4]. We will argue later that a view of UI designers as meta-designers and of (end) users as co-developers (as suggested in [6]) can change MBD practices and alleviate some of these limitations.

### Meta-Models in MBD

What makes MBD possible at all is the assumption that different task/domain/AUI/... models have common underlying structures that can be described by their corresponding meta-models. Figure 1 shows parts of ecore class diagrams of meta-models that will be used in the case study later on. Like in most MBD approaches, hierarchical task structures are assumed that include temporal constraints between sub-tasks. Domain models are described by concepts with attributes and relationships to other concepts. AUI/CUI models specify interactors in a platform independent/dependent way.

An obvious use of meta-models is in the specification of transformation rules to reason about models and to find appropriate mappings (see Code 1 for an example). Mappings themselves are often organized by a meta-model if they need to be maintained for traceability and for model changes at run time. This is required e.g. for UI plasticity [15] and, generally, in all approaches defining user interfaces on the basis of other UI models. In the suggested approach to support EUSE the concept of Extra-UIs is applied. An Extra-UI is a UI that represents and provides control over a UI [15]. Later in this paper, we will show how Extra-UIs can be used to allow users (re)configuring the application from a EUSE perspective.

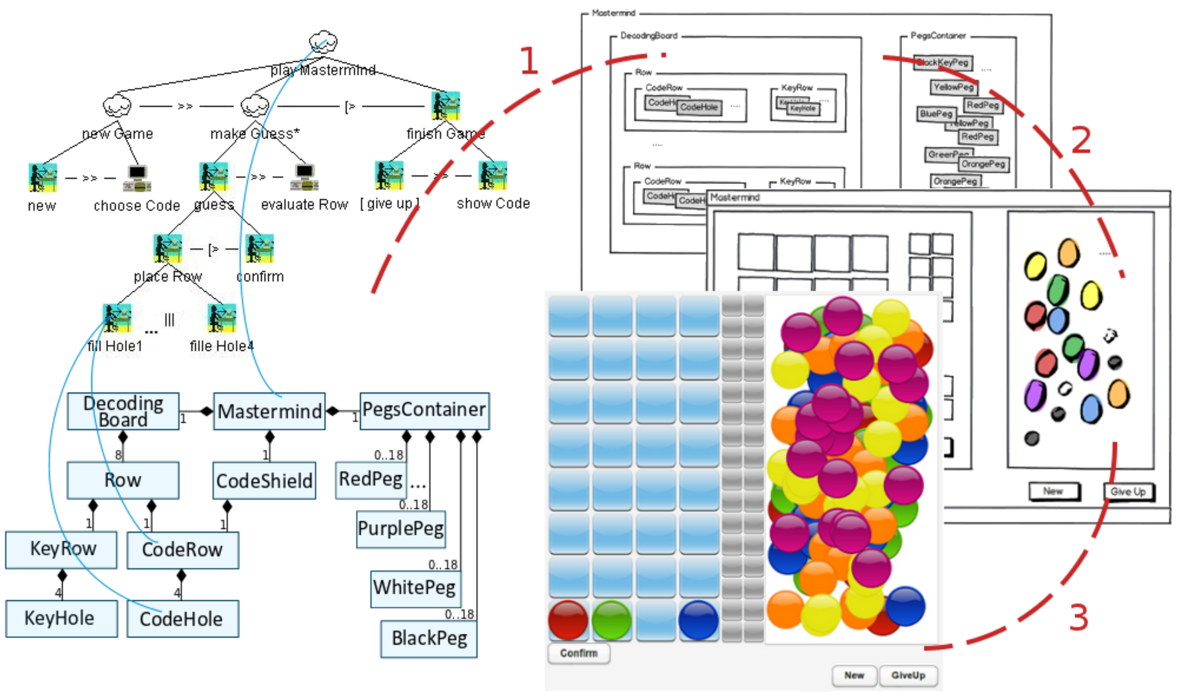


**Figure 2. Models in the mastermind example. Tasks in the task model (top left) are mapped (light, curved lines) to concepts from the Domain model (bottom left). The AUI -represented by containers (white background boxes) and units (grey background boxes)- is obtained by transformation of these models in 1. The CUI is produced from the AUI in 2 before the FUI is generated in 3.**

## INTRODUCTION OF THE CASE STUDY

This section demonstrates above mentioned ideas by illustrating the development of a user interface for the mastermind game[3]. The applied MBD approach is similar to [2]. The example was chosen for several reasons. First, it has a reasonable complexity and the domain is easy to understand. Second, its extension in the second part of the paper shows our contribution to EUSE. Third, it is useful to illustrate 'tinkering' activities of hobbyists who form one particular end-user group. They like to explore ways "to reconfigure and personalize technology with no definite end in mind" [9].

The example is designed by using the UsiComp tool [7]. The designer needs to create the following input models using the tool: the task model, the domain and the mapping model. The mapping model indicates what concepts from the domain are manipulated by each task. The AUI is automatically produced from these models by using transformation rules. An excerpt of one of these rules is shown in Code 1. The CUI is derived semi-automatically from the AUI, i.e., the designer specifies what predefined rules are applied to each element of the AUI model. Finally, a last automatic transformation generates the java code of the UI from the CUI model (see figure 2).

In the context of this paper, sketchy illustrations of models are often used for reasons of brevity and clarity. They focus on important aspects and avoid less well-known notations. They may also help to show the generality of the approach.

[3]`http://en.wikipedia.org/wiki/Mastermind_(board_game)`

```
rule Task2DataIU {
  from s : CTTE!Task (
        not thisModule.manipulates(s) and  -- Do not manipulate any Concepts
        s.compositions->size() = 0 and  -- Is leaf
        s.Category = #Interaction)  -- It is an Interactive Task
  to t : AUI!AbstractDataIU (
        name <- s.Name),
     m : Mapping!MappingDefinition (
        name <- s.Name,
        type <- #isReifiedBy,
        source <- s,
        target <- t)
}
```

**Code 1. Excerpt of an ATL transformation.**

## MODEL-BASED UI DESIGN FOR EUSE

We consider a user interface that allows for end-user programming (EUP UI) as consisting of two parts: a) UI of the actual application (core UI), and b) UI for modifications by end users that are not considered in the description of the core tasks. Both parts are developed by the designers within the model-based paradigm and require their own functional core. The latter part can be considered an Extra-UI because the separation makes possible that its design is informed by models that were developed during the design process of the core UI (core models). As a side effect, an opportunity is provided to reflect those core models and underlying modeling assumptions. This in turn is necessary to decide about the design space that the EUP UI should offer to end-user programmers. For example, the models in figure 2 describe a core UI that is designed to play mastermind with a four-pegs code. It would be reasonable to constrain EUP activities in such a way that

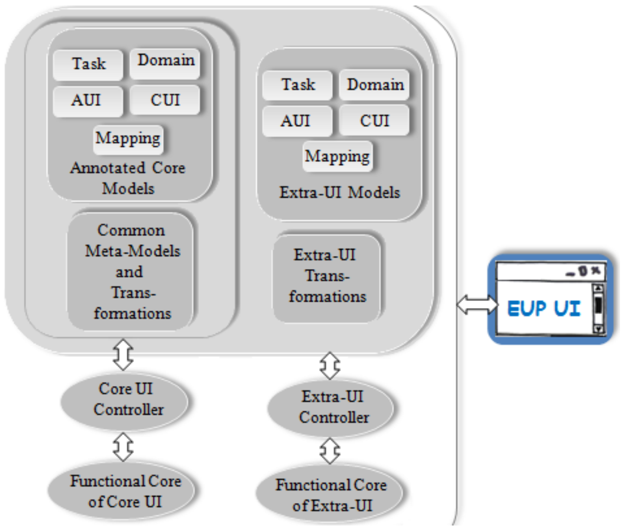


**Figure 3. Generic architecture for model-based EUP UI.**

the resulting UIs still include a board and some kinds of pegs. The functional core of the Extra UI implements the manipulation of core models at run time and the generation of modified core UIs that still preserve some original design decisions. In other words, an Extra UI opens a 'window' for the end user to underlying mechanisms and assumptions in the design. Figure 3 shows a generic architecture supporting these ideas. The coupling of elements of the core UI and the Extra-UI that is described later in this section (including figure 5) shows another perspective on how to develop the final EUP UI.

## Design Methodology

A four steps design methodology is suggested. It certainly needs to be integrated into a more holistic approach as it will be briefly discussed in the implications section.

1. Design of core UI,
2. Specification of design space constraints for end users,
3. Design of Extra-UI,
4. Coupling of core UI and Extra-UI to the final EUP UI.

In the first step, the UI of the core application is designed using a classical MBD approach as described above. Steps 2 and 3 deal with scope and complexity of the Extra-UI. People who act as end-user programmers are focused on their domain and have no primary interest in the models and techniques used in MBD. They should have the opportunity to change conceptual aspects and their representations in the UI within the scope of the domain, but without being confronted with all model details and notations in use (as e.g. in [15]). It is a design problem in itself to find a good representation of those UI parts end users can manipulate[4]. In the following, steps 2-4 will be explained in more detail. The example study (see figure 2) will be continued for illustration.

[4]A good description of the problem is given in [5]: "When creating means for users to modify their environment there is often a temptation to try to do everything - the spectre of Turing equivalence rises and before long a simple end-user customisation tool becomes a full-blown and complex programming language."

## Specification of design space constraints

In this step designers constrain the possible design space for end-user programmers. This is done by model annotation (but other approaches are possible). In the case study, model elements have two additional attributes *fixed:boolean* and *manipulable:boolean*. The first attribute serves to specify key elements in the core task and core domain model that cannot be deleted because they are considered as essential to the problem domain. The second attribute specifies elements that are not manipulable by the end user. Let us assume that in the core models of figure 2 tasks *playMastermind*, *makeGuess*, *startGame*, *finishGame*, *fillHole1* as well as concepts *Mastermind*, *CodeHole* and *RedPeg* are specified as fixed and their representations cannot be removed from the UI.

## Design of Extra-UI

The Extra-UI needs to be constructed following the same MBD approach as used for the core UI. Moreover, models from both UIs must conform to the same meta-models to allow for a coupling in the next step. Tasks in the task model of the Extra-UI describe possible manipulations of core models and need to be mapped to corresponding parts in the Extra-UI domain model (nothing is new). The domain model again consists of the annotated core models (including all mappings) that are represented according to the meta-models in use (see figure 1). As a consequence, the designer's view on what can be manipulated by the end user is strongly influenced by the meta-models. Of course, this is both a limitation and an advantage. Typical tasks the designer identifies may be restricted to the deletion, duplication, and renaming of tasks and concepts, the modification of their presentation in the UI and so on. On the other hand, meta-models make possible the description of design patterns for Extra-UIs.

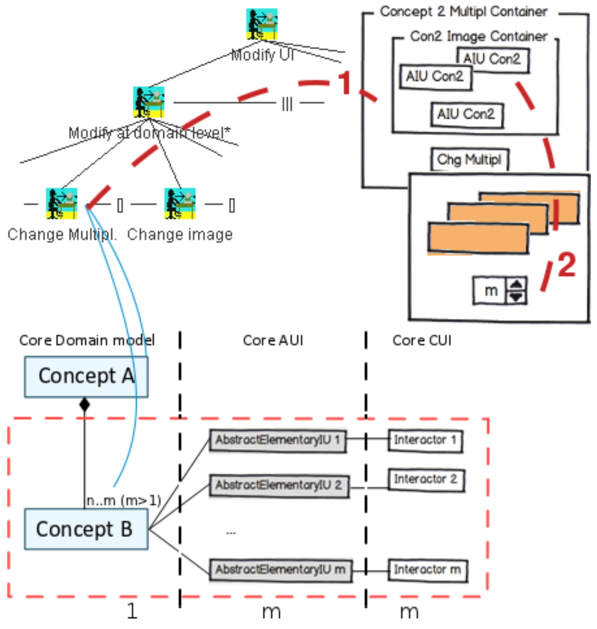


**Figure 4. A 1-m-m design pattern for Extra-UIs.**

Such patterns suggest representations of certain parts of annotated models in a core UI design that allow end users to modify these parts in a familiar notation. An important characteristic of design patterns for Extra-UIs is that they ensure consistency across core models. That means that the new core UI can be re-generated from modified core models. Figure 4 depicts a pattern that describes situations for applying the (Extra-UI) task *Change Multiplicity*. On the top of the left hand side, parts of the Extra-UI task model are to be seen. The bottom part of the figure describes in a generic way which parts of the core domain model, AUI and CUI model are involved. Corresponding mappings between task *Change Multiplicity* and domain concepts are indicated by light, curved lines as in figure 2. On the right hand side a transformation into an AUI and then CUI presentation for Extra-UIs is suggested that does not expect end users to be familiar with UML diagrams and the like.

### Coupling of core UI and Extra-UI

If the core and Extra-UI are designed they need to be coupled into the final application. Again, involved design models can be used to support a coupling at different levels of abstraction. Model composition is not in the focus of this paper, but it is discussed by many authors in MBD (e.g. [10]). Figure 5 indicates the coupling at the task and domain level that is used in our case study. Here, task models of the core UI and Extra-UI are integral parts of the overall task model.

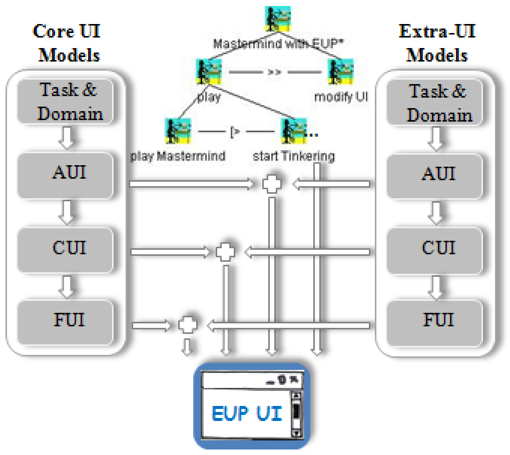


**Figure 5. A coupling of core UI and Extra-UI is possible at different levels of abstraction (here illustrated at the task & domain level).**

### Extra-UI for the mastermind game

Figure 6 shows an Extra-UI design for the running example. The above mentioned pattern was applied several times in transformation steps (partly recursively). Two other Extra-UI design patterns were used. First, a concept can be renamed within its final presentation if there is a 1:1:1 mapping between concept, AUI element, and CUI element. Additionally, there must be inverse rules for the transformation rules. In the example, the concept *Mastermind* can be renamed (see the text entry at the top container). Second, leaf tasks that are not fixed can be deleted. This includes the removal of all mappings to domain concepts and AUI elements. If it results in concepts with no mappings to tasks anymore the representations of the concepts are disabled in the Extra-UI and they are not represented in the modified core UI (similarly with AUI elements). In the screenshot, tasks *Evaluate Row* and *Give Up* are deleted which implies disabling the representation of the feedback part of the board and the "Give Up" button.

## FINALLY: USING THE EXAMPLE EUP UI (AND SHARE)

*One advantage of appropriation is the sense of ownership and empowerment it engenders. A sense of control is important for well being, and the act of tinkering gives this, whether to improve the user interface for its original purposes, or make it do something completely novel. [5]*

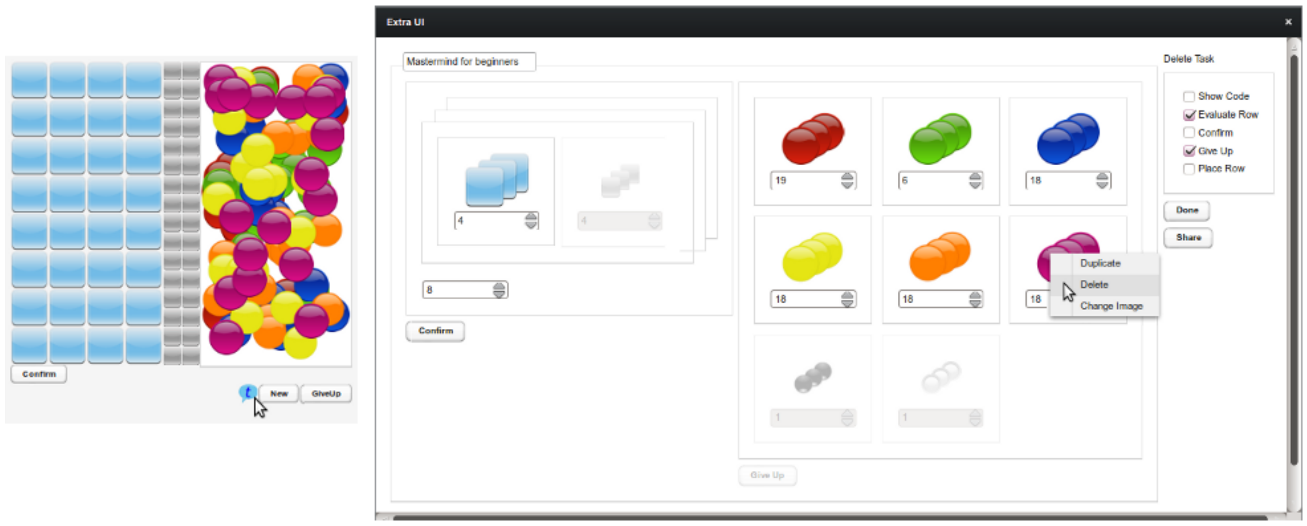
The provided Extra-UI lets end users produce different variations of the mastermind or even new peg based games. Please take a look at figure 6 and use your imagination to modify the original game on the left hand side. One possible example of a new game is a tic-tac-toe. A human-vs-human version of this game does not require the "confirm" button, responsible for linking the evaluation function from the functional core with the UI. For a human-vs-computer game, the "share" button must be used for asking the programmers to re-implement this evaluation function.

## IMPLICATIONS FOR MBD

Fischer et al. [6] encourage designers to conceptualize their activity as meta-design. In contrast to conventional design, meta-designers do not aim at the development of complete systems. Their task is to supply tools to users that empower them "to engage actively in the continuous development of systems rather than being restricted to the use of existing systems" [6]. Underdesign is suggested as a technique of meta-designers to create such design spaces for end users. The approach to MBD that is introduced in this paper supports the idea of underdesign. The separation between core UI and Extra-UI invites designers to reflect explicitly about their design assumptions that are incorporated in the core models. In a second design step, some of these assumptions are made modifiable by the user. The example illustrated that modifications of the core application by the end user can lead to new functional requirements. They need to be shared with the (meta-)designers to initiate extensions of the system, and possibly re-design steps. According to [4], MBD designers focus on the production of consistent and complete system specifications and, as a consequence, often consider task models as complete descriptions of the users' tasks. The proposed approach alleviates this problem by accepting design as an open and continuous co-operative process to find a better balance between 'too specific' and 'too universal' interactive applications.

## CONCLUSIONS AND FUTURE WORK

This paper investigates how methods and techniques from Model-Based UI Design can contribute to End-User Software Engineering through three main contributions. Firstly, the concept of Extra-UI is revisited from the EUP perspective. Secondly, a four steps methodology explains how to apply this concept to create an EUP UI. And finally, we discuss some interesting annotations on the core models that help to



- [ ] Show Code
- [x] Evaluate Row
- [ ] Confirm
- [x] Give Up
- [ ] Place Row

**Figure 6. An EUP UI for mastermind: the game was interrupted by pressing the "tinker" button (core UI on the left) to use the Extra-UI (on the right).**

decide about the design space for end users and some Extra-UI design patterns to support appropriate representations of this design space. The contributions are discussed and illustrated through a running example in which a mastermind game can be (re)programmed into a tic-tac-toe game. This example demonstrates the practicality of the method. Further work aims to make it more systematic. In addition, we want to consider sets of Extra-UIs for different groups of end-users who approach a system under design with different interests and background knowledge.

## ACKNOWLEDGMENTS

We are grateful to UPMF for their fincancial support to the collaboration between the authors. The second author is supported by the European ITEA UsiXML project, the third author by the ANR MOANO.

## REFERENCES

1. Burnett, M. What is end-user software engineering and why does it matter? In *Proc. of IS-EUD'09*, IS-EUD '09, Springer-Verlag (2009), 15–28.
2. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. A unifying reference framework for multi-target user interfaces. *Interacting with Computers 15*, 3 (2003), 289–308.
3. Dittmar, A., and Forbrig, P. The influence of improved task models on dialogues. In *Proc. of CADUI '04*, Kluwer (2004), 1–14.
4. Dittmar, A., and Forbrig, P. Task-based design revisited. In *Proc. of EICS '09*, ACM (2009), 111–116.
5. Dix, A. Opening the Box - Meta-level Interfaces Needs and Solutions. In *Proc. of Interfaces : SUI'11 Workshop at EICS'11* (2011).
6. Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., and Mehandjiev, N. Meta-design: a manifesto for end-user development. *Commun. ACM 47*, 9 (2004), 33–37.
7. García Frey, A., Ceret, E., Dupuy-Chessa, S., Calvary, G., and Gabillon, Y. UsiComp: an extensible model-driven composer. In *Proc. of EICS2012*, ACM Press (2012).
8. García Frey, A., Ceret, E., Dupuy-Chessa, S., and Calvary, G. C. QUIMERA: a quality metamodel to improve design rationale. In *Proc. of EICS2011*, ACM Press (2011), 265–270.
9. Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., and Wiedenbeck, S. The state of the art in end-user software engineering. *ACM Comput. Surv. 43*, 3 (2011), 21:1–21:44.
10. Lewandowski, A., Lepreux, S., and Bourguin, G. Tasks models merging for high-level component composition. In *Proc. of HCI'07*, Springer-Verlag (Berlin, Heidelberg, 2007), 1129–1138.
11. Lieberman, H., Paterno, F., and Wulf, V., Eds. *End-User Development*. Kluwer/ Springer, 2006.
12. Limbourg, Q., and Vanderdonckt, J. Addressing the mapping problem in user interface design with usixml. In *Proc. of TAMODIA '04* (2004), 155–163.
13. Paternò, F., and Santoro, C. One model, many interfaces. In *Proc. of CADUI '02*, Kluwer (2002), 143–154.
14. Sottet, J.-S., Calvary, G., Coutaz, J., and Favre, J.-M. A model-driven engineering approach for the usability of plastic user interfaces. In *Proc. of EIS '08*, Springer-Verlag (2008), 140–157.
15. Sottet, J.-S., Calvary, G., Favre, J.-M., and Coutaz, J. Megamodeling and metamodel-driven engineering for plastic user interfaces: Mega-ui. In *Human-Centered Software Engineering*. 2009, 173–200.