# Users need your models! Exploiting Design Models for Explanations

Alfonso García Frey
UJF, CNRS, LIG
41 rue des mathématiques,
38400 Saint Martin d'Hères, France
*Alfonso.Garcia-Frey@imag.fr*

Gaëlle Calvary
Grenoble INP, CNRS, LIG
41 rue des mathématiques,
38400 Saint Martin d'Hères, France
*Gaelle.Calvary@imag.fr*

Sophie Dupuy-Chessa
UPMF, CNRS, LIG
41 rue des mathématiques,
38400 Saint Martin d'Hères, France
*Sophie.Dupuy@imag.fr*

**End users can ask themselves about the User Interface (UI). Questions arise because users are not designers so both designers and users, have different perceptions of the same UI. Help Systems have naturally emerged to tackle this problem. Most of these Help Systems are predefined, so at design time designers need to anticipate the problems users may find at runtime, which limits the scope of the support. This paper explores Model-Driven Engineering to overcome this limitation: models created at design time are exploited at runtime for providing end users with explanations. Based on Norman's Theory of Action this paper introduces the *Gulf of Quality* as the distance between the models the designer creates at design time and the mental models the end user elaborates. This concept sets the basis of a Model-Driven method and a supporting architecture for computing explanations for the end user. The method deals uniformly with the UI of the help system and the UI of the application. They can be weaved or not, depending on the model transformations the designer selects. A software architecture is devised and implemented in a running IDE. The feasibility of the approach is shown through two use cases.**

*Self-Explanatory User Interfaces, Model-Driven Engineering, Gulf of Quality*

## 1. PROBLEM AND GOALS

One recurrent problem in interactive systems is that end users may require assistance while interacting with a User Interface (UI). As stated in Myers et al. (2006): *Modern applications such as Microsoft Word have many automatic features and hidden dependencies that are frequently helpful but can be mysterious to both novice and expert users*. Consequently, questions such as how to accomplish a task, where an option is or why a feature is not enabled, naturally arise from the interaction with the UI. Therefore, assistance is needed to overcome the obstacles the end users find in the interaction.

A classical approach to support end users is to provide them with predefined help systems such as FAQs, guides or precomputed tutorials. These solutions cover most of the general topics end users may find. However, the scope of these solutions is limited because predefined help systems rely on information considered at design time, making difficult to anticipate all the obstacles for different reasons. First, designers are not end users so designers have different perceptions of the same UI than end users have. Having a different perception entails encountering different problems. Second, as the end users' perception is mainly based on previous experience, different end users have different perceptions. Consequently, different end users will potentially find different obstacles. Designers cannot foresee all the different problems for all the potential end users. Third, plastic UIs (Calvary et al. (2003)), i.e. UIs able to dynamically adapt to the context of use (<environment, platform, user>), demand dynamic help systems as well because UIs may adapt themselves to unforeseen contexts. Developers cannot consider all the different contexts of use one by one.

One approach to overcome the limitation of predefined help systems is Model-Driven Engineering (MDE). See for instance (Hussmann et al. (2011)). Model-Driven UIs are able to explore the UI models at runtime, extracting the necessary information to support end-users. These kind of Model-Driven UIs with support facilities are also known as Self-Explanatory UIs (García Frey et al. (2010*a*)). Different models have been used in the literature for supporting purposes, such as Behavior models (Vermeulen et al. (2010)) or Tasks models (Pangoli and Paternó (1995), García Frey et al. (2010*b*)).



Proceedings of the BCS HCI 2012
People & Computers XXVI, Birmingham, UK

This paper unifies the extraction and exploitation of explanations from design models through three contributions. First, an extension of Norman's Theory of Action couples designers' models with end users' mental models under the same framework. This extension allows us to formulate the hypothesis that the models used by designers at design time are useful for supporting end users at runtime. Based on this hypothesis we propose a set of Model-Driven design principles for building help systems, which is the second contribution. The third contribution is a generic architecture supporting these principles, that has been implemented in a running Integrated Development Environment (IDE).

The first section of the paper explains the extension of Norman's Theory of Action through the concept of *Gulf of Quality*. A second section presents the Model-Driven design principles for help systems, along with a generic architecture supporting these principles. Next, we present an implementation of such architecture. Later, we show two use cases based on this implementation. Finally, we discuss the properties of these help systems before the section Conclusions and Perspectives.

## 2. GULF OF QUALITY

Inspired by the Isatine framework (López-Jaquero et al. (2008)), we reuse Norman's Theory of Action for defining the *Gulf of Quality*. Norman stated (Norman (1990)) that any action of the interaction between humans and computers consists of seven cyclic stages. These stages are categorized into two gulfs (figure 1) that designers must ideally overcome: the Gulf of Execution -getting from the intention to execution- and the Gulf of Evaluation -interpreting and evaluating the system response-. The Theory of Action relies on the hypothesis that end users elaborate mental models of the interactive systems, and that these models determine end users' behavior during the interaction. We extend the theory to explicitly consider design models (figure 1) with the two following working hypotheses.

### 2.1. Working Hypotheses

When the designer of a UI interacts with the interface as a normal user does, according to the Theory of Action he/she makes mental models that determine the interaction process. However, we claim that the designer's behavior is also determined by other models related to the design process.

**Hypothesis 1** *The UI design models influence the behavior of its designer while interacting with this UI.*

Examples of these models that may influence the designers's behavior are task models and design rationale, classically expressed using notations such as QOC (MacLean et al. (1991)) or DRL (Lee and Lai (1991)).

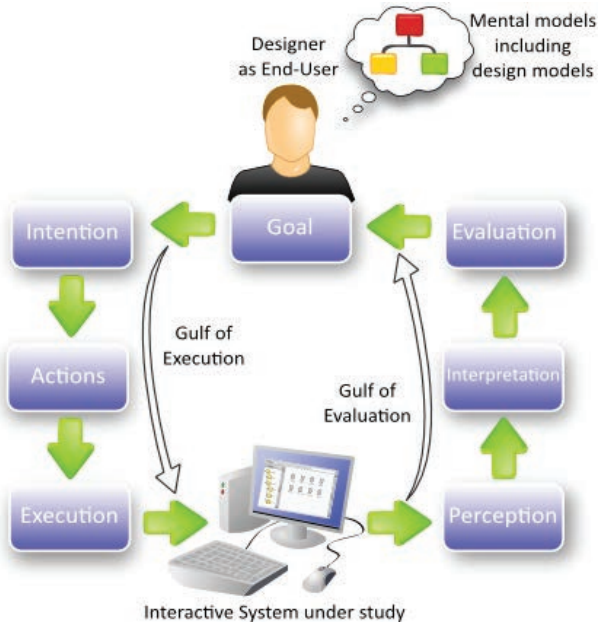


***Figure 1:*** *Hypothesis 1. Design models influence the designer's behavior in the interaction process.*

Because designers of an interactive system understand the system they design, their models are supposed to be more complete and accurate than end users' mental models. This fact explains why designers don't need the same support as end users and why they don't find the same problems in the interaction. Moreover, some works identify design models as being key for understanding the UI, for example Serna et al. (2010) stated that changing the platform of a UI leads to the reexamination of the initial designs. This fact leads us to a second hypothesis:

**Hypothesis 2** *Design models are suitable for supporting end users in the interaction process.*

The immediate consequence is that design models can enrich end users' support, so end users will better understand the UI, and therefore they'll have less problems while interacting. To directly take into account design models for supporting purposes, we introduce the concept of *Gulf of Quality*.

### 2.2. Definition

We define *Gulf of Quality in interaction* (or simply *Gulf of Quality*) as the distance between the design models the designers create at design time and the mental models the end users make at runtime while interacting with the system (figure 2).

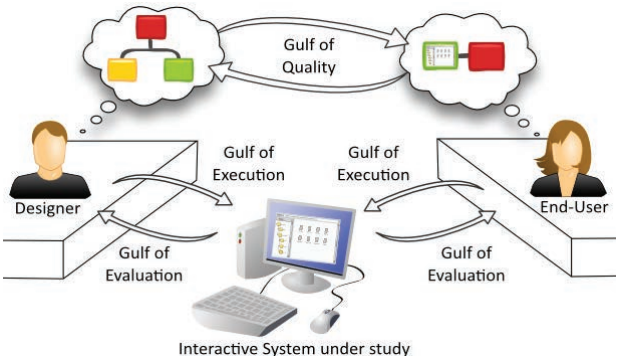


***Figure 2:*** *Gulf of Quality.*

Note how the term "design models" considered in the definition has a different sense as in the Theory of Action. Norman denotes "design models" to the designer's mental model (Norman and Draper (1986), page 47), while we explicitly consider the design models used to develop and produce the UI.

Model-Driven approaches are suited for reducing the *Gulf of Quality* because design models are explicitly defined by designers and a large effort has been put in MDE to keep models alive at runtime. Next section describes the design principles that are needed to reduce this gulf by dynamically extracting from design models the relevant information that is useful for supporting end users.

## 3. DESIGN PRINCIPLES AND SOFTWARE ARCHITECTURE

This section presents the necessary MDE concepts, the design principles described through a four steps methodology and a generic architecture with an implementation supporting these principles.

### 3.1. MDE for HCI in a nutshell

MDE of UIs consists of describing different features of UIs (e.g. tasks, domain, context of use) in models from which a final UI is produced. This paper is based on the Cameleon Reference Framework by Calvary et al. (2003) which promoted a MDE-compliant approach for developing UIs along four levels of abstraction: Tasks, Abstract User Interface (AUI), Concrete User Interface (CUI) and Final User Interface (FUI), being the FUI the source code of the generated UI. In a forward engineering approach the UI is obtained applying top-down transformations from tasks to code. Our approach can be used with any other model-driven approach and it supports the next functionality.

### 3.2. Help Systems Functionality

The help systems generated with our approach are responsible for:

- *Providing means for asking for support*. Designers must choose the way end users will ask for assistance so the system can understand the request. For instance, natural language dialogs or contextual help menus are valid for this purpose.
- *Computing the support the end user is asking for*. Once the question is understood by the system, its answer needs to be computed. For instance, if the end user asks how to configure the recto-verso printing option, one possible answer the help system can compute is the necessary steps the end user needs to do to access the dialog where this option is. Using our approach, the help system will query some design models to find the location of the recto-verso option and will compute the required steps that the end user needs to do to display it.
- *Presenting the computed support*. The computed answer must be provided to the end user in an understandable way. Natural language is a common option but designers can use any others with our approach, for instance, an animation of the mouse cursor that shows all the steps that are needed to configure the recto-verso option.

Next section describes the design principles for building Model-Driven help systems supporting this functionality.

### 3.3. Design principles

Design principles for explaining how to get end users' requests, how to extract explanations from design models according to these requests, and how to provide the extracted information back as support, are described through a four steps methodology:

#### *3.3.1. Building the UI of the application*

The UI of the application is built using classical UI models, obtaining the code at the end of the top-down transformation process (Application UI, figure 3). The methodology does not set any restrictions on what models are needed to generate the UI. For those applications having non Model-Driven UIs, reverse engineering techniques can be applied to obtain these models in a bottom-up transformation process from code to tasks (Limbourg et al. (2005)).

#### *3.3.2. Building the Help UI with classical UI models*

The UI of the Model-Driven help system needs to be constructed following the same model-based approach used for the application UI. Moreover, models from both UIs must conform to the same meta-models to allow a weaving of the two UIs in the next step.

#### *3.3.3. Adding support for computing help*

According to the second hypothesis, design models are suited for supporting end users. In this step

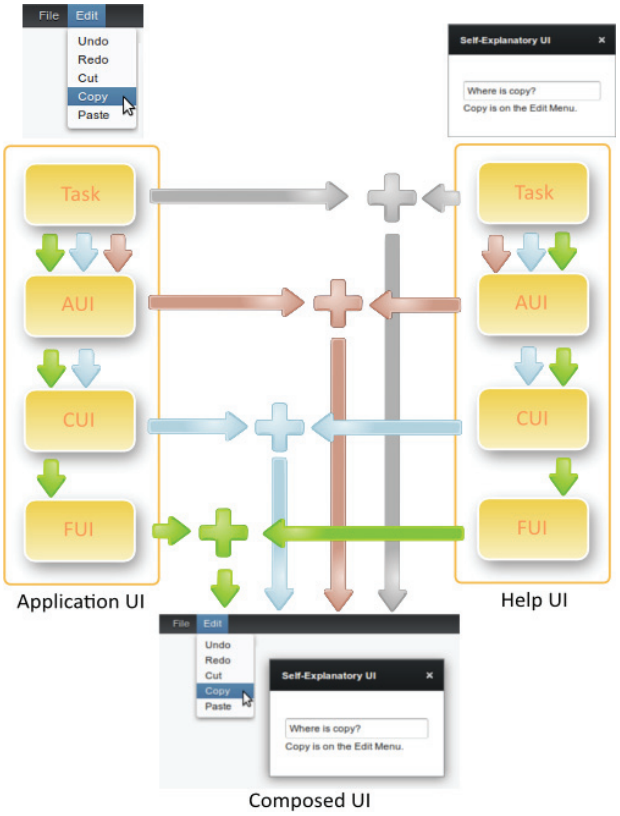


***Figure 3:*** *Possible combinations of weaving UIs.*

designers must add generic ways of computing answers from these models. Several works describe how to use specific models for this purpose. Vermeulen et al. (2010) present the PervasiveCrystal system, which benefits from a Behavior Model to answer why and why not questions in pervasive computing environments. This Behavior Model is based on the Event-Condition-Action (ECA) paradigm (Act-Net Consortium (1996)) and extended with inverse actions ($ECAA^{-1}$). Task models have been classically used for explanation purposes as shown by Pangoli and Paternó (1995). Design Rationale notations are also suited for explanations. For instance, the QOC notation -QOC for Question, Options and Criteria- which focuses on the discussion between different design alternatives, has been used by García Frey et al. (2011) for explaining why the UI is the way it is, using quality criteria as justification for design choices. The architecture presented later shows how to unify all these methods and how to use them together at runtime. Designers are free to exploit other models from other model-based approaches as the architecture does not set any restrictions about what models to use. Note that these approaches are based on generic answers, i.e. designers do not need to write all the possible answers for all the possible "why this happens?" questions, but only the mechanism that computes the answer from the underlying models.

In the case of our architecture and according to a MDE approach, this mechanism is based on transformations. The help system computes the answer by applying transformations on the models, independently of what model/s are implied or what is the question to be answered. These transformations query and extract the necessary information from the models, combining it if the information comes from different models, and converting it to something comprehensible by the end users.

#### *3.3.4. Weaving the UIs*

Designers can mix the help UI with the application UI at different levels. Models composition is discussed by many authors in MDE (e.g. by Lewandowski et al. (2007)). Its details are not the focus of this paper, but we briefly discuss some advantages and disadvantages of weaving the help UI with the application UI at different levels of abstraction (see figure 3).

Weaving at higher levels of abstraction implies a decrease in the total number of models. For instance, if we weave the task model of the help UI with the task model of the application UI, the composed UI can be obtained directly from the resulting task model of the weaving (transforming it to an AUI, CUI and FUI successively). Contrariwise, we can transform the task model of the help UI and the application UI independently to code (FUI) and make the weaving at this last level. In the first case, the transformations for computing the information from models become easier to manage as all the models have been unified so the help system only needs to query one task model instead of two, one AUI model instead of two, etc. The main disadvantage is that because the models have been weaved at the task level, the designer does not really know what elements of each model belongs to the help UI and what don't. One may need to make the distinction for many reasons such as for tuning the visual aspect of the help UI. In the case of weaving at the FUI level, the help models are completely separated from the application models, so customizing the UI of the help is easier. However, the transformations responsible for computing the help are more complex to manage because they need to query two tasks models, two AUIs and so on. This is mandatory as it guarantees the introspection property explained later in the paper.

Weavin at middle levels can be a good compromise depending on the requirements of the system (real-time re-generation of the UI, tuned help UI, ...)

### 3.4. A Model-Driven Generic Architecture

Figure 4 presents the architecture. The application UI is generated by transformation. The models,

meta-models and transformations implied in this generation are directly accessed by the *application controller*, in charge of this process. This controller also links the application logic from the functional core to the UI and vice versa. The help UI is generated by its respective *help controller*. However, this UI has its own models as seen in figure 3. These models conform to the same meta-models of the application UI. The same set of transformations are applied to derive both UIs. From the end user's point of view there is only one weaved UI. When the user requests support, the *help controller* receives the request and passes it to the *interpreter service* in charge of understanding it. This interpreter can be a natural language processor or even a gesture recognition system. The interpreter says to the *processor service* what support information needs to be computed. The *processor service* computes such information by accessing the models at runtime. Special help transformations are used for this purpose. The *processor service* can query all the models independently if they belong to the application or the help system, and using exactly the same *help transformations*. This is possible because all the models conform to the same meta-models. Once the information has been retrieved from models and computed by the *processor service*, it is prepared for the end user by the *generator service*. This service can update the UI with the desired information (so the user can use it) via the controller. The generator is also responsible for managing how the information is presented, for instance, in natural language or with an animation of the mouse cursor showing some procedure. Next section shows an implementation of the architecture.

### 3.5. UsiComp: an Implementation of the Generic Architecture

UsiComp (García Frey et al. (2012)) consists of a design module and a runtime module, both sharing common resources as meta-models and models (figure 5). The design module includes a visual editor (figure 6) for designing and prototyping purposes. It offers the following functionalities:

- Designers can manually define all the models and transformations needed to produce a UI (figure 6).
- Transformations of models are composed of rules represented by arrows. They are written in ATL[1]. Designers can select what rules they want to apply to a given model and the system will automatically compose and compile the resulting transformation.
- The resulting UI can be directly executed from the IDE (green play button) so designers can figure out what the generated UI looks like.

The runtime infrastructure works as follows:

- The Transformer Service (fig. 5) is a generic transformation service that can apply any transformation to any model or models, producing models or text (code) as output.
- To produce the UI, the Controller Service manages the transformations, their order of execution and

[1] `http://www.eclipse.org/atl/`

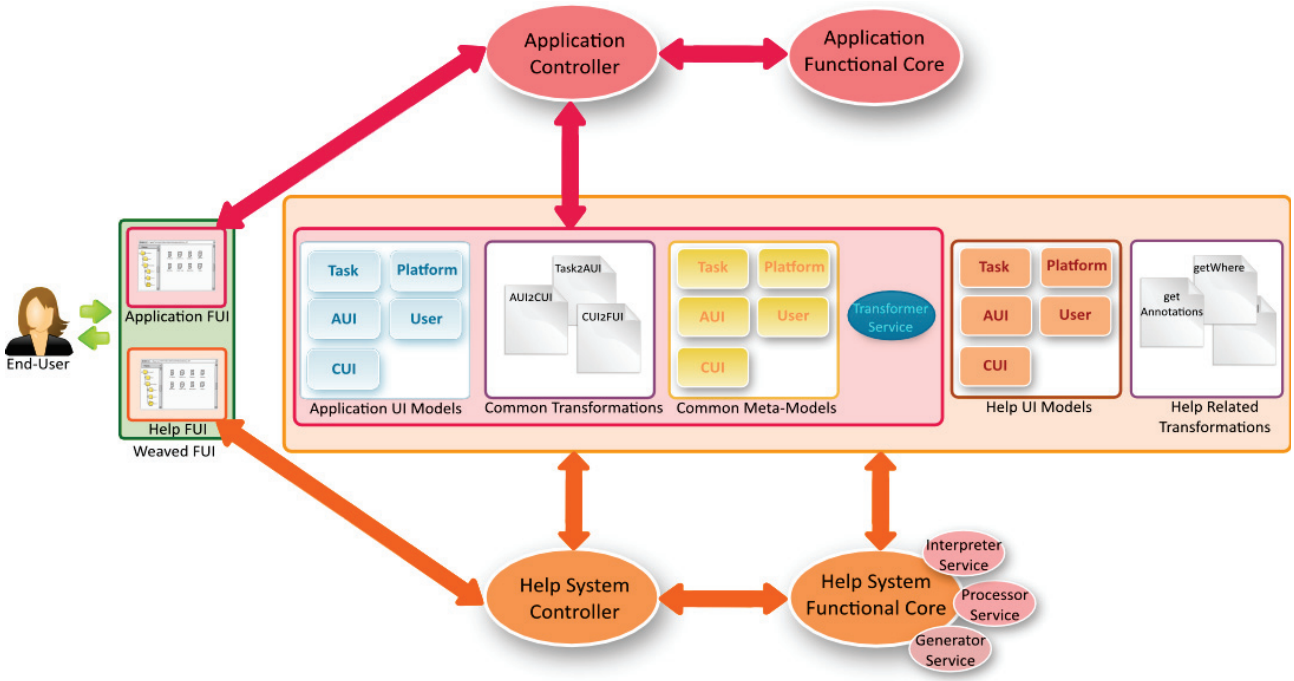


***Figure 4:*** *Generic architecture for Model-Driven help systems. The functional core of the help accesses any (meta-)model.*

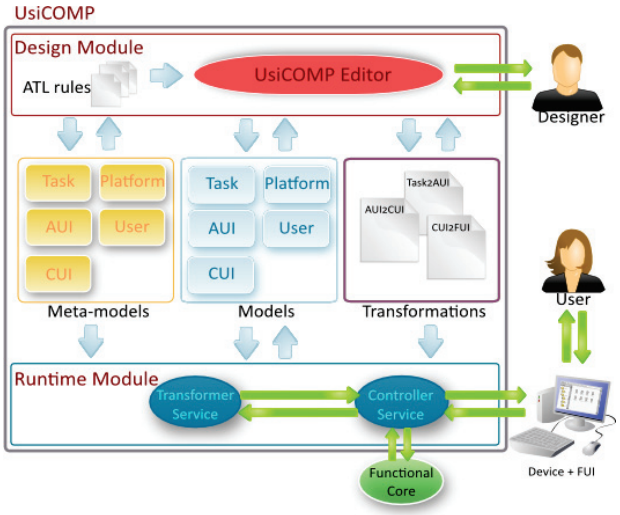


***Figure 5:*** *UsiComp architecture: IDE and runtime infrastructure.*

their related models and meta-models, calling the Transformer Service as many times as needed.

- In the transformation process, the Controller weaves the Functional Core of the application into the UI, embedding the calls from and to the UI.

UsiComp is implemented in Java, EMF[2] and OSGi services[3], which allows us to, for instance, change the way the help is computed without stopping the application.

## 4. USE CASES

The feasibility of the contribution is shown through two different use cases, both of them based on different help UIs that extract different explanations.

### 4.1. Use Case 1: Exploiting a CUI Model

This use case shows how to exploit a CUI model from which a RIA (Rich Internet Application) has been generated. This RIA has two classical desktop menus (figure 7) containing some options. Using the approach, we want to generate a help UI that makes it possible to the end-userto ask Where questions about the UI of the RIA. Where questions localize elements in the UI, so the end user can ask where an option is and the system will automatically retrieve the position in the interface, will compute the answer in natural language, and will return it to the end user using the UI of the help system. The approach has been implemented as follows:

*4.1.1. Building the UI of the application*
The UI of the RIA (background application in figure 7) is built using the models on figure 6 thanks to the

[2]http://www.eclipse.org/modeling/emf/
[3]http://www.osgi.org/

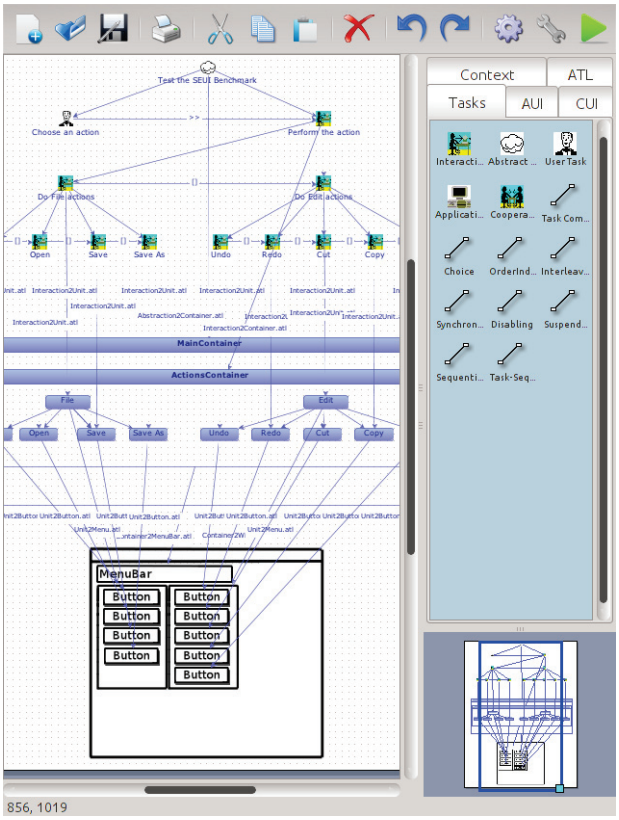


***Figure 6:*** *UsiComp editor with the models of the RIA application from the first use case. A task model in CTT notation is transformed into an AUI model (blue boxes), which is transformed into a graphical CUI (mockup).*

UsiComp editor. A task model defining the actions that the end user can perform on the RIA is modeled at first. It is then transformed into an AUI model, which is in turn transformed into a CUI model, that finally generates the final UI shown on figure 7.

*4.1.2. Building the Help UI with classical UI models*
The help UI is derived from the models illustrated on figure 8. The task model is composed of four main iterative tasks: *Think a question*, *Ask a question*, *Compute the answer* and *Provide the answer*. This kind of help UI is completely generic so it can be reused and weaved with any other different application and not only the RIA of this example.

*4.1.3. Adding support for computing help*
To answer Where questions, we use the CUI model to localize where an element of the UI is. An excerpt of the CUI meta-model used for computing the the right answer is shown in figure 9. The Where questions supported by the system are of the form:

"Where is + *label*?"

where label can be any label of any element (buttons, menus, windows, checkboxes, ...) of the UI, even those of the help UI (remember that the models of

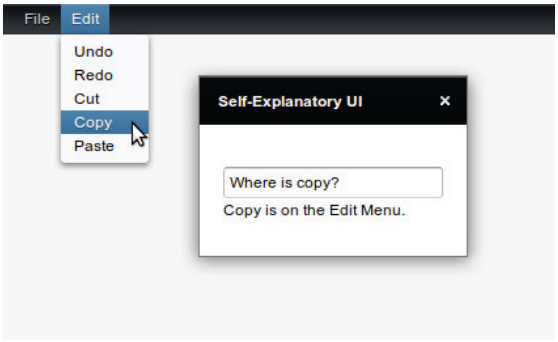


***Figure 7:*** *Example of a Model-Driven help UI weaved at the CUI level. Where questions are asked in a limited form of natural language.*

both UIs -RIA and help UI- conform to the same meta-models).

We have implemented the functional core of the help system according to the general architecture. The controller and the specific help services (interpreter, processor and generator) are built on OSGi. Every time the end user asks a Where question using the dialog, the *controller service* calls the interpreter passing the question as parameter. The *interpreter service* checks that the answer is well-formed, and passes the type of the question (*Where*) and the *label* of the element to be found to the *processor service*. The processor computes the answer launching the ATL transformation that processes Where questions with the parameter *label*. This ATL transformation accesses the CUI model at runtime looking for all the labels of the model (figure 9). If the parameter *label* matches the field "Text" (figure 9) of any label of the CUI model, it looks for its first visible parent. Visible parents can be buttons, menus, titles or any other widget of the UI. We skip invisible elements because we do not want to provide answers containing elements such as layouts, as end users are supposed to be unaware of such elements. Once a visible parent has been found, the transformation returns the name and type of the parent to the *processor service*, which calls in turn the *generator service* with these parameters. The *generator service* implements a very basic natural language composer. It formulates answers according to the following grammar:

"*Label* is on the *ParentName ParentType*"

where *Label* is the label to be found, *ParentName* is the name of the visible parent containing such label, and *ParentType* is the type of such container (Window, Menu, ...). Figure 7 shows an example.

*4.1.4. Weaving the UIs*

We choose to weave the help UI as a subwindow of the main UI of the application. For this, we manually

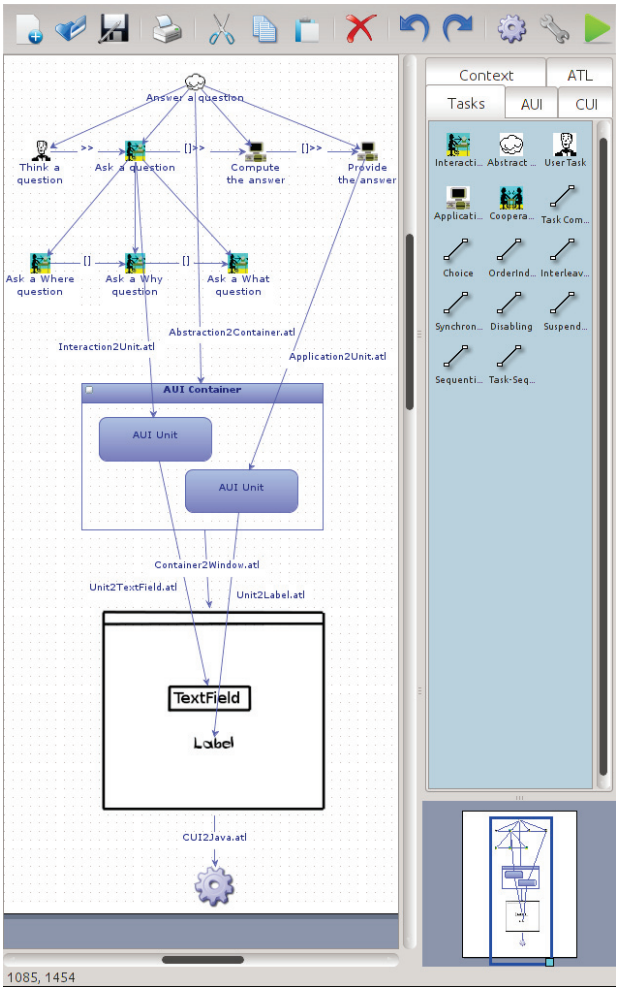


***Figure 8:*** *The IDE showing all the models of the Model-Driven help system. From top to bottom: task model, AUI model and CUI model.*

mix both UIs at the Window level thanks to UsiComp as shown in figure 10. We choose this level for weaving the UIs because the ATL transformation inspects the CUI models, and because if we have only one CUI model (result of the weaving) we need to launch the transformation only once. If we had weaved the UIs at the FUI level, we would have had two CUI models to inspect (application and help) so we would have needed to launch the transformation twice, once per model (the answer can be an element of the help UI as well).

This example illustrates how the design principles make it possible to use the CUI model to answer a specific kind of questions. Note how this generic approach lets designers to reuse the same help UI with other applications based on the same meta-models, just by weaving the UIs. Independently of the simplicity of the example, designers can add to their help UIs any of the approaches of the literature we have seen previously. Moreover, once one approach is added, it can be reused for different

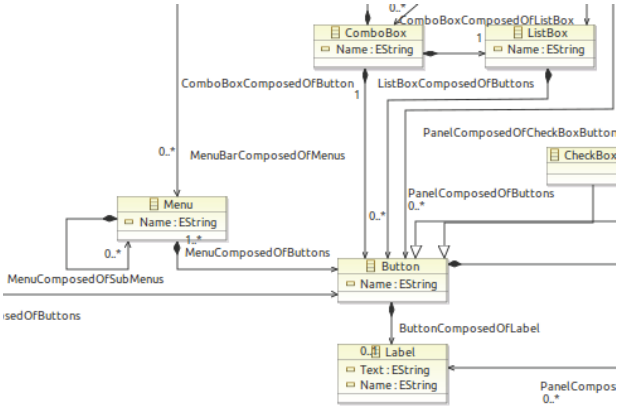


***Figure 9:*** *Excerpt of the CUI Meta-model in Ecore notation. Menu entries are modeled as buttons. The Menu element is the direct parent of all their menu entries.*

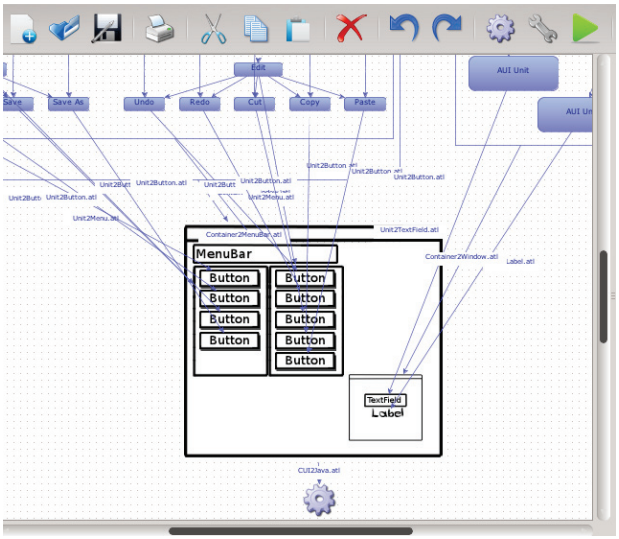


***Figure 10:*** *Manually weaving two CUI models coming from different AUIs.*

applications adapting the visual aspect of the help UI according to the circumstances.

### 4.2. Use Case 2: Exploiting Annotated Models

Figure 11 shows a new RIA application where the end user needs to fill in some information. The task model of such UI was annotated at design time by the designers, so each interactive task has a brief description that designers want now to reuse for supporting end users. Designers want to generate a UI for taking benefit of such annotations. The approach and architecture are used as follows:

#### *4.2.1. Building the UI of the application*

The Java code of this RIA is obtained with the same top-down transformation process from the task model to the code.

#### *4.2.2. Building the Help UI using classical UI models*

Designers make a task model for the help UI. They design only one interactive task called *Require description*. The motivation of this choice is to weave the help UI with the application UI at the task level.

#### *4.2.3. Adding support for computing help*

An annotated task model is used, so descriptions of the tasks will become available for the end user at runtime.

The *Require description* task is activated every time the end user presses the F1 key. The *controller service* has been now implemented with a listener for that key. When pressed, the controller passes the name of the widget having the focus directly to the *processor service* (the interpreter does not need to compute anything now). The processor launches the transformation responsible for retrieving the widget in the CUI model, looking for its associated task in the task model, and extracting its annotation. Once the processor gets the annotation containing the description of the task, it passes it to the *generator service*. The generator calls a new transformation to generate a new pop-up dialog that shows the description to the end user (figure 11).

#### *4.2.4. Weaving the UIs*

Using the UsiComp editor, the task *Require description* has been manually added to the root of the task model of the application so it will be available during all the life of the application.

This example shows how the transformations computing help deal with different models (CUI and task models). Approaches with any number of models are also possible.

## 5. PROPERTIES OF THIS APPROACH

Our method provides the generated help systems with the properties of Introspection, Flexibility, Distributability, Reusability and Customization.

### 5.1. Introspection

An introspective help system is able to provide support not only from the models coming from the application but also from its own models, for instance, to answer end users' questions about how to use the help system. This is possible because both UIs (application and help) are unified by construction as their models conform to the same meta-models, so the same transformation for extracting explanations can be applied. In the use case one, if the end user asks the question

"Where is Self-Explanatory UI?"


***Figure 11:*** *Second use case. Weaved help is triggered by pressing the F1 key. The help UI is generated by transformation.*

which is the title of the dialog, the system will automatically answer

"Self-Explanatory UI is on the RIA window"

as the direct parent of the dialog is the main window of the application, called RIA, which is an element of type "window" in the CUI model.

### 5.2. Flexibility for Weaving

The method provides different forms of flexibility regarding how the help UI is integrated into the target application. Help systems can be then *Weaved*, where the help UI and the application UI share the same space of interaction (figures 7 and 11), *Non-Weaved*, where the help UI runs in a different interaction space, or *Mixed*, where some of the options of the help UI are directly weaved into the application UI, and some others aren't. Non-weaved options can be directly accessible from the weaved ones if needed.

### 5.3. Distributability

Distributability is the property allowing a UI to run on multiple platforms. Distributing non-weaved UIs is specially easy because the models of the help UI are clearly separated from those of the application UI. This form of flexibility is specially useful for ubiquitous systems where not all the platforms are always available, or we want to require support without stopping other interaction processes. For instance, when playing a film on a laptop, the end user may want to ask about some options of the video player interface without stopping the film. The UI of the help system can be distributed to the smartphone for this purpose.

### 5.4. Reusability

Once designers know how to exploit a specific model for supporting purposes, they can easily apply the same procedure to the same kind of models of different applications. For instance, the use case two can benefit of the Where questions of the use case one, as the models of both use cases conform to the same meta-models. Designers can create their Model-Driven help systems once and reuse them everywhere.

### 5.5. Customization

The architecture allows to perform different customizations of the generated help UIs to fit specific application requirements. For instance, the look and feel of the application UI is normally fixed at the CUI level, using some mechanism based on stylesheets or skins stored in the CUI model. Designers would like to preserve the same look and feel for their help systems and applications. This can be accomplished by applying the same mechanism to the CUI model of the help UI. This is the technique we have implemented in the first use case (figure 7), where the same "stylesheet" has been applied to both CUI models. If the help UI is weaved before the CUI level the look and feel is automatically preserved as there is no specific CUI model for the help system (only one CUI model containing both UIs).

### 5.6. Open Architecture

The design principles presented in the paper do not set any restrictions on how designers let end users ask for support. No assumption is made about how the information supporting the end user is provided. There is no restriction on what models designers can use and how they can be exploited. For instance, the computation of the help in the functional core of the help system can be done with rule-based systems based on the application models, or on machine-learning algorithms.

## 6. CONCLUSIONS AND PERSPECTIVES

This paper presents three different contributions. First, the concept of *Gulf of Quality* is introduced based on Norman's Theory of Action. It couples the perception of designers and end users under the same framework, allowing to formulate the hypothesis that models used by designers at design time are useful for supporting end users at runtime. Second and based on this hypothesis, design principles for building Model-Driven help systems are described through a four steps approach. Third, a generic architecture supporting these principles is presented.

We have implemented this architecture in a prototype called UsiComp. We have evaluated the feasibility of the contribution along two use cases. As the presented principles do not rely on any particular model, it presents a means for unifying the different help systems of the literature under the framework of

the MDE of UIs. Our research agenda includes the generation of help UIs for real applications, as well as the evaluation of their usefulness.

## 7. ACKNOWLEDGMENTS

This work is funded by the European ITEA UsiXML project.

## REFERENCES

Act-Net Consortium, C. (1996), 'The active database management system manifesto: a rulebase of adbms features', *SIGMOD Rec.* **25**(3), 40–49.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L. and Vanderdonckt, J. (2003), 'A unifying reference framework for multi-target user interfaces', *Interacting With Computers Vol. 15/3* pp. 289–308.

García Frey, A., Calvary, G. and Dupuy-Chessa, S. (2010*a*), Self-explanatory user interfaces by model-driven engineering, *in* 'Proceedings of the CHI'10 Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI'10)', pp. 1–4.

García Frey, A., Calvary, G. and Dupuy-Chessa, S. (2010*b*), Xplain: an editor for building self-explanatory user interfaces by model-driven engineering, *in* 'Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems', EICS '10, ACM, New York, NY, USA, pp. 41–46.

García Frey, A., Céret, E., Dupuy-Chessa, S. and Calvary, G. (2011), Quimera: a quality metamodel to improve design rationale, *in* 'Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems', EICS '11, ACM, New York, NY, USA, pp. 265–270.

García Frey, A., Céret, E., Dupuy-Chessa, S., Calvary, G. and Gabillon, Y. (2012), Usicomp: an extensible model-driven composer, *in* 'Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems', EICS '12, ACM, New York, NY, USA, pp. 263–268.

Hussmann, H., Meixner, G. and Zuehlke, D. (2011), *Model-driven development of advanced user interfaces*, Springer, Berlin.

Lee, J. and Lai, K.-Y. (1991), 'What's in design rationale?', *Hum.-Comput. Interact.* **6**(3), 251–280.

Lewandowski, A., Lepreux, S. and Bourguin, G. (2007), Tasks models merging for high-level component composition, *in* 'Proc. of HCI'07', Springer-Verlag, Berlin, Heidelberg, pp. 1129–1138.

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L. and López-Jaquero, V. (2005), USIXML: a language supporting multi-path development of user interfaces, *in* R. Bastide, P. Palanque and J. Roth, eds, 'Engineering Human Computer Interaction and Interactive Systems', Vol. 3425 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 134–135. 10.1007/11431879_12.

López-Jaquero, V., Vanderdonckt, J., Montero, F. and González, P. (2008), Engineering interactive systems, Springer-Verlag, Berlin, Heidelberg, chapter Towards an Extended Model of User Interface Adaptation: The Isatine Framework, pp. 374–392.

MacLean, A., Young, R., Bellotti, V. and Moran, T. (1991), 'Questions, Options, and Criteria: Elements of Design Space Analysis', *Human-Computer Interaction* **6**(3), 201–250.

Myers, B. A., Weitzman, D. A., Ko, A. J. and Chau, D. H. (2006), Answering why and why not questions in user interfaces, *in* 'Proceedings of the SIGCHI conference on Human Factors in computing systems', CHI '06, ACM, New York, NY, USA, pp. 397–406.

Norman, D. (1990), *The design of everyday things*, New York: Doubleday.

Norman, D. A. and Draper, S. W. (1986), *User Centered System Design; New Perspectives on Human-Computer Interaction*, L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

Pangoli, S. and Paternó, F. (1995), Automatic generation of task-oriented help, *in* 'Proceedings of the 8th annual ACM symposium on User interface and software technology', UIST '95, ACM, New York, NY, USA, pp. 181–187.

Serna, A., Calvary, G. and Scapin, D. L. (2010), How assessing plasticity design choices can improve ui quality: a case study, *in* 'Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems', EICS '10, ACM, New York, NY, USA, pp. 29–34.

Vermeulen, J., Vanderhulst, G., Luyten, K. and Coninx, K. (2010), Pervasivecrystal: Asking and answering why and why not questions about pervasive computing applications, *in* 'Proceedings of the 2010 Sixth International Conference on Intelligent Environments', IE '10, IEEE Computer Society, Washington, DC, USA, pp. 271–276.