A Proposal for Modelling Usability in a Holistic MDD Method

Jose Ignacio Panach[1], Nathalie Aquino[2,3], Óscar Pastor[3]

[1]Escola Tècnica Superior d'Enginyeria, Departament d'Informàtica, Universitat de València
Avenida de la Universidad, s/n, 46100 Burjassot, Valencia, Spain
joigpana@uv.es

[2]Departamento de Electrónica e Informática
Universidad Católica "Nuestra Señora de la Asunción"
Tte. Lidio Cantalupi y Guillermo Molinas, Asunción, Paraguay

[3]Centro de Investigación en Métodos de Producción de Software - PROS
Universitat Politècnica de València
Camino de Vera s/n, 46022 Valencia, Spain
{naquino, opastor}@pros.upv.es

Abstract. Holistic methods for Model-Driven Development (MDD) aim to model all the system features in a conceptual model. This conceptual model is the input for a model compiler that can generate software systems by means of automatic transformations. However, in general, MDD methods focus on modelling the structure and functionality of systems, relegating the interaction and usability features to manual implementations at the last steps of the software development process. Some usability features are strongly related to the functionality of the system and their inclusion is not so easy. In order to facilitate the inclusion of functional usability features from the first steps of the development process and bring closer MDD methods to the holistic perspective, we propose a Usability Model. The Usability Model gathers conceptual primitives that represent functional usability features in a sufficiently abstract way so that the model can be used with different holistic MDD methods. This paper defines all the primitives that can be used to represent functional usability features. Moreover, we have defined a process to include the Usability Model in any MDD method without affecting its existing conceptual model. The proposal is based on model-to-model and model-to-code transformations. As proof of concept, we have applied our proposal to an existing MDD method called OO-Method and we have measured its efficiency.

Keywords: Model-driven development, usability, conceptual model.

1 Introduction

The Model-Driven Development (MDD) [10] approach proposes that analysts must focus all their efforts on building a conceptual model that represents all the system features, i.e., a holistic conceptual model. A conceptual model is used to represent the activity that elicits and describes the general knowledge that a particular information

system needs to know. In other words, a conceptual model is a way of viewing domains specifically [24]. In this context, "holistic conceptual model" means that the conceptual model is composed of complementary models that represent all the relevant system perspectives: structure, functionality and interaction. While existing MDD methods generally include models to represent system structure and functionality, the interaction modelling has usually received less attention. Our work focuses on modelling interaction adequately, giving to it the same level of importance as the structural and functional perspectives receive in order to provide a full system description in the conceptual model.

Each MDD method has its own conceptual model, which represents the system features through different models (for example, functional model, structural model, etc.). Each complementary model of a holistic MDD method must provide its own conceptual primitives. These conceptual primitives are modelling elements having the capability to represent a feature of the system in an abstract way. Examples of conceptual modelling elements for the structural model are classes of a class diagram, attributes, and services. Examples of conceptual modelling elements for the functional model include service pre/post conditions, valid states and transitions. In this paper, our intention is to focus on conceptual modelling elements for the interaction perspective, and in particular, on usability features. The holistic conceptual model, composed by conceptual primitives, can then be seen as the input for a model compiler that can generate the full software application automatically (or semi-automatically, depending on the model compiler capacity).

According to ISO 9126-1 [16], usability is a key issue to obtain a good acceptance from software users [8]. Some authors have divided usability recommendations into two groups [17]: recommendations that only affect the interface presentation (e.g., meaningfulness of a label), and recommendations that affect the system functionality (e.g., a cancel function). The second group of recommendations also receive the name of *functional usability features*. Dealing with these functional usability features is not very easy [17] since they affect not only the system interface, but also its structure and behaviour (architecture). For example, the feature Cancel aims to cancel the execution of a service. The implementation of this usability feature is not limited to the addition of a button in the interface; on the contrary, this feature also affects data persistence and functionality.

There are several authors in the software engineering community that have identified functional usability features and have proposed methods to include them in software development [13]. All these works propose including functional usability features from the early steps of the software development process, since they can involve many changes in the system architecture if they are considered at latter steps, when interfaces are designed. However, including usability features from the early steps has also some disadvantages:

- **Cost/benefit ratio**: The analyst must deal with usability throughout the entire development process, from the requirements capture until the implementation. This increases the analyst's effort and the cost/benefit ratio is not always favourable for features that are difficult to implement [17].
- **Changeable requirements**: Usability requirements (like other system requirements) are continuously evolving [18] and the adaptation to new requirements can involve a lot of rework in the system architecture.

- **Dependency on the implementation language**: The architecture design depends on the language used in the implementation and on the target platform.

Our research work is based on the idea that MDD methods avoid all these disadvantages [31][30]. There are currently several MDD methods that are able to model fully functional systems, such as WebRatio [1], AndroMDA [2], NDT [11] and OO-Method [27], among others. However, MDD methods are not able to model most of the functional usability features in their conceptual models. They solve all the mentioned disadvantages only for functional requirements, relegating usability features to a manual implementation in the code. This contradicts the MDD paradigm, which states that all the efforts of the analyst must be focused on models, relegating the code generation to transformation engines. Moreover, manual changes in the code can have an undesirable collateral impact: the model and the code can contradict each other. If the analyst must modify the code to include usability features (or any other feature), we cannot ensure that the modified code is a result of the model. This clearly contradicts the MDD paradigm, which states that the code must be an accurate reflection of the solution expressed within a conceptual model.

This paper aims to extend existing MDD methods with conceptual primitives that represent functional usability features, which are well known in the human-computer interaction community [17]. All these primitives are gathered in what we call the Usability Model. This proposal is a step forward to obtain holistic MDD methods, which can model not only structure or functionality, but also usability features. Our approach is valid for any MDD method but, as proof of concept, we have applied the approach to a specific MDD method called OO-Method [27]. In this proof, we have compared the effort to include usability features through a conceptual model with the effort to include these features manually. Results show that analyst's efficiency improves satisfactorily when the MDD method supports the modelling of usability features.

A preliminary version of this work can be found in [25]. There are two main contributions of this paper with regard to the previous one: (1) This paper discusses how to work with a composition of several functional usability features; (2) This paper shows an exploratory evaluation of the proposal that measures the effort employed by the analyst when working with the Usability Model. Moreover, we compare this effort with the effort employed when working with a manual implementation of functional usability features.

The remainder of this paper is structured as follows. Section 2 describes the functional usability features used in our proposal. Section 3 explains our proposed Usability Model and its meta-model. Section 4 describes how to include the Usability Model in a holistic MDD method, using OO-Method as an example. Section 5 evaluates the analyst's effort to apply the proposal. Section 6 presents the state of the art comparing existing proposals with our work. Finally, section 7 presents the conclusion and future work.

2 Background: Properties of Functional Usability Features

This work focuses on usability features that are strongly related to the functionality of a software system. Cancel, undo, and feedback facilities are examples of these usabil-

ity features. If this kind of usability features are not considered from the early steps of the software development process, there is a high risk of reworking to incorporate them later [4][13], and this is the reason for our choice. For example, the implementation of an undo facility involves a complex business logic, apart from the button that is used to display the functionality to the user.

Each usability feature can be defined as a set of properties that the system must support in order to implement it. In a previous work [26], we have extracted the properties that are needed to configure a set of usability features defined by Juristo [17] and called FUFs (Functional Usability Features). From all the sets of usability features that can be found in the literature, we chose FUFs because it is a set specific for management information systems, which are the target systems of our work. Moreover, FUFs have templates to capture usability requirements that are very useful for identifying the properties of the features. In the FUFs definition, each FUF has a main objective that can be specialized into more detailed goals called mechanisms. From each usability mechanism, we derived different Use Ways, and each Use Way was configured with a set of properties. These derivations and configurations were achieved carrying out the following steps [26]:

1. **Identify Use Ways:** Each usability mechanism can achieve its goal through different means. We call each such mean Use Way. For example, the FUF *User Input Error Prevention* aims to prevent users from making a mistake in the process of providing data to the system. This FUF is decomposed into one usability mechanism called *Structured Text Entry*, which helps the user to provide data with a specific structure. From the usability mechanism definition, we identified that this goal can be achieved in at least three Use Ways: (1) *Specify the input widget visualization type for enumerated values* (UW_STE1), which specifies the type of the input widget that better helps the user to insert information with a specific format; (2) *Mask definition* (UW_STE2), which prevents the user from entering data in an invalid format; (3) *Default values* (UW_STE3), which provides the user with guidance on which format to use to enter data.
2. **Identify Properties for each Use Way:** We call Properties to the different configuration options of Use Ways needed to satisfy usability requirements. For example, we identified that for *Specify the input widget visualization type for enumerated values* (UW_STE1) there are two Properties: (1) *Input field selection* and (2) *Type of input widget.* By means of *Input field selection* the analyst can specify which input fields she/he aims to customize; by means of *Type of input widget* the analyst can define which type of widget will be displayed to the user. With regard to *Mask definition* (UW_STE2), we identified that it is composed of two Properties: (1) *Input field selection* and (2) *Regular expression*. By means of *Input field selection* the analyst can specify the input fields that need a mask; by means of *Regular expression* the analyst specifies the regular expression that defines the mask. With regard to *Default values* (UW_STE3), we identified that it is composed of two Properties: (1) *Input field selection* and (2) *Definition of the default value*. By means of *Input field selection* the analyst can specify the input fields that need a default value; by means of *Definition of the default value* the analyst defines a default value for the widget.

Figure 1 shows a diagram that illustrates how FUFs are decomposed into usability mechanisms, usability mechanisms are decomposed into Use Ways, and Use Ways are decomposed into Properties. The definition of the elements displayed with grey background is not a contribution of the authors of this work (they were defined by Juristo [17]). The definition of the elements displayed with white background is a contribution of the authors, but it is not a contribution of this current work. More details about how the elements displayed with white background were defined can be found in [26].

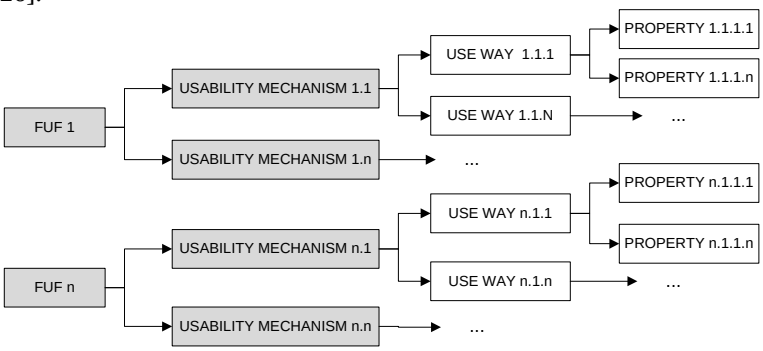


Fig. 1. An example of hierarchy among all the elements that compose the proposal

FUF	Usability Mechanism	Use of Way	Property
User input error prevention	Structured text entry	Specify the input widget visualization type for enumerated values (UW_STE1)	Input field selection
			Type of input widget
		Mask definition (UW_STE2)	Input field selection
			Regular expression
		Default values (UW_STE3)	Input field selection
			Default expression
Wizard	Step by step	Define a wizard (UW_WD)	Service selection
			Division into steps
			Steps description
			Steps execution flow

Table 1. Summary of Use Ways and their Properties

We applied this two-steps process to every usability mechanism that composes the list of FUFs [26]. Table 1 shows a summary of FUFs, usability mechanisms, Use Ways, and Properties used in this paper. Note that there is a total of 22 Use Ways whose description is out of scope of the paper. A detailed explanation of the Use Ways with all their Properties that resulted from the run of the process can be consulted in [20].

In [26], apart from the derivation of Use Ways and Properties, we defined a method that includes Use Ways and Properties in a MDD method. However, that proposal was specific for a MDD method, since it consisted in enriching an existing conceptual model with more primitives to support the Properties. This paper is a step forward of the previous proposal. In this work, we aim to define a generic approach that does not affect an existing conceptual model. This new approach is based on a Usability Model, such as the next section explains.

3 Representing Use Ways and their Properties in a Model: Usability Model

First of all we show a summary of our approach to deal with functional usability features in a MDD method. The approach is based on a Usability Model where we can represent by means of conceptual primitives every property of the Use Ways. We aim to define a Usability Model compliant for every MDD method. For this aim, we need a process to include the Usability Model in any existing MDD. This is performed thanks to model-to-model transformations (ATL rules [3]) and model-to-code transformations (Xpand rules [38]). Next sections describe the primitives that compose the Usability Model and how to include the Usability Model in any MDD method without affecting its existing conceptual model.

3.1 Conceptual Primitives of the Usability Model

The main goal of our work is to demonstrate that Properties of Use Ways can be included in the models that are used in MDD approaches, what is often just ignored. We need thus to incorporate a set of conceptual primitives that represent Use Ways and their Properties in a model we call Usability Model. Since currently there is not a standard notation to represent usability features, we have used a notation very similar to UML, which is broadly used in the software engineering community [36]. We use graphical elements already defined in UML and we have extended these elements with textual descriptions and new graphical elements.

The primitives that compose the Usability Model are grouped into two levels: **packages** and **elemental primitives**. Packages are primitives that contain a set of other primitives (packages or elemental primitives). Elemental primitives constitute the building blocks from which packages are composed. There are two types of packages in our Usability Model: Use Ways and interfaces. These packages are defined with the following procedure:

- First, for each **Use Way**, the analyst must define a package to group all the primitives that define the Use Way. Each Use Way is represented by means of an element similar to a UML package whose name is the name of the Use Way with the label *Use Way*. A package Use Way can be composed of other packages Use Way.
- Second, inside each package Use Way, the analyst must define the interfaces involved in the Use Way definition. Each **interface** groups the main interactive operations that the user can perform with the system. We propose defining interfaces by means of an element similar to a UML package with the label *Interface*.

Once we have defined the packages, the next step in our proposal is to define elemental primitives inside them. Elemental primitives are navigations, attributes and services, formulas, and displays. These elemental primitives can be defined with the following procedure:

- First, the analyst must define **navigations** in each Use Way with a Property to navigate among several interfaces. These navigations determine the target interfaces that can be reached from a source interface. We propose specifying these navigations by means of an arrow with a source and a target.
- Second, the analyst must specify attributes and services used in the Properties of the Use Way. An **attribute** is an element used to ask the user for data or to query stored data. A **service** is an element that represents an action that can be executed by the user. Attributes and services are related to a class; therefore we propose modelling them according to the UML notation used to represent classes.
- Third, the analyst must define **formulas** for the Properties of Use Ways that use conditions or dynamic information. The textual language to specify the formulas depends on analysts' preferences but we recommend OCL (Object Constraint Language), which is widely used in UML notations.
- Finally, the analyst must specify how the interface will be displayed to the user. We have called this primitive **display**, and it is defined textually using the UsiXML notation [19] (USer Interface eXtensible Markup Language), an XML-based markup language for defining user interfaces.

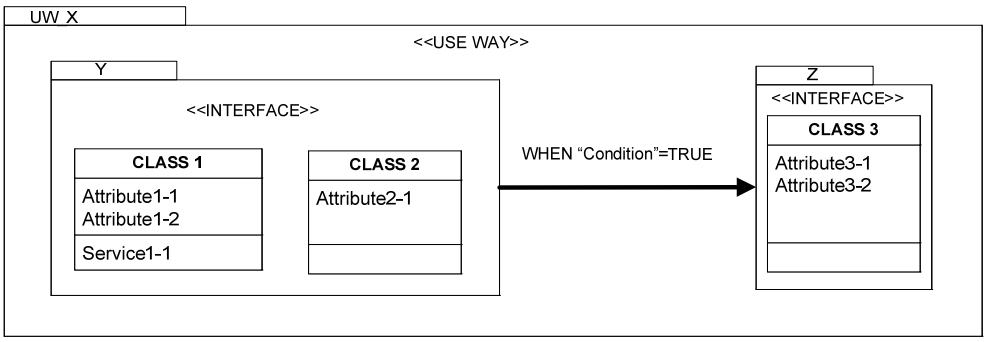


Fig. 2. Graphical notations to represent Use Ways in the Usability Model

Figure 2 shows the graphical elements that can be instantiated to represent any Use Way. This figure focuses on representing a generic example of the elements to represent a Use Way named "UW_X". Inside UW_X we have two interfaces "Y" and "Z". Interface "Y" has two classes with their own attributes and services ("Class 1" and "Class 2") and interface "Z" includes "Class 3". Navigation between both interfaces is represented with an arrow, and the navigation is only possible when a condition is satisfied. With the introduced primitives we have enough expressiveness to represent any property of the 22 Use Ways extracted from Juristo's FUFs. Note that depending on the Use Ways we aim to include in the system, we will use all these primitives or

some of them. The list of the primitives needed to work with each Use Way can be consulted in [20]

3.2 How to integrate the Usability Model in an Existing MDD Method

Each MDD method has its own conceptual model, since each method has a specific expressiveness to represent the system features. For example, the conceptual model of WebRatio [1] has a model called Hypertext Model to represent the interaction. This model is used to define pages, published content, operations, navigation links, and activity boundaries. However, other tools such as NDT [11] represent the interaction with a more basic conceptual model that does not support modelling activities or platform specific features. We propose extending the existing conceptual model of any MDD method with a Usability Model to obtain holistic MDD methods. As we commented above, by holistic we mean that all the relevant system perspectives (structure, functionality, and interaction) are properly incorporated into the modelling strategy.

Figure 3 represents a graphical summary of our proposal to include the Usability Model in an existing MDD method. In the figure, the existing conceptual model of the MDD method used as example is composed of two models: one model that represents the system structure (Class Model) and another model that represents the interaction (Task Model). In this example, we are considering two models, but the actual number of models that compose the conceptual model depends exclusively on the MDD method. Moreover, the existing MDD method can support code generation from its conceptual model by means of a model compiler. The level of automation of this process also depends on the existing MDD method, some of them are automatic (the model compiler generates full functional systems), others are semi-automatic (some manual implementations are needed).

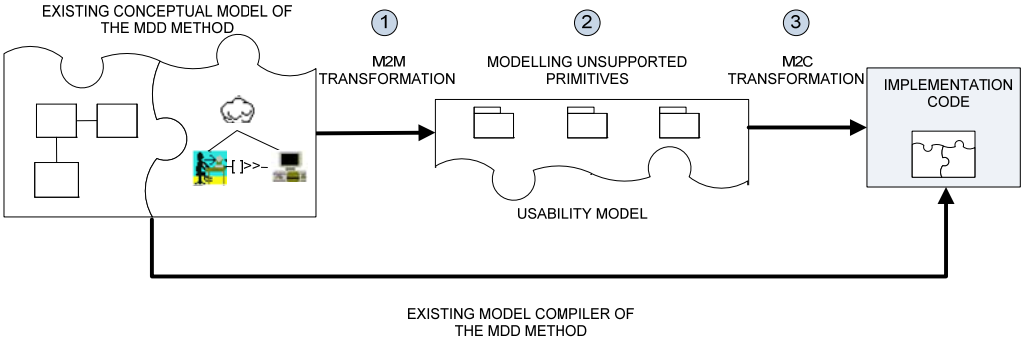


Fig. 3. An overview of the process that integrates the Usability Model in a holistic MDD method

Next, we present our integration process, which consists of three steps. Prior to apply these steps, the system must have been modelled with the conceptual model of the existing MDD method.

1. **Derivation of conceptual primitives from the existing conceptual model**: The primitives of the Usability Model are related to functionality, persistency, navigation or interaction elements that could have been defined previously in other

models of the MDD method (depending on the expressiveness of the existing MDD method). Elements that have been previously defined in the existing conceptual model do not need to be defined again in the Usability Model. In this first step, we automatically extract information defined in models of the MDD method and we include it as primitives of the Usability Model. For example, the Usability Model needs classes, attributes and services. In the conceptual model displayed in Figure 3, we can obtain all this information from the Class Model. The derivation of conceptual primitives from the existing conceptual model is achieved by means of model-to-model transformations. We propose performing these transformations with ATL [3], which is a language that allows transformations using a source meta-model and a target meta-model to be specified. In the case of Figure 3, the source meta-models are the meta-model of the Class Model and the meta-model of the Task Model, while the target meta-model is the meta-model of the Usability Model. The metamodel of the Usability Model used in the transformations can be viewed in [20].

2. **Modelling with the Usability Model**: Once the primitives of the Usability Model that were already defined in the conceptual model of the MDD method have been derived, the analyst must complete the specification of all the remaining Properties of the Usability Model. In this step, the analyst edits the Usability Model to add this new information. Note that the amount of primitives to model in the Usability Model depends exclusively on the number of derivations from the existing conceptual model of the MDD method. A method with a poor derivation (or even without any derivation) will involve more work in the Usability Model than a method where most primitives of the Usability Model can be derived automatically.
3. **Code generation**: Once the Usability Model has been fully defined, we can generate code from this model by means of automatic Xpand transformations [38] (model-to-code). The code generated from the Usability Model can be combined with the code generated by the model compiler of the MDD method, which generates code from the existing conceptual model. In the end, the final code should reflect all the elements specified in the existing conceptual model and in the Usability Model. The combination of the code generated from the Usability Model and the code generated with the existing model compiler should be as automatic as possible. If the Xpand transformations can be included inside the model compiler of the MDD method, the transformation will be completely automatic. If the Xpand templates cannot be included in the existing model compiler, both chunks of code must be manually assembled (with some tool to facilitate the integration) in order to build a single system. Note that, the most code is generated with the existing model compiler, the least code we must combine with the code generated from the Usability Model.

It is important to mention that the model-to-model transformations related to step 1 and the model-to-code transformations related to step 3 must be defined only once for each MDD method. After their definition, they can be reused to develop any software system. Therefore, steps 1 and 3 can be executed automatically by means of transformation templates. Note also that these transformations are specific for a MDD

method, since they depend on the existing conceptual model and the existing model compiler.

4 A Proof of Concept with an Existing Industrial MDD Method

The approach is introduced with the help of an illustrative example based on a car rental system. The example is focused on the functionality to create a new renting (*Create a renting*). This service is composed of three sub-functionalities: selection of the date to perform the renting, store information about the customer, and choose car preferences. In the example, we aim to improve the system usability using the mechanism *Structured Text Entry* (Table 1), which helps the user to provide data with a specific structure, and its three related Use Ways: UW_STE1, UW_STE2 and UW_STE3. Moreover, we have also considered the usability mechanism called *Step-by-step*, which aims to help users to do tasks that require different steps (Table 1). This mechanism has only one Use Way called *Wizard Definition* (UW_WD), which aims to define a wizard. This Use Way has several Properties: (1) *Service selection*, which defines the service that will be executed using a wizard*;* (2) *Division into steps*, which defines the steps that compose the wizard; (3) *Step description*, which provides a short description to guide the user in each step; (4) *Execution flow*, which defines all the possible flows throughout the steps.

We have focused our example on the combination of 4 Use Ways (UW_STE1, UW_STE2, UW_STE3, and UW_WD) because the primitives needed to represent all their Properties are enough to represent any other Property of the remaining 18 Use Ways. Therefore, the selected Use Ways with their Properties allow us to illustrate our proposal in its entirety. Table 2 shows the Properties of the Use Ways used in the example, the primitives of the Usability Model used to represent them, and the values of these properties in our example to create a renting. It is important to note that *Input field selection*, *Type of input widget* and *Step description* appear in several steps of the wizard. This is the reason why these Properties have several values.

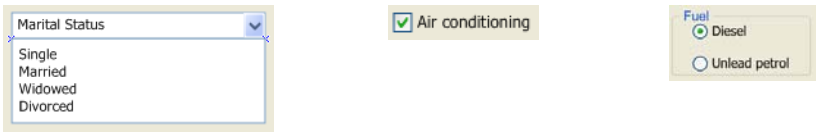
Next, we explain how these four Use Ways can be combined to improve the usability of the service that creates a new renting.

1. UW_STE1 (*Specify the input widget visualization type for enumerated values*): Some input elements can only accept a few possible values; the set of possible values is called *enumerated.* The interface can help the user to introduce a correct value by means of a widget which restricts the possible entries. This Use Way specifies how enumerated values will be displayed to the user. For example, when the customers are being registered in the system, they must provide their civil status. This information is enumerated, since only these four values are possible: single, married, widowed and divorced. In this case, the Property *Input field selection* has the value "Marital Status" and *Type of input widget* has the value "ListBox" (Figure 4a). The marital status is an enumerated with several possible values; therefore, the most suitable widget is a ListBox. Other examples of enumerated values are related to the description of the desired car, such as: whether or not the car has air conditioning, and the type of fuel. In the first case, the Property *Input field selection* has the value "Air conditioning" and the Property *Type of input widget* has the value "CheckBox*"* (Figure 4b); while in the second case the Property *Input field selection* has the value "Fuel" and the Property *Type of input widget* has the value "Ra-

dioButton" (Figure 4c). The existence or not of air conditioning can only accept two values: true or false; therefore the most suitable widget is the CheckBox. The type of fuel is an enumerated with two possible values (not Boolean); therefore, the most suitable widget is the RadioButton. The choice of the most suitable widget depends exclusively on the characteristics of the set of possible values.

Use Way	Property	Primitive of the Usability Model	Value
UW_STE1	Input field selection	Attribute	Marital Status, Doors, Fuel, h.p., Air conditioning
	Type of input widget	Display	ListBox, RadioButton and CheckBox, depending on the possible values to insert
UW_STE2	Input field selection	Attribute	Collection date and return date
	Regular expression	Display	"dd/mm/yyyy"
UW_STE3	Input field selection	Attribute	Collection date
	Default expression	Display	Today
UW_WD	Service selection	Service	Create a renting
	Division into steps	Interface	Selection of dates, store customer's data, choose car preferences
	Step description	Display	Each step has a descriptive text
	Steps execution flow	Navigation, Formula	If the customer's ID introduced in the first step already exists in the system, the next and last step is the car selection. Otherwise, the next step is the user registration and the last step is the car selection.

Table 2. Properties of UW_STE1, UW_STE2, UW_STE3 and UW_WD



a) b) c)

Fig. 4. Example of different widgets to display an enumerated attribute

2. UW_STE2 (*Mask definition*): There are some input elements where the user can insert any data but according to a specific format. This Use Way specifies a mask in order to ensure that the user inserts the data with the correct format. For example, in

the rent-a-car system, the collection date and the return date must be specified according to this format “dd/mm/yyyy” (Figure 5). In this example, the Property *Input field selection* has the value of “Collection date” and “Return date”, while the Property *Regular expression* has the value “dd/mm/yyyy”.

Fig. 5. Example of a mask in widgets to insert dates

3. UW_STE3 (*Default values*): Default values, apart from reducing the users’ work, also contribute to show how the user must provide the information. For example, the “Collection date” in the renting system (Figure 5) can display the current date in the correct format. The user can change the date taking as example the format of the default value. In this example, the Property *Input field selection* has the value “Collection date” and the Property *Default expression* has the value “today”.

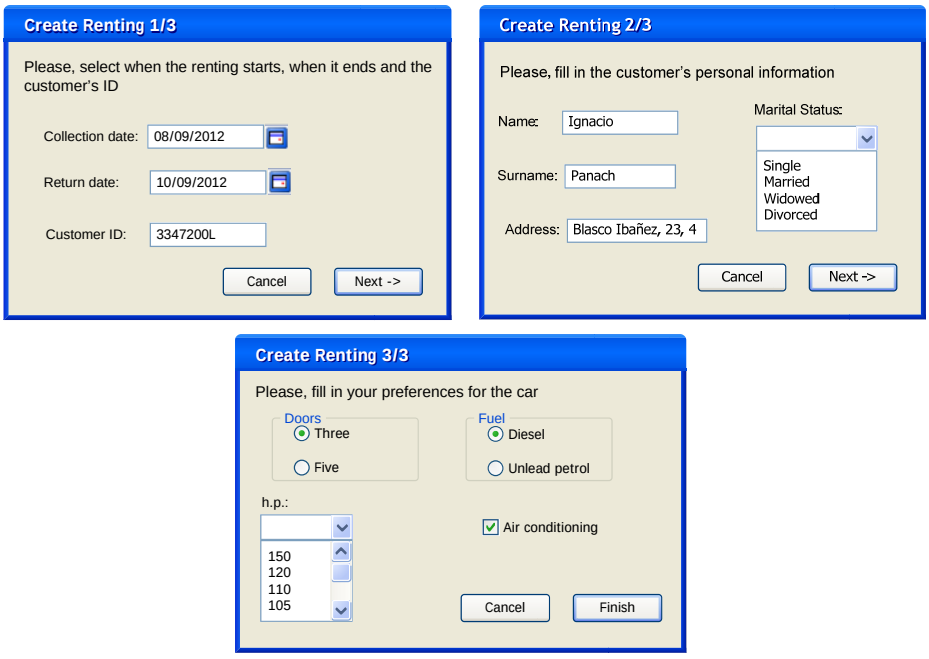


Fig. 6. Example of a wizard to create a customer

4. UW_WD (*Wizard definition*): Complex tasks should be divided into easier subtasks which guide the user. For example, in the rent-a-car system, the task to create a renting can be divided into three different subtasks: selection of dates, store customer’s data, and choose car preferences (Figure 6). In this example, the Property *Service selection* has the value “Create a renting”; the Property *Division into steps* is defined with the three subtasks that compose the service; the Property *Step description* is composed of the three texts displayed in the upper part of each window of Figure 6;

the Property *Execution flow* defines a navigation from the first step directly to the third step only if the customer already exists in the system. If the customer does not exist, there is a navigation from the first step to the second one and another navigation from the second step to the third one.

Next, we tackle how to deal with UW_STE1, UW_STE2, UW_STE3, and UW_WD in a MDD method. As development method, we use OO-Method [27], a MDD method that has been successfully implemented in industry with a tool called INTEGRANOVA [6], which can automatically generate full functional systems from a conceptual model. Its use in industry and its capability to generate full functional systems are the reasons why we have chosen OO-Method for the proof of concept of our proposal. The OO-Method conceptual model is composed of four complementary models:

1. **Object Model**: Specifies the structure of the system in terms of classes of objects and their relations. It is modelled as an extended UML [36] class diagram.
2. **Dynamic Model**: Represents the valid sequence of events for an object.
3. **Functional Model**: Specifies how events change the state of objects.
4. **Presentation Model**: Represents the interaction between the system and the user [23]. This model represents the interface and its component elements by means of Interaction Units composed of Elementary Patterns such as masks, filters, or navigations, among others.

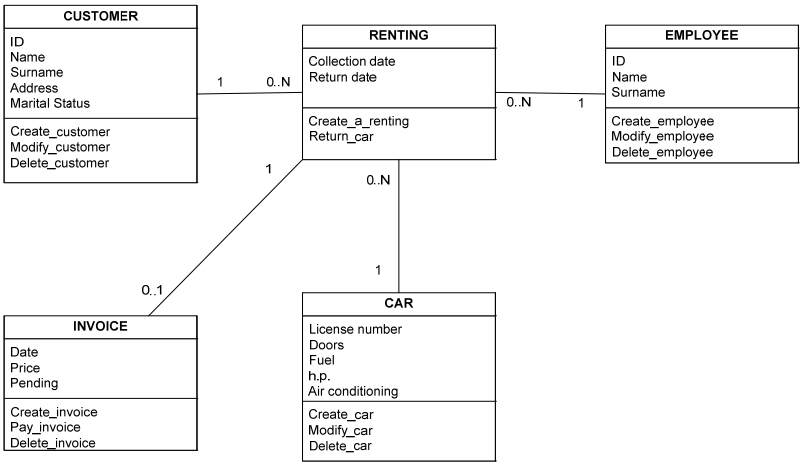


Fig. 7. Object Model of the car rental system

Previous to modelling Use Ways, we need to model the car rental system with the conceptual model of OO-Method. Figure 7 shows the OO-Method Object Model of the car rental system. The renting is performed by an employee and it involves a customer and a car. When the customer returns the car, the employee can create the invoice related to the renting. From all the services, our example is focused on the method Create_a_renting of the class *Renting*.

The other three models that compose the conceptual model of OO-Method (Dynamic, Functional and Presentation) were also defined but are not presented here for space reasons. Next, we explain how to model UW_STE1 (*Specify the input widget visualization type for enumerated values*), UW_STE2 (*Mask definition*), UW_STE3 (*Default values*) and UW_WD (*Wizard definition*) for developing the car rental system in OO-Method. Considering that we have already defined the four OO-Method models, the **first step** of our integration proposal consists in extracting information useful for the Usability Model from those OO-Method models. Taking into account the list of Properties of the four Use Ways considered (see Table 2), we can extract the information described below from the OO-Method's conceptual model:

- **UW_STE1**: The Property *Input field selection* can be derived from the attributes defined in the Object Model. We cannot obtain information for *Type of input widget* since OO-Method does not allow specifying the type of widgets.
- **UW_STE2**: *Input field selection* and *Regular expression* can be derived from the Presentation Model, where the analyst can define masks.
- **UW_STE3**: *Input field selection* and *Default expression* can be derived from the attributes of the Object Model, where the analyst can define default values for any attribute of a class.
- **UW_WD**: We cannot obtain information.

In order to derive these Properties from the existing OO-Method models, we have used ATL transformations (that must be previously defined). The source meta-models are the meta-models of the four models that define OO-Method (object model, dynamic model, functional model and presentation model); and the target meta-model is the meta-model of the Usability Model (it can be queried in [20]. Next, we show the ATL transformation rules that generate the part of the Usability Model which represents UW_STE2 and UW_STE3. In both cases we can derive all their Properties. In other examples with other Use Ways (such as UW_STE1), we can derive only some of their Properties (not all of them).

```
rule STE2_2_Usability
      from
      a: Presentation!Input
      to
      b: Usability!Attribute (Name <- a.Name, Type <-
a.type)
      c: Usability!UW_STE2 (Mask_expression <- a.format,
         UW_STE2_Attribute <- a.Mask_InputElement)

rule STE3_2_Usability{
      from
      a: Object!Attribute
      to
      b: Usability!Attribute (Name <- a.Name, Type <-
a.type)
```

```
      c: Usability!UW_STE3 (Default_value <-
a.Default_value,
         UW_STE3_Attribute <- a.Class_Attribute)
}
```

Figure 8 and Table 3 show, with grey background, the primitives of the Usability Model which have been derived from the OO-Method models. With regard to UW_STE1, the Property *Input field selection* is represented with the primitives *attribute* (marital status, doors, fuel, h.p., air conditioning) inside the packages *UW_STE1* (Figure 8). For UW_STE2, the Property *Input field selection* is represented with the primitives *attribute* (collection date, return date) inside the package *UW_STE2* (Figure 8). The Property *Regular expression* is represented with the UsiXML code which implements the primitive *display* (Table 3, tag *mask*). With regard to UW_STE3, the Property *Input field selection* is represented with the primitive *attribute* (collection date) inside the package *UW_STE3* (Figure 8) and the Property *Default expression* is represented in UsiXML with the primitive *display* (Table 3, tag *default*).

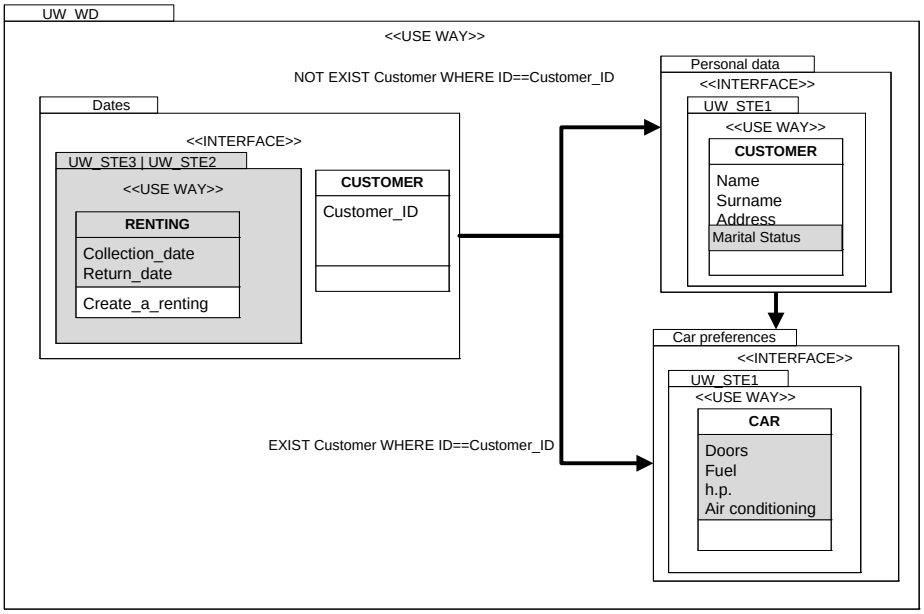


Fig. 8. Model to represent UW_STE1, UW_STE2, UW_STE3, UW_WD

<inputText id="Collection_date" name="Collection_date" mask="dd/mm/yy" default="today" maxLength="8" isEditable="true"/> <inputText id="Return_date" name="Return_date" mask="dd/mm/yy" maxLength="8" isEditable="true"/>
<outputText id="Description_wizard_1" name="Description_wizard_1" isVisible="true"

```
        value="Please, select when the renting starts, when it ends, and the customer's ID"/>
<listBox id="marital_status"
        name="marital_status" isVisible="true"
        isEnabled="true" textColor="#000000" multiple_selection="false"
        <option> Single</option>
        <option> Married</option>
        <option> Widowed</option>
        <option> Divorced</option>/>
 <radioButton id="three"
        name="three" isVisible="true"
        isEnabled="true" textColor="#000000"/>
 <radioButton id="five"
        name="five" isVisible="true"
        isEnabled="true" textColor="#000000"/>
<checkBox id="air_conditioning"
        name="air_conditioning" isVisible="true"
        isEnabled="true" textColor="#000000"/>
```

Table 3. A portion of the UsiXML code that represents the Properties *display* of our example

Once we have derived the Properties supported by the conceptual model of OO-Method, the **second step** of our integration proposal is to complete the Usability Model with the unsupported Properties. Next, we detail how to complete each Use Way in the Usability Model. Primitives added in this step are drawn with white background in Figure 8 and in Table 3. In our example, we have two Use Ways to complete in this step: UW_STE1 and UW_WD, since the OO-Method models do not have primitives to specify which widget is more suitable for enumerated attributes or which services will be executed in a wizard due to its complexity. UW_STE1 has only one Property that cannot be derived from OO-Method: *Type of input widget*. This Property is modelled with the primitive *display* and it is represented with the UsiXML code in Table 3 (tags: *ListBox*, *RadioButton* and *CheckBox*). With regard to UW_WD, it is a composition of other Use Ways, which includes UW_STE1 (*Specify the input widget visualization type for enumerated values*), UW_STE2 (*Mask definition*) and UW_STE3 (*Default values*). Figure 8 shows how a package *Use Way* can be an aggregation of other packages.

For UW_WD, the Property *Service selection* is represented with the primitive *service* (Figure 8). The Property *Division into steps* is represented with the primitive *interface*; we have defined as many interfaces as steps compose the wizard (Figure 8). The Property *Step description* is represented with the primitive *display* in the UsiXML code (Table 3, tag *OutputText*). Finally, the Property *Execution flow* is represented with two primitives: *navigation* and *formula* (Figure 8). Primitive *navigation* is used to specify which step is the next and which one is the previous. In case we have more than one possible navigation (such as in our example), we can specify a formula to define which is the correct navigation depending on a condition. In the example, the condition depends on whether or not the customer already exists in the system. If the customer exists, the user navigates from the selection of dates to choose

car preferences, but if the customer does not exist, the user navigates from the selection of dates to store information about the customer.

Finally, in the **third step** of our integration proposal, the Usability Model must be transformed into code that implements all the characteristics represented in it. This transformation is performed with Xpand [38]. The code derived from the Usability Model must be included in the code generated with the OO-Method model compiler. Xpand templates (that must be previously defined) have been combined with the transformation rules of the OO-Method model compiler; therefore, both generations can be performed automatically as a simple generation. Below, we show a small chunk of Xpand code used in the transformation from UW_WD into C# code. This example of code generates a C# form for each step defined in the wizard, including the descriptive text of the step.

```
«DEFINE CSClass FOR Class»
      «FOREACH Step AS st»
      «FILE Class.name+".cs"»

public class «st.name» {
private System.ComponentModel.IContainer components =
null;

private void InitializeComponent()
        {
         this.components = new Sys-
tem.ComponentModel.Container();
this.AutoScaleMode = Sys-
tem.Windows.Forms.AutoScaleMode.Font;
this.labeldescription.Text = «st.description»;
            }
       }
            «ENDFOREACH»
      «ENDFILE» «ENDDEFINE»
```

After applying our proposal, we have a full functional system that has been developed exclusively with conceptual models (OO-Method and the Usability Model). The analyst has not written a single line of code to implement the system. At this moment a question about the efficiency of the proposal can arise: is it more efficient modelling usability features or implementing these features manually in the code generated by OO-Method? The next section shows a discussion about both options.

5 An Exploratory Efficiency Analysis

This section focuses on comparing the efficiency of the analyst working with our approach with the efficiency of the analyst implementing the Use Ways manually. We have measured the efficiency as the time to develop a system. Note that in the software development process, the cost of developing a system is strongly related to developing time. There are many authors who have studied the benefits of the MDD

paradigm with regard to a manual software development method, such as, Sendall [31] and Selic [30]. According to those works, the efficiency is one advantage of MDD, but this advantage is not ensured if the analyst has to manually implement part of the code. Moreover, manual coding to implement usability features is hardly reused, which decreases the analyst's efficiency. For example, UW_STE2 and UW_STE3 depend on a specific system, since some masks and default values are exclusive for an interface. Another advantage of dealing with usability features using the MDD paradigm is that if a flaw is propagated from the models to the generated system, once this flaw has been detected, the analyst can change some primitives in the model and regenerate the code quickly. This advantage is in accordance with some recommendations from the system architecture area, which claim that the construction of rapid prototypes is essential to obtain good usability levels [4]. The goal of this paper is not to study the advantages and disadvantages of the MDD paradigm with regard to manual software development methods. Our aim is to evaluate whether or not modelling usability features improves the efficiency in the software development process.

For this aim, we compare the effort of the analysts when they work without the Usability Model to their effort when they work with the Usability Model. The metrics used to evaluate the effort not using the Usability Model are: time spent and written lines. The metrics used to evaluate the effort using the Usability Model are: time spent and number of primitives used (since in this case, the analyst does not write lines of code). We focus our study on the Use Ways used in our illustrative example to create a rental: UW_WD (*Wizard definition)*, UW_STE1 (*Specify the input widget visualization type for enumerated values*), UW_STE2 (*Mask definition*) and UW_STE3 (*Default values*).

We have used two subjects in the evaluation: Subject1 and Subject2. Subject1 is an expert in C# and he has developed more than 10 applications using this language. We used this subject to evaluate the effort of including Use Ways manually in C# code (without the Usability Model). This subject is not an author of the approach, and therefore, he did not know the Usability Model. We described textually which Use Ways were needed in the system and he implemented them. Subject2 was an author of the paper and perfectly knew the Usability Model. We used this subject to evaluate the effort of working with the Usability Model. Furthermore, Subject2 has defined previously several transformations among models and he has good knowledge of ATL and Xpand. Both subjects were researchers of PROS ("Centro de Investigación en Métodos de Producción de Software") and knew perfectly OO-Method, the MDD method used in the exploratory study. They had developed more than 20 applications using INTEGRANOVA.

We have modelled the create rental functionality of the car rental system using OO-Method and we have generated its code with INTEGRANOVA. INTEGRANOVA generates the code in C#; therefore, our evaluation is based on this language. For other languages, the number of lines and the time spent could vary, but not significantly. The code generated with INTEGRANOVA and the OO-Method models are the starting point for the experiment. Next, we study how to include unsupported Use Ways by means of two techniques: manual implementation and the Usability Model.

First, we present the case in which the analyst manually implements unsupported Use Ways in the code generated by INTEGRANOVA (without using the Usability Model). Table 4 shows the time spent and the number of lines written to manually implement unsupported Properties of the Use Ways. UW_STE2 and UW_STE3 have not been included in this table since their code generation is fully supported by OO-Method and we did not have to write any line of code to implement them. These metrics have been extracted from the work of Subject1. Time spent includes not only the time to write the sentence, but also the time to test and to correct the faults.

Use Way	Time	Number of lines
UW_WD (*Wizard definition)*	120 minutes	120
UW_STE1 (*Specify the input widget visualization type for enumerated values*)	30 minutes	20

Table 4. Time spent and number of lines to implement unsupported Use Ways

Second, we present the case in which the analyst (Subject2) uses the Usability Model to include unsupported Use Ways. Our bases are the OO-Method models that represent the create rental functionality of the car rental system. According to our proposal, firstly Subject2 applies ATL transformations to obtain an initial version of the Usability Model. This initial version already contains values for the Properties of the Use Ways supported by OO-Method. In this case, the primitives that represent the Property *Input field selection* of UW_STE1 and the primitives used to represent all the Properties of UW_STE2 and UW_STE3 are automatically derived from the OO-Method models by means of ATL transformations. Since these transformations are automatic, the time spent to apply them can be considered insignificant. Once we have the initial version of the Usability Model with the supported Use Ways, Subject2 had to manually add the unsupported Use Ways (UW_WD and the Property *Type of input widget* of UW_STE1 in our example). For this activity, we do not have a graphical tool that supports the creation and edition of Usability Models. We use Eclipse [9] to work with instances of the usability meta-model, using a tree-view. Table 5 shows the time used by Subject2 to model UW_WD and UW_STE1 with Eclipse. Since we do not have to write code when using our approach, we have measured the number of primitives we had to manually add to our Usability Model instead of the number of written lines of code.

Use Way	Time	Number of primitives
UW_WD (*Wizard definition)*	10 minutes	14
UW_STE1 (*Specify the input widget visualization type for enumerated values*)	5 minutes	5

Table 5. Time spent and number of primitives to model unsupported Use Ways

Once we have completed the Usability Model, the code is generated with Xpand templates in an insignificant time and it is combined with the code generated by INTEGRANOVA, which implements all the other features of the system.

Comparing the efforts to include Use Ways manually in the code (Table 4) and using the Usability Model (Table 5), we can state that the analyst's effort decreases with

our proposal. However, there is a disadvantage with our approach: the definition of ATL transformations and Xpand templates. Both elements must be defined only once and then they can be applied automatically for the development of any system, but their definition is complex. Next, we measure the effort made by Subject2 for defining these transformations for our illustrative example. Table 6 shows the time spent to define ATL transformations and the amount of lines of code written. We consider that the usability meta-model and the meta-model of the models that compose OO-Method already exist. In our example, we have three ATL transformations, one to derive the Property *Input field selection* of UW_STE1 and two transformations to derive all the Properties of UW_STE2 and UW_STE3 respectively. The presented times also include the time needed to test the transformations and to correct errors.

Use Way	Time	Number of lines
UW_STE1(*Specify the input widget visualization type for enumerated values*)	15 minutes	7
UW_STE2 (*Mask definition)*	15 minutes	7
UW_STE3 (*default values*)	15 minutes	7

Table 6. Time spent and number of lines to define ATL transformations

With regard to the definition of Xpand templates, it is more difficult, since they require many more lines of code than the ATL transformations. Table 7 shows the time and the number of lines that Subject2 used to generate the code for the conceptual primitives involved in our illustrative example. The efforts shown in Table 7 do not include the whole transformations for all the possibilities of the primitives. Subject2 only defined the possibilities for our example. For instance, the primitive *display* can have many different values and their complete transformation will be much more complex than the values expressed in the table.

Primitive of the Usability Model		Time	Number of lines
UW_STE1	Attribute	15 min	56
	Display	25 min	112
UW_STE2	Attribute	20 min	49
	Display	20 min	87
UW_STE3	Attribute	15 min	52
	Display	25 min	96
UW_WD	Interface	35 min	325
	Navigation	30 min	126
	Display	130 min	425
	Attribute	30min	100
	Service	15 min	24

Table 7. Time spent and number of lines to define XPand templates

Note that Subject2 had a previous knowledge of OO-Method, INTEGRANOVA and the Usability Model, which benefits his development times. If our approach were used by analysts without experience in any MDD method or without knowledge of the

Usability Model, these times would be higher. Next, we discuss which are the skills required by the analyst to work with an existing MDD method and the Usability Model. First, the analyst must have previous experience working with the existing MDD method. The Usability Model depends on the conceptual model of the existing MDD method and the analyst should have a good knowledge of it. Second, using the Usability Model does not ensure that every developed system will be usable. The Usability Model offers a set of conceptual primitives to allow analysts to adapt the system to usability requirements. However, the approach cannot ensure that the primitives that compose the Usability Model are combined properly. Therefore, the analyst must have a previous knowledge of usability recommendations to model usability features such a way usability will be optimized.

In summary, the Usability Model improves the analyst's effort, but it needs an initial investment of time to define rules (ATL and Xpand). If we are going to develop many systems and we would like to include usability features in many of these systems, our proposal is better than a manual implementation, since ATL and Xpand rules are defined only once. However, the proposal is not suitable for MDD methods that usually do not need to include usability features in their developments.

6 State of the Art

If we look for existing proposals that deal with usability in MDD, we notice that, currently, there are not many works in the literature. This could be related to the existence of a gap between the communities of software engineering and human-computer interaction [29]. Our work is a step forward to fill in this gap.

Authors that propose considering usability in MDD methods are Taleb [34], Gull [15], Cleland-Huang [7] and Luna [21]. Taleb describes how the principles of the Object Management Group with regard to Model-Driven Architecture can be used to develop web applications, and at the same time, to ensure their cross-platform portability and usability. Gull defines a process model for web-based applications that is divided into three phases: requirement engineering, design, and implementation. In all these three phases, the emphasis is on the usability. Cleland-Huang has proposed a goal-centric approach to manage the impact of change upon non-functional requirements (which include usability). First, usability is modelled as goals and, second, traces due to changes in usability requirements are generated using a probabilistic network model. Luna focuses on how to address usability requirements in a test-driven and model-based web engineering approach. However, in these three proposals, the authors do not specify the traceability of usability among the different development steps. Moreover, a specific notation to represent usability features in each development step has not been provided. In our proposal, traceability among models is hidden for the analyst thanks to model-to-model and model-to-code transformations. Moreover, we have provided an unambiguous notation to represent usability features. A precise notation is essential for performing transformations throughout the whole software development process.

There are also works that propose techniques from the human-computer interaction field to be integrated in MDD, such as the work of Wang [37]. Wang proposes a user-centred design where the users play an important role in modelling the interface. This work focuses only on usability features related to the interface display, and disregards

the features related to functionality. In contrast, Sottet [32] is an author that deals with usability considering functional usability features. Sottet uses MDD mappings for embedding both usability description and control. For Sottet, a user interface is a graph of models, and usability is described and controlled in the mappings between these models. In Sottet's proposal, the analyst must specify the transformation rules for each system and this is not trivial. Garcia-Frey [14] has defined a quality meta-model (which includes usability features) that unifies aspects from the Human-computer interaction with MDD. The approach is useful to explain design decisions through quality models. Wang, Sottet and Garcia-Frey have focused their approaches on modelling usability features strongly related to interaction, while usability features related to functionality have not been analyzed. The main contribution of our work is just to cover this gap between usability and functionality. There are other characteristics that are supported by Sottet's approach but we do not solve with our proposal, such as the adaptability of user interfaces.

Other proposals use existing models to represent usability features, such as Sousa [33]. Sousa has defined an activity-based strategy to represent usability goals. However, in this proposal, we cannot model how usability features are related to the system functionality. Röder [28] proposes a method to specify functional usability features at requirements elicitation step using textual templates. In that proposal, usability requirements are specified together with functional requirements using use case specifications. The main difference of that work with regard to our proposal is that use cases are described textually, and model-to-model transformations are difficult to perform using this ambiguous notation. Other authors that represent usability in existing models are Tao [35] and Brajnik [5], who propose modelling usability by means of state-transition diagrams. However, state-transition diagrams are only able to represent interactions, so they cannot represent all the usability features. Our approach combines interaction and functionality since it does not depend on a specific model (such as state-transition diagrams in the proposals of Tao and Brajnik). Our process has been defined to work with several models thanks to model-to-model transformations. This enhances the expressiveness to represent usability features not only related to interaction but also related to functionality, since usually, interaction and functionality are represented in different models.

There are also some works [12][22] related to measuring the system usability in MDD conceptual models. Fernandez [12] proposes a model to evaluate system usability from conceptual models. According to Fernandez, the evaluation performed at the conceptual model level produces a platform-independent usability report, which provides feedback to the system analysis stage. Molina [22] proposes measuring usability attributes focused on navigational models. The approach focuses on navigational models and provides a tool which offers automatic support for all the activities. However, in both proposals, many usability attributes are subjective, and therefore they cannot be measured automatically, without taking into account the user. For instance, attributes related to the attractiveness sub-characteristic [16] cannot be measured by means of conceptual models. Therefore, the results of the early usability evaluation do not necessarily represent the usability of the overall software system. Our approach aims to model usability features, but currently we have not defined metrics to measure the level of usability represented within the Usability Model.

After studying related works, we conclude that existing proposals for dealing with functional usability features in MDD present some problems when we want to include them in a real software development process. First, few works have a specific notation to represent functional usability features in a model, and the existing notations do not cover all the existing features. Second, it is not clear how to include usability features throughout the whole software development process since existing proposals do not specify the traceability among models.

7 Conclusion

This paper presents a proposal to model functional usability features in a MDD method, taking the advantages of the MDD paradigm with regard to manual implementations. The paper is based on a Usability Model composed of several primitives that work like building blocks. Different combinations of primitives allow any FUF to be represented. It is important to note that there are many other non-functional usability features that are out of scope of this paper, such as, understandability or attractiveness. Moreover, systems from other areas different than management information systems, such as multimedia applications or virtual reality systems are out of scope too.

The main advantages of our proposal with regard to existing proposals to deal with usability in MDD are: (1) The Usability Model can represent most functional usability features for a management information system (we can ensure that it supports all the FUFs defined by Juristo); (2) The notation used in the Usability Model has an unambiguous syntax and semantics, which allows transformations to be performed; (3) The Usability Model can be used in any MDD method (we have used OO-Method as example).

It is important to note that this paper does not evaluate the benefits of the MDD paradigm. We start from the idea that there are previous works that claim MDD is suitable for reducing the cost/benefit ration when producing software, for dealing with changeable requirements, and for developing software with independency of the programming language. These advantages are not exclusive of our proposal but they are shared with all the proposals based on the MDD paradigm.

We have learned some lessons and identified some limitations applying the proposal to OO-Method. First, we have used ATL and Xpand transformations because we have OO-Method meta-models in ecore, however, any other transformation language such as QVT or XSLT are also applicable to our proposal. Second, the complexity and amount of ATL and Xpand transformations that have to be written depends mainly on the MDD method chosen. OO-Method generates the whole system (structure, functionality, and interaction), but MDD methods with less powerful model compilers will need more effort to define transformations. However, it is important to mention that these transformations are defined only once and can be used indefinitely in every software development. Third, the existence of a Usability Model does not ensure that generated systems are usable. The analyst must follow usability guidelines to combine the primitives properly. Fourth, the analyst does not need to know the correspondence between primitives of the Usability Model and functional usability features. The analyst must only know the meaning of the primitives that compose the Usability Model and how to combine them. Fifth, the approach does not

prevent the occurrence of any usability problem. We only propose a solution for functional usability features. Features unsupported by our approach (such as adaptability or customizability) must be included in the system manually. Sixth, another identified limitation is that the approach is platform and modality dependent, since the analyst specifies what widgets to use.

As future work, we plan to define metrics to measure the usability of the system based on the conceptual primitives of the Usability Model. In this way, the analyst will be able to measure the usability of the system before generating the code that implements it. Another future work is to design and implement a tool to draw the Usability Model graphically. Currently, we can only work with an instance of the usability meta-model with a tree-view in Eclipse. Moreover, we plan to define ATL and Xpand templates to generate code for every Use Way. Nowadays, we only have the code for UW_WD, UW_STE1, UW_STE2 and UW_STE3. This enhancement will help to perform a deeper experiment with more subjects and more Use Ways. Finally, we plan to study other usability features apart from Juristo's FUFs.

Acknowledgments

This work has been developed with the support of MICINN (PROS-Req TIN2010-19130-C02-02), UV (UV-INV-PRECOMP12-80627), GVA (ORCA PROMETEO/2009/015), and co-financed with ERDF. We acknowledge the support of the ITEA2 Call 3 UsiXML (20080026) and funding by the MITYC (TSI-020400-2011-20).

References

1. Acerbis, R., Bongio, A., Brambilla, M., Butti, S.: WebRatio 5: An Eclipse-Based CASE Tool for Engineering Web Applications. LNCS 4607 (2007) 501-505.
2. AndroMDA, http://www.andromda.org/.
3. ATL: http://www.eclipse.org/atl/
4. Bass, L., Clements, P., Kazman, R. Software Architecture in Practice (3d Edition), Addison-Wesley Professional (2012).
5. Brajnik, G.: Is the UML appropriate for Interaction Design? Università di Udine (2010) 6.
6. CARE Technologies S.A. http://www.care-t.com
7. Cleland-Huang, J., Settimi, R., BenKhadra, O., Berezhanskaya, E., Christina, S.: Goal-centric traceability for managing non-functional requirements. Proc. of the 27th international conference on Software engineering. ACM, St. Louis, MO, USA (2005) 362-371.
8. Davis, F.D.: User acceptance of information technology: system characteristics, user perceptions and behavioral impacts. Int. Journal Man-Machine Studies 38 (1993) 475-487.
9. Eclipse : http://www.eclipse.org
10. Embley, D.W., Liddle, S., Pastor, Ó.: Conceptual-Model Programming: A Manifesto. Handbook of Conceptual Modeling. Springer (2011) 3-16.
11. Escalona, M.J. and Aragon, G. "NDT. A Model-Driven Approach for Web Requirements." Software Engineering, IEEE Transactions on 34(3): 377-390, (2008).
12. Fernandez, A., Abrahao, S., Insfran, E.: Empirical validation of a usability inspection method for model-driven Web development. J. Syst. Softw. 86 (2013) 161-186.
13. Folmer, E., Bosch, J.: Architecting for usability: A Survey. Journal of Systems and Software, Vol. 70 (1) (2004) 61-78.
14. Frey, A.G., Céret, E., Dupuy-Chessa, S., Calvary, G.: QUIMERA: a quality metamodel to improve design rationale. Proc. of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems. ACM, Pisa, Italy (2011) 265-270.

15. Gull, H., Azam, F., Iqbal, S.Z.: Design of Novel Usability Driven Software Process Model. (IJCSIS) Int. Journal of Computer Science and Information Security 8 (2010) 46-53.
16. ISO/IEC 9126-1, Software engineering - Product quality - 1: Quality model (2001).
17. Juristo, N., Moreno, A.M., Sánchez, M.I.: Analysing the impact of usability on software design. Journal of Systems and Software, Vol. 80 (2007) 1506-1516.
18. Lawrence, B., Wiegers, K., Ebert, C.: The top risk of requirements engineering. IEEE Software, Vol. 18 (2001) 62-63.
19. Limbourg, Q., Vanderdonckt, J.: Usixml: A User Interface Description Language Supporting Multiple Levels Of Independence. Engineering Advanced Web Applications. Rinton Press, Paramus, New Jersey (2004).
20. List of Use Ways and Properties: http://hci.dsic.upv.es/UsabilityModel/UseWaysList.html
21. Luna, E.R., Panach, J.I., Grigera, J., Rossi, G. and Pastor, O. Incorporating Usability Requirements In A Test/Model-Driven Web Engineering Approach Journal of Web Engineering, (2010), 132-156.
22. Molina, F. and Toval, A. 2009. Integrating usability requirements that can be evaluated in design time into Model Driven Engineering of Web Information Systems. Advances in Engineering Software. vol. 40, 1306-1317.
23. Molina, P.J., Meliá, S., Pastor, Ó. JUST-UI: A User Interface Specification Model.: Proc of Computer Aided Design of User Interfaces, CADUI'2002, Valenciennes, Francia. (2002).
24. Olive, A.: Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. Proc. of the 16th Conference on Advanced Information Systems Engineering, LNCS 3520 , Springer-Verlag, Porto, Portugal, (2005) 1-15.
25. Panach, J.I., Aquino, N. and Pastor, O. A Model for Dealing with Usability in a Holistic MDD Method. User Interface Description Language (UIDL), Thales Research and Technology, Lisbon (Portugal), (2011), 68-77.
26. Panach, J.I., España, S., Moreno, A., Pastor, Ó. Dealing with Usability in Model Transformation Technologies. ER 2008. Springer LNCS 5231, Barcelona (2008) 498-511.
27. Pastor, O., Molina, J.: Model-Driven Architecture in Practice. Springer, Valencia (2007).
28. Röder, H.: A pattern approach to specifying usability features in use cases. Proceedings of the 2nd International Workshop on Pattern-Driven Engineering of Interactive Computing Systems. ACM, Pisa, Italy (2011) 12-15.
29. Seffah, A., Vanderdonckt, J.,Desmarais, Michel C. . Human-Centered Software Engineering: Software Engineering Architectures, Patterns, and Sodels for Human Computer Interaction. Human-Centered Software Engineering, Springer: 1-6, (2009).
30. Selic, B.: The Pragmatics of Model-Driven Development. IEEE software 20 (2003) 19-25
31. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software 20 (2003) 42-45.
32. Sottet, J.-S., Calvary, G., Coutaz, J., Favre, J.-M.: A Model-Driven Engineering Approach for the Usability of Plastic User Interfaces. In Proc. of Engineering Interactive Systems (2007) 22-24.
33. Sousa, K., Mendonça, H., Vanderdonckt, J.: Towards Method Engineering of Model-Driven User Interface Development. TAMODIA, LNCS 4849. Springer, Toulouse (France) (2007) 112-125.
34. Taleb, M., Seffah, A., Abran, A.: Investigating Model-Driven Architecture for Web-based Interactive Systems. e-Minds: Int. Journal on Human-Computer Interaction 2 (2010).
35. Tao, Y.: An Adaptive Approach to Obtaining Usability Information for Early Usability Evaluation. IMECS (2007) 1066-1070.
36. UML: http://www.uml.org/
37. Wang, X., Shi, Y.: UMDD: User Model Driven Software Development. IEEE/IFIP Int. Conference on Embedded and Ubiquitous Computing, Shanghai (China) (2008).
38. XPAND: http://www.eclipse.org/modeling/m2t/?project=xpand