A Model for Dealing with Usability in a Holistic MDD Method

Jose Ignacio Panach, Óscar Pastor, Nathalie Aquino
Centro de Investigación en Métodos de Producción de Software
Universitat Politècnica de València
Camino de Vera s/n, 46022 Valencia, Spain
{jpanach, opastor, naquino}@pros.upv.es

ABSTRACT
Currently, the importance of developing usable software is widely known. For this reason, there are many usability recommendations related to system functionality (called functional usability features). If these functional usability features are not considered from the very early steps of the software development, they require many changes in the system architecture. However, the inclusion of usability features from the early steps in a traditional software development process increases the analyst's workload, who must consider not only features of the business logic but also usability features. In the Software Engineering community, holistic MDD methods are a solution to reducing the analysts' workload since analysts can focus all their efforts on the conceptual model (problem space), relegating the architecture design and the implementation code (solution space) to automatic transformations. However, in general, MDD methods do not provide primitives for representing usability features. In this paper, we propose what we call a Usability Model that gathers conceptual primitives to represent functional usability features abstractly enough to be included in any holistic MDD method.

Keywords
Model-Driven Development, usability, conceptual model

INTRODUCTION
According to ISO 9126-1 [10], usability is a key issue in obtaining good user acceptance of the software [6]. Some authors have divided usability recommendations into two groups [11]: recommendations that only affect the interface presentation (e.g., a label meaning), and recommendations that affect the system functionality (e.g. a cancel function). This second type is called functional usability features, and it is the most difficult type to include in the software [11], since these features affect the system interface as well as the system architecture. For example, the feature Cancel aims to cancel the execution of a service. The implementation of this usability feature is not a single button in the interface; on the contrary, this feature also affects data persistency 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 [8]. All these works propose including functional usability features from the early steps of the software development process, since they involve many changes in the system architecture if they are considered only when interfaces are designed. However, these approaches have some disadvantages:

- **Cost/benefit ratio**: The analyst must deal with usability from the requirements capture step until the implementation throughout the entire development process. This increases the analyst's effort and the cost/benefit ratio is not always favourable for features that are difficult to implement [11].
- **Changeable requirements**: Usability requirements (like other system requirements) are continuously evolving [12] 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 the Model-Driven Development (MDD) method is a suitable solution for reducing all these disadvantages [21][20]. MDD proposes that the analyst must focus all their efforts on building a conceptual model that represents all the system features (a holistic conceptual model) [17]. What we mean by "holistic" is that the conceptual model must include all the relevant systems perspectives: class structure, functionality and interaction. While class structure and functional view are supported in almost all the existing MDD approaches, interaction view is usually designed once the architecture has been finished. Our work focuses on modelling interaction adequately, at the same level of data and functionality, to provide a full system description in the conceptual model.

Space for copyright

These complementary perspectives must provide the corresponding conceptual primitives, which are modelling elements having the capability of abstractly representing a feature of the system. Examples of conceptual modelling elements for the class structure view are classes of a class diagram, attributes, and services. Examples of conceptual modelling elements for the functional view include service pre/post conditions, valid states and transitions. In this paper, our intention is to focus on conceptual modelling elements for the interaction view, and in particular, for functional usability features. The holistic conceptual model can then be seen as the input for a model compiler that can generate the software application automatically (or semi-automatically, depending on the model compiler capacity). Providing such a MDD-based software production environment, the problems of existing approaches that deal with functional usability features can find an adequate solution.

There are currently several MDD methods which model full functional systems, such as WebRatio [1], AndroMDA [2], and OO-Method [19] among others. However, none of them can model most functional usability features in their conceptual model. These features must be manually implemented, inheriting all the disadvantages of manual development mentioned above. This paper aims to extend existing MDD methods with conceptual primitives that represent functional usability features well known in the Human Computer Interaction community [11]. All these primitives are gathered in what we call the Usability Model.

The paper is structured as follows. Section 2 presents the state of the art. Section 3 describes the functional usability features used in our proposal. Section 4 explains our proposed Usability Model with its primitives. Section 5 describes how to include the Usability Model in a holistic MDD method using OO-Method as example. Finally, section 6 presents some conclusions and future work.

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. Two examples of authors that propose considering usability in MDD methods are Taleb [24] and Gull [9]. The main disadvantage of both proposals is that these authors do not specify the usability traceability among the different development steps. Moreover, a specific notation to represent usability features in each step does not exist.

There are also works that propose integrating HCI techniques in MDD, such as Wang [27]. 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, not to the functionality. In contrast, Sottet [22] is an author that deals with usability considering functional usability features. This author investigates 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. The main disadvantage of Sottet's proposal is that the analyst must specify the transformation rules for each system and this is not trivial.

Other proposals use existing models to represent usability features, such as Sousa [23]. Sousa has defined an activity-based strategy to represent usability goals. The main disadvantage of this proposal is that we cannot model how usability features are related to the system functionality. Other authors that represent usability in existing models are Tao [25] and Brajnik [4], 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.

There are also some works [7][15] related to measuring the system usability in MDD conceptual models. Fernandez [7] proposes a model to evaluate system usability from conceptual models. Molina [15] proposes measuring usability attributes focused on navigational models. The main shortcomings of these proposals is that many usability attributes are subjective, and therefore cannot be measured automatically without taking into account the user. For instance, attributes related to the attractiveness subcharacteristic [10] cannot be measured by means of conceptual models. Therefore, the result of early usability evaluation is a sort of prediction.

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 existing notations do not cover the model of 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.

BACKGROUND: PROPERTIES OF FUNCTIONAL USABILITY FEATURES

In previous works, we have extracted the properties that are needed to configure a set of usability features [18] defined by Juristo [11] and called FUFs (Functional Usability Features). We chose these usability features from all the existing ones since FUFs are specific to management information systems, the target systems of our work. Moreover, FUFs have templates to capture usability requirements that are very useful for identifying features' properties. In the FUF definition, each FUF has a main objective that can be specialized into more detailed goals named mechanisms. A detailed explanation of properties derived from FUFs is out of scope of this paper, but can be consulted in [14]. Next, we summarize the process we followed to extract these properties:

1. We defined **Use Ways (UW)** using the usability mechanisms description. Each usability mechanism can achieve its goal through different means. We called each such mean Use Way (UW). Each UW has a specific target to achieve as part of the overall goal of the mechanism. For example, the usability mechanism called *System Status Feedback* aims to inform the user about the internal system state. We identified that this goal can be achieved in at least three use ways: (1) *Inform about the success or the failure of an execution* (UW_SSF1): it informs if an action execution finished successfully; (2) *Show the information stored in the system* (UW_SSF2): it shows information of the system before the user triggers an action; (3) *Show the state of relevant actions* (UW_SSF3): it indicates which actions cannot be triggered in the current system state.
2. We defined **properties** for each UW. We called the different UW configuration options to satisfy usability requirements as Properties. For example, we identified that the *Inform about the success or the failure of an execution* Use Way is composed of two properties: (1) *Service selection*; (2) *Message visualization*. By means of *Service selection* the analyst can specify the service that will inform about the success or failure; by means of *Message visualization*, the analyst can specify how the message will be displayed (format, position, icon, etc.).

In previous works [18], we defined Use Ways and properties but their inclusion in an MDD method affected its conceptual model. This paper aims to define a generic approach that does not affect the existing conceptual model. This new approach is based on a Usability Model explained in the next section.

A USABILITY MODEL

This section presents the conceptual primitives needed to represent Use Ways extracted from FUFs. Since currently there is no a standard notation to represent usability features, we have used a set of primitives very similar to UML notation [26]. We use graphical elements already defined in UML but we have extended these elements with some textual descriptions. The new conceptual primitives are gathered in a model called Usability Model.

Next, we detail primitives to represent the set of 22 Use Ways [14] and their properties in a Usability Model. As an illustrative example to introduce these primitives, we use a system to manage a library. In the example, we aim to improve the system usability by means of the three Use Ways described above: UW_SSF1, UW_SSF2 and UW_SSF3. We have focused our example in these three elements because the primitives needed to represent all their properties are enough to represent any other property of the remaining 19 Use Ways.

Table 1 shows the properties of the three Use Ways used in the example and their values. These properties must be specified by the analyst in order to develop a usable system. Next, we present how these three Use Ways improve the usability of the system to manage a library.

Table 1. Properties of UW_SSF1, UW_SSF2 and UW_SSF3

Use Way	Property	Value
UW_SSF1	Service selection	All the services
	Message visualization	Display the failure message textually
UW_SSF2	Dynamic information	Number of loans and number of penalties
	Static information	The labels “Loans” and “Penalties”
	Message visualization	Textually with Arial, black. Size 10
UW_SSF3	Service selection	Lend book
	Condition to disable	When the member is penalized
	Descriptive text	“The member is currently penalized”

1. UW_SSF1 (*Inform about the success or the failure of an execution*): Each service of the system must inform whether or not the execution finished successfully with a textual message. For example, if the system does not support this Use Way, when the librarian tries to lend an inexistent book, she/he will not notice about the reason of the system failure. However, including UW_SSF1, when the librarian tries to lend an inexistent book, the system displays an interface similar to Fig 1, detailing information about the reason of the mistake. According to Table 1, when the services specified in the property *Service selection* fail in the execution, an error message is displayed such as *Message visualization* indicates.

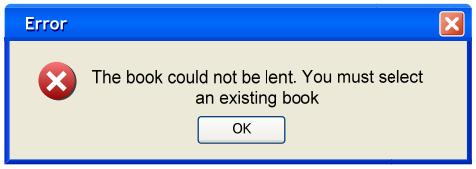

Fig. 1. Error derived from UW_SSF1

2. UW_SSF2 (*Show the information stored in the system*): When the librarian queries the list of members, she/he can choose a member and navigate towards the list of her/his current loans and towards the list of her/his penalties (buttons Loans and Penalties in Fig 2). The librarian would like to see how many loans and penalties the member has without performing the navigation. This functionality is supported with UW_SSF2, which provides dynamic aliases for the navigation buttons by means of the property *Dynamic information*. Fig 3 shows the navigations buttons after including UW_SSF2 in the

system. According to Table 1, navigation buttons aliases are composed of *Static information* (Loans and Penalties) together with *Dynamic information* (the number of loans and penalties for the selected member), and they are displayed such as *Message visualization* indicates.

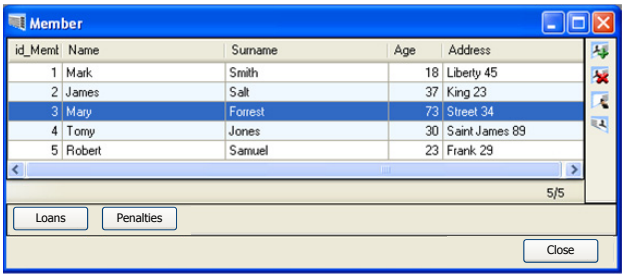

Fig. 2. List of members with navigations towards Loans and Penalties

Fig. 3. Navigation buttons with dynamic aliases

3. UW_SSF3 (*Show the state of relevant actions*): If this Use Way is not included in the system, when the user tries to execute a service that cannot be triggered in the current system state, the execution results in a failure. For example, when the librarian executes the service to lend a book to a member that is currently penalized, the execution will finish unsuccessfully. Including UW_SSF3, the button to lend a book is disabled if the member is penalized, avoiding the librarian to make a mistake. Fig 4 shows an example of an interface that has disabled the service to create a new loan, since the selected member is currently penalized. According to Table 1, if the *Service selection* (Lend book) has a *Condition to disable* (when the member is penalized) that becomes true, the service is disabled and a *Descriptive text* ("The member is currently penalized") is displayed.

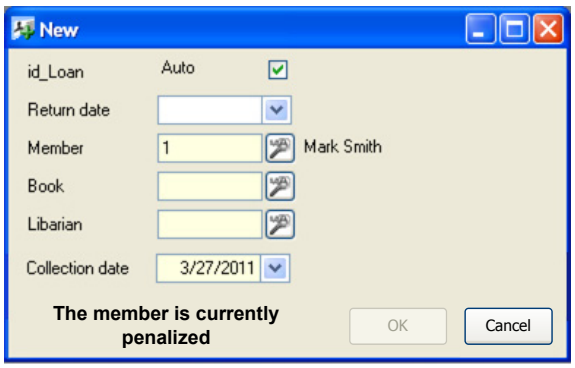

Fig. 4. Disabling the button to execute an action when it cannot be triggered

The main goal of our work is to demonstrate that these aspects related to usability improvement can and should be included in the conceptual schemas that are used in MDD environments, what is often just ignored. We need thus incorporate the corresponding set of conceptual primitives for that purpose. 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 constructed. There are two types of packages in our Usability Model:

- First, for each Use Way, the analyst must define a package that groups 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*. Fig 5 shows an example to represent UW_SSF2.
- Second, the analyst must define inside each package Use Way, 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*. Fig 5 shows an example of two interfaces: *Member* and *Loans*.

Once we have defined the packages, the next step in our proposal is to define elemental primitives inside them:

- 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. For example, in Fig 2 the librarian can navigate towards the list of loans and penalties for a selected member. We propose specifying these navigations by means of an arrow with a source and a target. Fig 5 represents the primitives to define the navigation from member to loans displayed in Fig 2.

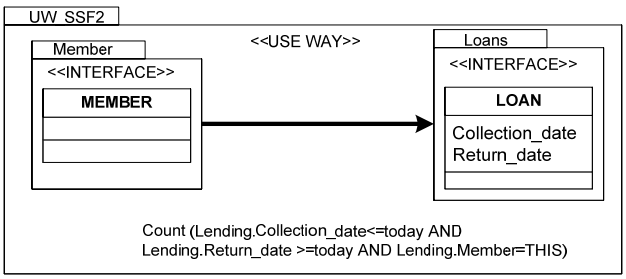

Fig. 5. Primitives to model UW_SSF2 in the library management system

- 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. Fig 5 shows the attributes of the class *Loan* used in the

definition of the dynamic alias formula (explained in the next step).

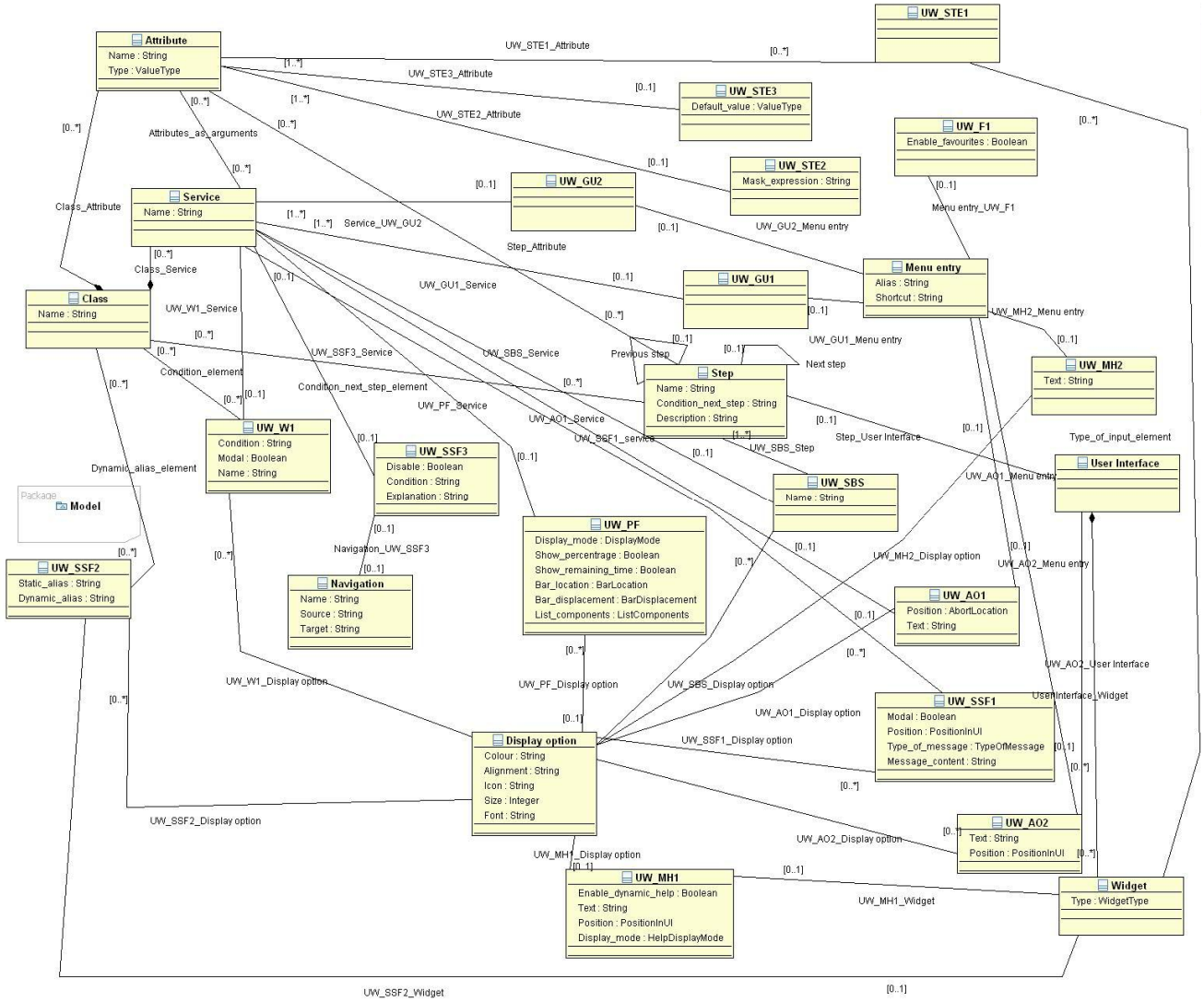

Fig. 6. The Metamodel of the Usability Model

- Third, the analyst must define **formulas** needed in Use Ways with properties that use conditions or dynamic information. These formulas are represented textually. For example, the formula to describe the dynamic alias used in the navigation towards Loans in Fig 2 has been textually defined in Fig 5.

Table 2. Example of UsiXML to represent how the error window will be displayed

```
<window id="1" name="Error_window" width="400" height="158">
<outputText id="M1" name="Error_message" isVisible="true" isEnabled="true" isBold="true" textColor="#000000" value="The book could not be lent. You must select an existing book"/>
<imageComponent id="I1" name="Error_Icon" />
<button id="OK1" name="Ok_Button" "/>
</window>
```

- 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 [13] (USer Interface eXtensible Markup Language), an XML-based markup language for defining user interfaces. Table 2 shows a piece of code of UsiXML to express how the window of Fig.1 is displayed.

These elements compose the suite of conceptual primitives used in the Usability Model. They are enough to represent any of the 22 Use Ways extracted from Juristo's FUFs. A description of the primitives needed to represent each Use Way is detailed in [14].

We have defined a usability metamodel to specify the properties of the Use Ways and how these properties are related to system functionality. The aim of the metamodel is to identify the elements needed to represent all the

properties. All these elements can be represented with the conceptual primitives described above. The usability metamodel is drawn in Fig 6 (an also downloadable from [14]), where each Use Way is represented with a class with the prefix UW. Attributes and relationships represent properties of the Use Ways. For example, the class UW_SSF2 that represents the Use Way *Show the information stored in the system*, represents the properties *Static information* and *Dynamic information* by means of two attributes. The property *Message visualization* is represented with the relation to the class *Display option*, which is a class to represent how the visual elements will be displayed in the interface. It is important to note that in the metamodel we can see how the properties are involved in the functional and interaction features. For example, in UW_SSF2, the property *Dynamic information* depends on the information stored in the system (which is a functional feature), therefore, we need to relate the class that represents UW_SSF2 to the class that represents the system persistency, called Class in the metamodel. Moreover, there is a relationship with the class Widget, since the navigation alias (linking *Static information* and *Dynamic information*) is displayed in a button (which is an interaction feature).

Depending on the MDD method where we would like to include usability features, some of the primitives that compose the Usability Model can already be supported by the existing conceptual model of the MDD method. For example, if the MDD method has a model to represent the class structure, the primitives Attributes and Services are already supported. Next section explains how the Usability Model can be included in an existing MDD-based approach without affecting the models. In a first step, we propose extracting the information represented in existing primitives by means of model-to-model transformations. In a second step, the analyst models unsupported primitives in the Usability Model.

THE USABILITY MODEL IN AN INDUSTRIAL MDD METHOD: A LAB CASE

The Usability Model in a Holistic Method

This section explains how to integrate the Usability Model into a holistic MDD method without changing its existing conceptual model. As we commented above, by holistic we mean that all the relevant systems views (static, dynamic, interaction) are properly incorporated into the used modelling strategy. Fig 7 represents graphically a summary of the process. In the example, there are two existing models in the conceptual model of the MDD method: one model to represent the system persistency and another model to represent the interaction. In this example, we have depicted only two models, but the number of models that the MDD method uses depends exclusively on the chosen MDD method. Moreover, the existing MDD method can support code generation from the conceptual model by means of a model compiler. The level of automation of this process also depends on the MDD method chosen. Some methods are automatic (generate full functional systems), and others semi-automatic (some manual implementation).

Our proposal to include the Usability Model in a holistic MDD method consists of three steps:

1. Derivation of conceptual primitives defined in the existing conceptual model: The notation of the Usability Model includes functionality, persistency, navigation and interaction elements that can be defined in other models of the MDD method (depending on the expressiveness of the MDD method chosen). For example, in Fig 7, the classes, attributes, and services needed in the Usability Model have been previously defined in the Class Model. 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 primitives defined in other models and include them in the Usability Model (model-to-model transformations).

   We propose performing these transformations with ATL [3], which is a language to specify transformations using a source metamodel and a target metamodel. In the example of Fig 7, the source metamodels are the metamodel of the Class Model and the metamodel of the Task Model, while the target metamodel is the metamodel of the Usability Model.

2. Modelling unsupported conceptual primitives: Once supported primitives have been automatically derived, the analyst must manually specify properties that are not supported by existing models.

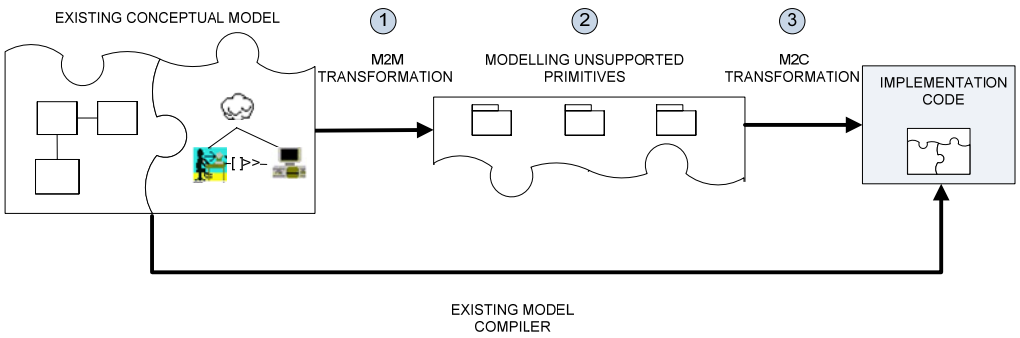

Fig. 7. An overview of the process to integrate the Usability Model in a holistic MDD method

3. Code generation: Once the Usability Model has been fully defined, we can generate code from this model by means of automatic Xpand transformations [28] (model-to-code). The code generated from the Usability Model can be combined with the code generated by the model compiler, which generates code from the existing conceptual model. In the end, the final code includes all the elements specified in the existing conceptual model and in the Usability Model.

It is important to mention that ATL and Xpand transformations must be defined once only for a specific MDD method since these transformations based on metamodels are valid for developing any system. Therefore, steps 1 and 3 can be executed automatically by means of transformation templates.

A Lab Case with OO-Method

In order to demonstrate that our proposal works for a real software development process, we have used OO-Method [19]. OO-Method has been successfully implemented in industry with a tool called OLIVANOVA [5], which can generate full functional systems automatically from a conceptual model. This is the reason why we have chosen OO-Method as proof of concept for our proposal. The OO-Method conceptual model is composed of four complementary models:

1. **Object Model**: Specifies the system structure in terms of classes of objects and their relations. It is modelled as an extended UML [26] class diagram.
2. **Dynamic Model**: Represents the valid sequence of events for an object.
3. **Functional Model**: Specifies how events change object states.
4. **Presentation Model**: Represents the interaction between the system and the user [16]. This model represents the interface by means of Interaction Units. Moreover, this model represents Elementary Patterns that will be displayed inside the interfaces, such as masks, filters, or navigations, among others.

With regard to the system used in the lab case, we use the system to manage a library. This example is simple enough to facilitate the understanding of our proposal. Fig 8 shows the OO-Method Object Model of the system.

The other three models that compose the conceptual model of OO-Method (Dynamic, Functional and Presentation) are not displayed for space reasons. Next, we explain how to model UW_SSF1 (*Inform about the success or the failure of an execution*), UW_SSF2 (*Show the information stored in the system*) and UW_SSF3 (*Show the state of relevant actions*) for developing the library management system in OO-Method. The first step of our proposal consists of extracting information from the existing primitives.

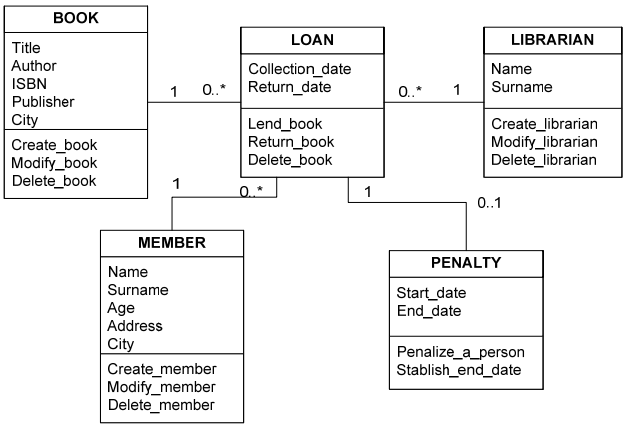

Fig. 8. Object Model of the system to manage a library

From the list of properties of the three Use Ways (Table 1), we can extract from the OO-Method's conceptual model the followings:

- **UW_SSF1**: *Service selection* can be derived from the Object Model.
- **UW_SSF2**: *Static information* of navigations can be derived from the Presentation Model.
- **UW_SSF3**: *Service selection* and *Condition to disable* can be derived from the services of the Object Model and their preconditions. A precondition in the Object Model is a condition that must be satisfied to execute a service.

In order to derive these properties from existing OO-Method models, we have used ATL transformations. The source metamodels are the four metamodels that define the four conceptual models of OO-Method (object, dynamic, functional and presentation); and the target metamodel is the metamodel of the Usability Model. Next, we show the ATL transformation rule that generates a first version of the Usability Model to represent UW_SSF1 when the service execution finishes with a failure. From the two properties, we can only derive *Service selection* from the Object Model since OO-Method does not have any model to represent *Message visualization* (the Presentation Model does not have primitives to represent this property). The ATL rule is simple in order to be as illustrative as possible and avoid technical terms. We have defined similar rules to extract supported primitives of UW_SSF2 and UW_SSF3.

```
rule Service2UWSSF1Failure{
        from
        b: Object!Service
        to
        e: Usability!Service (Name <- b.Name)        }
```

Once we have extracted the properties supported by the OO-Method's conceptual model, the second step of our proposal is to complete the Usability Model with

unsupported conceptual primitives. Next, we detail how to complete each Use Way in the Usability Model. Primitives that are automatically extracted from existing models are drawn on grey background in the figures. Primitives added manually are drawn on white background.

For UW_SSF1, the property *Service selection* has been extracted from the Object Model automatically (in the first step). If the users want to visualize failure messages for the service Lend_book like Fig 1, we must model the property *Message visualization* with the values shown in Table 2. Fig 9 shows the Usability Model for representing UW_SSF1. The property *Service selection* is represented with the primitive Service and the property *Message visualization* with the primitive Display. The primitives Use Way and Interface are generated from scratch in the ATL transformation.

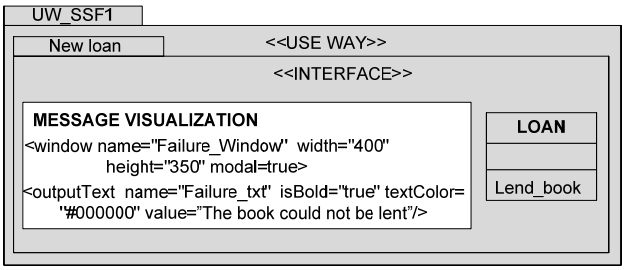

Fig. 9. Model to represent UW_SSF1

The second usability feature that must be included in the library system is to provide dynamic labels in navigation buttons to display how many loans and penalties a selected member has (UW_SSF2). As we have commented above, navigations, their static aliases and interfaces can be extracted from the OO-Method Presentation Model (in the first step). In the second step, the analyst must specify the properties *Dynamic information* and *Message visualization* since these elements are not supported by the OO-Method Presentation Model. Fig 10 shows the Usability Model for UW_SSF2. The property *Static information* is represented with the primitive Navigation and Interface, *Dynamic information* is represented with the primitives Formula and Attributes, and *Message visualization* is represented with the primitive Display. The primitive Use Way is generated from scratch in the ATL transformation.

The third usability feature to include is for disabling the service Lend_book when the member currently has a penalty (UW_SSF3). The properties *Service selection* and *Condition to disable* have been extracted from preconditions of the Object Model (in the first step). The aim of the preconditions is to trigger an error if the user tries to execute a service when a condition is not satisfied. Therefore, the definition of these preconditions can be used to know when to disable the service in order to avoid a user mistake. In this second step, the analyst must only specify the descriptive text that will be shown when the service is disabled (property *Descriptive text*). Fig 11 shows the model to disable *Lend_book* when the member currently has a penalty. The property *Service selection* is represented with the primitive Service, *Condition to disable* with the primitives Formula and Attributes and *Descriptive text* with the primitive Display.

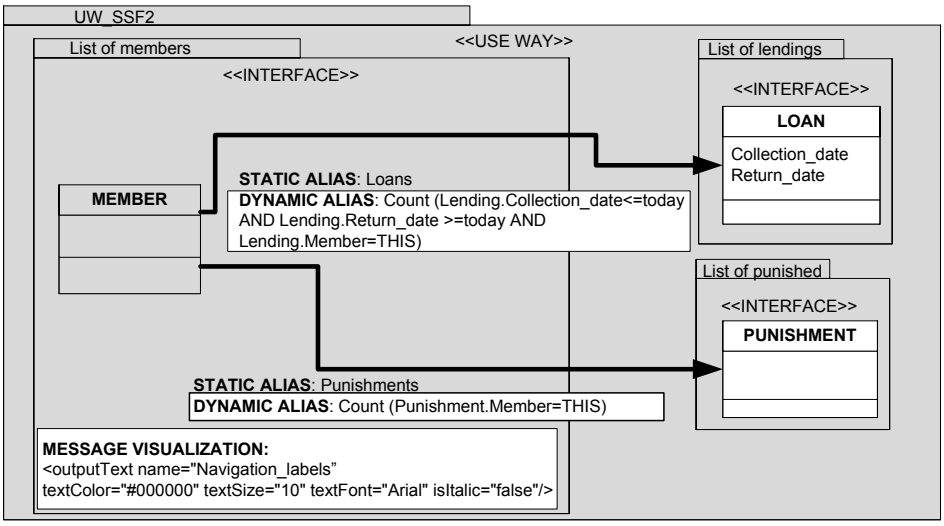

Fig. 10. Model to represent UW_SSF2

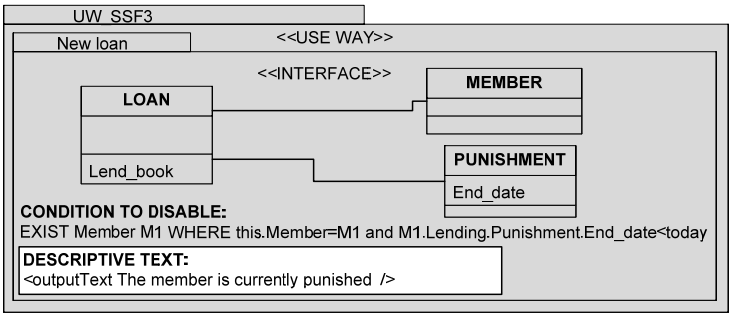

Fig. 11. Model to represent UW_SSF3 to disable Lend_book

Finally, in the third step, the Usability Model must be transformed into code that implements all the characteristics represented in it. This transformation is performed with Xpand [28]. The code derived from the Usability Model must be included in the code generated with the OO-Method model compiler. Below, we show a small chunk of Xpand code used in the transformation from UW_SSF1 into Java code.

```
«DEFINE javaClass FOR Class»
        «FILE Class.name+".java"» public class «name»
{
        «FOREACH service1 AS s»
        DisplayOption show
        public void «s.name»(
                «FOREACH attribute1 AS a»
                «a.type» «a.name»)                    }
        «ENDFOREACH»
        «IF s.uwSsf11 != null» If ("error")
show.display(«s.uwSsf11.modal», «s.uwSsf11.position»,
«s.uwSsf11.typeOfMessage»,
«s.uwSsf11.messageContent»);
        «ENDIF»
        «ENDFOREACH»}
        «ENDFILE» «ENDDEFINE»
```

Fig 1, Fig 3 and Fig 4 show screenshots of the system to manage a library developed with OLIVANOVA after including UW_SSF1, UW_SSF2 and UW_SSF3 respectively. Therefore, we can state that our proposal can be successfully applied to an industrial MDD method.

CONCLUSIONS

The contribution of this paper is the definition of a Usability Model to deal with usability features in a holistic MDD method. We have defined conceptual primitives to represent usability features defined by Juristo for management information systems and we have gathered them in a Usability Model. 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 of other areas 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).

We have learned some lessons applying the proposal to OO-Method: First, the difficulty of writing ATL and Xpand transformations depends exclusively on the MDD method chosen. OO-Method generates the whole system, but MDD methods with less powerful model compilers 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. Second, 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. As future work, we plan to define metrics to measure the usability of the system based on the conceptual primitives of the Usability Model. Moreover, we plan to measure the effort required to implement this approach in an MDD method. This measure will be done considering analysts who know previously FUFs and analyst who do not know them yet.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of the ITEA2 Call 3 UsiXML project (20080026) and financed by the MITYC under the project TSI-020400-2011-20; the MICINN under the project PROS-Req (TIN2010-19130-C02-02) co-financed with ERDF; the Generalitat Valenciana under the project ORCA (PROMETEO/2009/015).

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. Brajnik, G.: Is the UML appropriate for Interaction Design? Università di Udine (2010) 6.
5. CARE Technologies S.A. http://www.care-t.com
6. Davis, F.D.: User acceptance of information technology: system characteristics, user perceptions and behavioral impacts. Int. Journal Man-Machine Studies 38 (1993) 475-487.
7. Fernández, A., Abrahao, S., Insfran, E.: A Web Usability Evaluation Process for Model-Driven Web Development. 23rd International Conference on Advanced Information Systems Engineering (CAiSE 2011). Springer, London (2011) 108-122
8. Folmer, E., Bosch, J.: Architecting for usability: A Survey. Journal of Systems and Software, Vol. 70 (1) (2004) 61-78.
9. 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.
10. ISO/IEC 9126-1, Software engineering - Product quality - 1: Quality model (2001).
11. 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.
12. Lawrence, B., Wiegers, K., Ebert, C.: The top risk of requirements engineering. IEEE Software, Vol. 18 (2001) 62-63.
13. 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).
14. List of Use Ways and Properties: http://hci.dsic.upv.es/UsabilityModel/UseWaysList.html
15. 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.
16. 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).
17. 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.
18. 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.
19. Pastor, O., Molina, J.: Model-Driven Architecture in Practice. Springer, Valencia (2007).
20. Selic, B.: The Pragmatics of Model-Driven Development. IEEE software 20 (2003) 19-25
21. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software 20 (2003) 42-45.
22. 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.
23. Sousa, K., Mendonça, H., Vanderdonckt, J.: Towards Method Engineering of Model-Driven User Interface Development. TAMODIA, LNCS 4849. Springer, Toulouse (France) (2007) 112-125.
24. 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).
25. Tao, Y.: An Adaptive Approach to Obtaining Usability Information for Early Usability Evaluation. IMECS (2007) 1066-1070.
26. UML: http://www.uml.org/
27. Wang, X., Shi, Y.: UMDD: User Model Driven Software Development. IEEE/IFIP Int. Conference on Embedded and Ubiquitous Computing, Shanghai (China) (2008).
28. XPAND: http://www.eclipse.org/modeling/m2t/?project=xpand