Document downloaded from:

http://hdl.handle.net/10251/65105

This paper must be cited as:

Panach Navarrete, JI.; Juristo, N.; Valverde Giromé, F.; Pastor López, O. (2015). A framework to identify primitives that represent usability within Model-Driven Development methods. Information and Software Technology. (58):338-354. doi:10.1016/j.infsof.2014.07.002.

The final publication is available at

http://dx.doi.org/10.1016/j.infsof.2014.07.002

Copyright Elsevier

Additional Information

A Framework to Identify Primitives that Represent Usability within Model-Driven Development Methods

Jose Ignacio Panach[1], Natalia Juristo[2], Francisco Valverde[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]Universidad Politécnica de Madrid, Campus de Montegancedo,28660, Boadilla del Monte, Spain
natalia@fi.upm.es
[3]Centro de Investigación en Métodos de Producción de Software - ProS
Universitat Politècnica de València, Camino de Vera s/n, 46022 Valencia, Spain
opastor@pros.upv.es

Pre-print version

Abstract. Context: Nowadays, there are sound methods and tools which implement the Model-Driven Development approach (MDD) satisfactorily. However, MDD approaches focus on representing and generating code that represents functionality, behaviour and persistence, putting the interaction, and more specifically the usability, in a second place. If we aim to include usability features in a system developed with a MDD tool, we need to extend manually the generated code. **Objective:** This paper tackles how to include functional usability features (usability recommendations strongly related to system functionality) in MDD through conceptual primitives. **Method:** The approach consists of studying usability guidelines to identify usability properties that can be represented in a conceptual model. Next, these new primitives are the input for a model compiler that generates the code according to the characteristics expressed in them. An empirical study with 66 subjects was conducted to study the effect of including functional usability features regarding end users' satisfaction and time to complete tasks. Moreover, we have compared the workload of two MDD analysts including usability features by hand in the generated code versus including them through conceptual primitives according to our approach. **Results:** Results of the empirical study shows that after including usability features, end users' satisfaction improves while spent time does not change significantly. This justifies the use of usability features in the software development process. Results of the comparison show that the workload required to adapt the MDD method to support usability features through conceptual primitives is heavy. However, once MDD supports these features, MDD analysts working with primitives are more efficient than MDD analysts implementing these features manually. **Conclusion:** This approach brings us a step closer to conceptual models where models represent not only functionality, behaviour or persistence, but also usability features.

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

1 Introduction

The Model-Driven Development (MDD) paradigm [20] states that all the analysts' effort must be gathered in the conceptual model and the system is implemented by means of transformation rules that can be automated. In other words, the MDD paradigm distinguishes between conceptual models (where analysts work) and the code that implements the system (which can be generated with as much automation as possible from the conceptual model).

Nowadays, there are several tools which implement the MDD paradigm, such as WebRatio [2], UWE [19], NDT [9] and OO-Method [29][28], among others. All these tools are very powerful to represent and generate the system functionality, behaviour and persistency by means of conceptual models. However, in most MDD methods, there is a lack of expressiveness to represent usability features [1][24]. Nowadays, if these features are to be included in systems developed by these MDD methods, the generated code needs to be changed manually. These manual changes involve some disadvantages:

- Changes in the code can be inconsistent with the characteristics expressed in the conceptual model.
- Every time we regenerate the code from the conceptual model, the manual changes to the code must be applied.
- Understanding the code to enhance the system usability can be difficult for the analyst.

In order to overcome all these problems, we propose including usability features in a conceptual model similarly to what it is currently done with functionality, behaviour and persistency in most MDD methods [18][34]. This proposal is a step forward to incorporate software systems characteristics not combined to date in MDD methods. Note that the target audience of our proposal are analysts that work frequently with MDD tools, since they are the persons that tweak the code to support usability features nowadays. Our approach does not deal with benefits or disadvantages of the MDD paradigm versus a traditional method or how to improve the learnability of novice users with MDD tools.

In the past, many SE authors considered usability as a non-functional requirement [7]. Recently, however, some authors have identified several usability features that are strongly related to functionality [4][11][16]. We focus on these features, since they affect not only interface but also the architecture, and are hard to deal with unless they are considered from the early stages of development. The contribution of our work is the definition of a process to represent functional usability features in a conceptual model in such a way that a model compiler can automatically generate their code.

The benefits of incorporating functional usability features in a MDD method through conceptual primitives are [35][36]:

- Unambiguously defined functional usability features. This is an essential characteristic for performing model-to-model and model-to-code transformations.
- Reduced development effort with respect to including usability features by hand, since functional usability features are added to the system code by a model compiler.
- Evolutions of usability requirements need to be applied to the conceptual model only. Therefore, system will be able to evolve more easily.

Our proposal to include usability features is valid for any MDD method. However, it has been necessary to select a specific MDD method to fully define our proposal. We have chosen OO-Method [29][28], since it is supported by a commercial tool that is being

Pre-print Version

regularly used to develop real systems by a company (INTEGRANOVA) [6]. Such MDD tool generates fully functional systems from a conceptual model. Another advantage of the MDD method used as benchmark for our research is that its conceptual model is abstract enough to straightforwardly add new primitives that represent usability features.

This paper is the ongoing work of two previous publications: [25] and [26]. [25] offers a first draft of the idea to represent functional usability features in a conceptual model. The contribution of this paper with regard to the previous one consists of: (1) A more detailed definition of the procedure to include functional usability features in a conceptual model; (2) A proof of concept with different usability features in a real MDD tool. [26] is a poster that introduces a short description of an experiment to analyze the benefits of including functional usability features in a system. The contribution of this paper with regard to the previous one consists of: (1) an exhaustive description of the design, threats and results of the experiment to know whether or not users' satisfaction and users' efficiency improves after including functional usability features in the systems; (2) a comparison of effort to include functional usability features in a MDD method manually with the effort to include them through conceptual primitives.

The paper is structured as follows. Section 2 introduces the usability and MDD background necessary to understand our proposal. Section 3 describes our proposal for adding usability features to a MDD method. Section 4 illustrates the application of our proposal to a specific MDD method. Section 5 discusses an experiment to evaluate user satisfaction improvement applying our proposal. Section 6 studies the improvement of the efficiency of analysts working with functional usability features represented as conceptual primitives versus including them manually. Section 7 describes related work. Finally, Section 8 presents some conclusions.

2 Background

The **MDD paradigm** aims to develop software using a conceptual model that abstractly represents the system under development [20]. This conceptual model is the input for a model compiler that generates the code implementing the system. Usually, this generation is performed by transformation rules that are applied automatically. A MDD conceptual model is divided into different views or models. View stands for the set of formal elements that describe something that has been built for a purpose. For example, there can be a view to represent the user interaction, another view to represent system functionality and another view to represent information persistence. Views are composed of conceptual primitives. Conceptual primitives are modelling elements that have the capability of abstractly representing an aspect of the system. Examples of conceptual primitives are class diagram classes, class attributes and services, etc. The system is generated from the conceptual model by a model compiler. The level of automation for code generation is more or less powerful depending on the MDD method.

Usability is a very broad concept. According to ISO 9241-11 [14], usability is *"the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specific context of use"*. Human-Computer Interaction (HCI) literature provides many different recommendations to improve software system usability. HCI recommendations can be classified into three groups [16]:

Pre-print Version

- Usability recommendations with impact on the user interface (UI). They refer to presentation issues which imply slight modifications of the UI design (e.g. buttons, pull-down menus, colours, fonts, layout).
- Usability recommendations with impact on the development process. To follow these advices the development process needs to be tuned. For example, recommendations designed to reduce the user cognitive load state that software development should implicate users.
- Usability recommendations with high impact on architectural design. They involve building certain functionalities into the software in order to improve user-system interaction. This set of usability recommendations are referred to as functional usability features (FUF). Examples of such features are cancel, undo and feedback facilities. Unless these features are considered from the early stages of the software development process, it takes a lot of rework to build them into a software system [4]. We focus our approach on this group of recommendations.

Table 1 shows a summary of FUFs, the mechanisms into which they are divided and their goals. We have selected four mechanisms to illustrate here our approach (shaded in grey in Table 1). This choice is based on the usefulness of the mechanisms for the examples used in this paper.

Table 1. List of FUFs and their mechanisms [17]

Usability Feature	Usability Mechanism	Goal
Feedback	System Status	To inform users about the internal status of the system
	Interaction	To inform users that the system has registered a user interaction, i.e. that the system has heard users
	Progress	To inform users that the system is processing an action that will take some time to complete
	Warning	To inform users of any action with important consequences
Undo Cancel	Global Undo	To undo system actions at several levels
	Abort Operation	To cancel the execution of an action or the whole application
User Input Error Prevention	Structured Text Entry	To help prevent the user from making data input errors
Wizard	Step-by-Step	To help users to do tasks that require different steps with user input and correct such input
User Profile	Preferences	To record each user's options for using system functions
	Personal Object Space	To record each user's options for using the system interface
	Favourites	To record certain sites of interest for the user
Help	Multilevel Help	To provide different help levels for users

Pre-print version

As shown in [17], a full description and elicitation guidelines for each and every FUF can be found at http://www.grise.upm.es/sites/extras/2/. FUFs were derived from interaction patterns described in the literature as [40][42][31]. FUFs contribute a detailed description of how usability features affect the system architecture, whereas interaction patterns only define how usability features affect the system interface. Another difference between FUFs and interaction patterns is that FUFs are defined with a terminology that can be understood easily by end users. In contrast, interaction patterns are usually more oriented for analysts.

FUFs are expected to be incorporated into the development process as functional requirements, since usability features that are properly described in the requirements specification are more likely to be successfully built into the system [11]. As an aid for analysts, the FUF definition provides guidelines [17] for capturing FUFs requirements and designing the system. Once FUFs have been incorporated into requirements (following guidelines), they are manually designed and implemented.

3 Incorporating Usability Functionalities into a Model-Driven Development Method

Our approach for incorporating FUFs into a MDD method is divided into four steps, as Fig 1 shows:

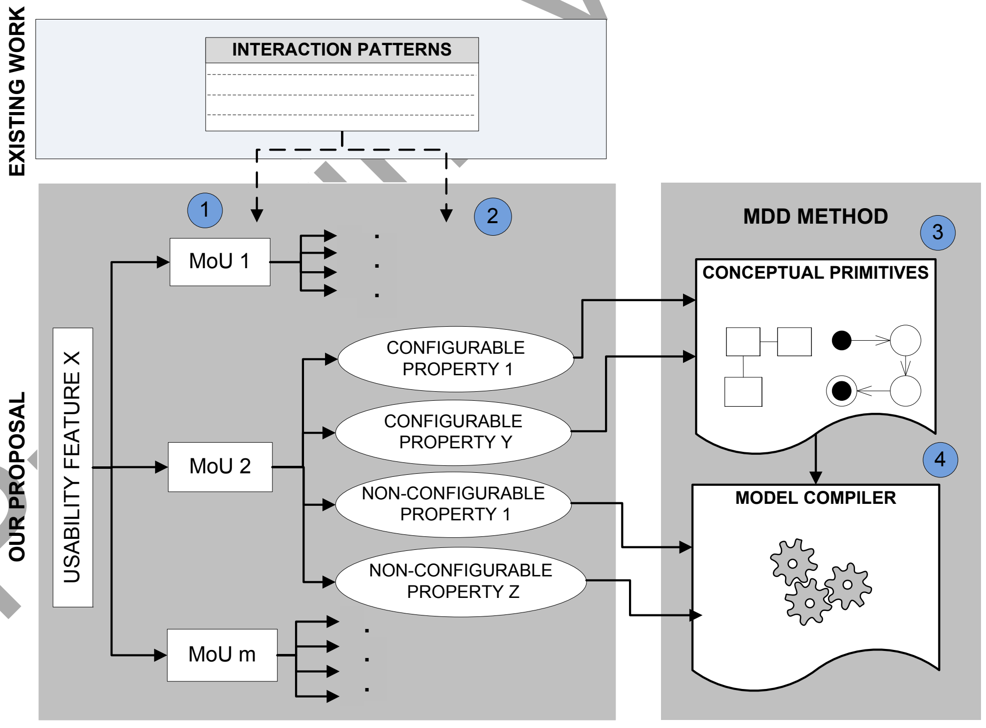


Fig 1. The four steps of our proposal

Preprint Version

1. Identify the possible *modes of use* of each usability functionality.
2. Identify the *properties* that configure each mode of use with regard to usability requirements.
3. Define *conceptual primitives* to abstractly represent the mode of use properties.
4. Describe the changes that must be made to the *model compiler* to implement the identified properties.

The first and second steps are based on earlier research defining how to deal with FUFs: interaction patterns, usability guidelines, usability heuristics or any other research defining how to build usability features into a software development process. The third and fourth steps depend on a specific MDD method. We focus on OO-Method [29][28] as illustrative example of MDD method.

With our proposal, an analyst can ensure that functional usability features will be included in systems. What analysts need to do is select the usability features to include in the system and select some feature parameters. Note that in the same way as with a manual implementation, our approach based on primitives does not ensure usability improvements in every system. These improvements depend on how the analyst models usability features according to the context of use. In the following, the four steps of our approach are explained in detail.

3.1 Identification of Modes of Use

The first step for incorporating a usability feature into a MDD method is to identify its modes of use. Each functional usability feature can achieve its goal by different means, which we have termed **Mode of Use (MoU)**. Each MoU achieves a specific target, which is part of the overall goal of the usability feature. Different MoUs that are part of the same usability feature target the same overall goal without conflicting each other.

For example, the usability mechanism *System Status Feedback* (from the feature *Feedback)* aims to inform the user about the internal system state [8][40]. Using the information provided by interaction patterns, we have identified that this goal can be achieved by at least three modes of use: (1) *Inform about the success or failure of an execution (MoU1)*; (2) *Display the information stored in the system (MoU2)*; (3) *Display the state of relevant actions (MoU3)*. The first MoU is derived from the interaction pattern called Modeless Feedback Area [8], which aims to provide feedback that the program has accepted the command for every action the user takes. The second and the third MoUs are derived from the interaction pattern called Status Display [40]. This pattern aims to monitor the state of something that changes. Note importantly that even though the last two MoUs were generated from the same interaction pattern, the goal of each MoU that we have generated is different. The second MoU aims to display the state using information stored in a repository, whereas the third MoU is designed to display the state by indicating which actions can be triggered at any time.

MoUs can be generated from the information contained in the FUF elicitation guidelines [17]. For each FUF question (a total of 62) we have needed to consider all possible ways of achieving the usability goal established by the guidelines. We have obtained 22 MoUs valid for incorporating the six FUFs shown in Table 1 into any MDD method. The 22 MoU can be found in [27] and are easily accessible at [23].

Pre-print Version

3.2 Identification of Properties

The second step to deal with functional usability features through the MDD paradigm is to configure the identified MoUs. We refer to the different MoU configuration options for satisfying usability functionalities as **properties**. In this second step, we also identify properties from interaction patterns.

For example, *Inform about the success or failure of an execution (MoU1)* is composed of two properties extracted from the Modeless Feedback Area pattern [8]: (1) *Service selection* and (2) *Message visualization*. The first property is derived from the description of the interaction pattern that states that every action must inform about its success or failure. The second property is defined because the pattern also states that how and where the information is to be displayed needs to be specified.

In some cases, analysts need to adapt properties to the system under development. In other cases, properties can be configured automatically without any intervention by analysts. Therefore, MoUs have two types of properties:

- **Configurable properties,** which require an analyst to make decisions about how they are to be configured. Based on user requirements, the analyst specifies the most suitable configuration for these properties. For instance, the Message visualization property from MoU1 is configurable because for a specific system the analyst needs to specify how the information will be displayed according to user preferences. *Display the information stored in the system (MoU2)* has three configurable properties extracted from the interaction pattern called Status Display [40]: (1) *Dynamic information to show,* (2) *Static information to show* and (3) *Message visualization.* The first property is derived from the description of the pattern that states that the system must display information about the status that is likely to change over time. The second property is derived from the need of information that remains constant for each interaction. The third property is defined to allow specifying all the visualization possibilities claimed in the pattern description. *Display the state of relevant actions (MoU3)* has three configurable properties extracted from pattern Status Display [40]: (1) *Actions selection,* (2) *Condition to disable* and (3) *Descriptive text.* The first property is derived from the need of the pattern to specify the actions that require to display their state. The second property is defined to specify the condition to disable. The third property is derived from the description of the pattern that recommends displaying a descriptive text when the action has been disabled.
- **Non-configurable properties,** which have an unchanging configuration for all systems. For example, the Service selection property from *Inform about the success or failure of an execution (MoU1)* is non-configurable because the ergonomic Immediate Feedback criterion [5] states that the system must report the success or failure of an action at the end of each execution. We propose that the MDD method model compiler is responsible for including non-configurable properties in generated systems. This approach improves efficiency since the model compiler automatically or semi-automatically includes non-configurable properties in the system without analyst intervention.

As for MoUs, we have had to work out properties from FUF elicitation guidelines. We have generated 57 properties valid for any MDD method. All the 57 properties can be found in [27], and they are easily accessible through [23]. Of the 57 properties identified from the FUF list in Table 1, 50 are configurable and 7 are non-configurable.

Pre-print version

3.3 Definition of Conceptual Primitives

We propose defining configurable properties through conceptual models of a MDD method. The third step of our approach involves verifying whether or not there are already **conceptual primitives** in the MDD method representing a configurable property. If no conceptual primitive has been set up to represent a configurable property or existing conceptual primitives are unable to represent some configuration options, the conceptual model needs to be expanded with new conceptual primitives that ensure the required expressiveness. Note that how each configurable property is represented in the conceptual model depends exclusively on the chosen MDD method; there are as many conceptual models as MDD methods.

As illustrative example, Table 2 shows the primitives needed to represent all the properties derived from the usability mechanism System Status Feedback. Each primitive is used to specify a system characteristic.

Table 2. Necessary primitives to represent properties of System Status Feedback

Mode of Use	Property	Needed primitives
MoU1	Message Visualization	-Define an error message text -Define a success message text -Define if the message is textual or graphical -Define the message format -Define icons that indicate error or success
MoU2	Dynamic information to show	Define a formula that specifies how to obtain the dynamic information
	Static information to show	Define static text
	Message visualization	Define the format that displays both dynamic and static information
MoU3	Actions selection	Define what actions can be disabled
	Condition to disable	Define the formula that disables an action
	Descriptive text	Define the text that explains the reason of disabling an action

This step is method dependent and one single solution cannot be provided for any existent MDD method. For the 50 configurable properties generated in the second step of the process, we have identified 68 primitives needed to support all the configurable properties (accessible through [23]).

3.4 Description of Changes in the Model Compiler

Finally, in fourth step, the **model compiler** needs to be modified in order to make it able to deal with new conceptual primitives and non-configurable properties. This step also depends on the MDD method since the model compiler is method specific but again solutions are similar between MDD methods. So the solution we show here is useful to guide changes for any MDD method. The changes needed in the model compiler are:

- **New conceptual primitives**: The model compiler must have the capability to recognize and generate the code that implements the new conceptual primitives

Pre-print version

(generated in step three) according to the configuration represented in the conceptual model.

- **Non-configurable properties**: Although these properties do not imply changes to the conceptual model, they do affect the model compiler. The model compiler must build the functionality of non-configurable properties into the generated code without analyst participation.

As illustrative example, Table 3 shows an overview of the necessary code to implement all the properties derived from System Status Feedback. Note that even non-configurable properties, such as *Service selection*, involve some lines of code for their implementation.

Table 3. Necessary code to implement properties of System Status Feedback

Mode of Use	Property	Needed code
MoU1	Service selection	Report the results after executing an action
	Message Visualization	-Display an error message when an action fails -Display a success message when an action finishes -Display all messages according to the characteristics defined with primitives
MoU2	Dynamic information to show	Calculate and display the dynamic information
	Static information to show	Display static text
	Message visualization	Display information according to the characteristics defined with primitives
MoU3	Actions selection	Allow to disable actions
	Condition to disable	Disable an action when a condition is satisfied
	Descriptive text	Display text that describes the reason for disabling an action

Notice that MoUs and properties (steps 1 and 2) can be used for any MDD method. Changes to the conceptual model and to the model compiler (steps 3 and 4) are MDD method specific since every MDD method has its own conceptual primitives and its own transformation rules. However, solutions provided to one specific method are analogous to those needed for a different MDD method. For the 57 properties generated in the second step of the process, we have identified 68 characteristics to implement through code (accessible through [23]).

4 Proof of Concept

We have selected OO-Method [29][28] as the specific MDD method to be used to validate our proposal. INTEGRANOVA [6] is a commercial tool which implements OO-Method that can generate code in Java, C# and ASP.NET. Code is automatically generated by INTEGRANOVA from a conceptual model using a model compiler. The company INTEGRANOVA makes business using the tool INTEGRANOVA to develop software systems to be used in the real life. OO-Method conceptual model is composed of four complementary models (or views):

Pre-print Version

- **Object model**, which specifies the system structure in terms of classes of objects and their relations. It is modelled as an extended UML [39] class diagram. A class is based on attributes and services.
- **Dynamic model**, which represents the valid sequence of events for an object. It is modelled as a UML statechart diagram.
- **Functional model**, which specifies how events change object states.
- **Interaction model**, which represents the interaction between the system and the user. It has two views: (i) the **Abstract Interaction Model** [22], which defines the interface without taking into account definite visualization features, representing the interface independently of the interaction types and the platform features; and (ii) the **Concrete Interaction Model** [3], which specifies details of the interface in terms of elements that end users can perceive. The Abstract Interaction Model is structured through interaction patterns divided into three levels:
  - Level 1 - Hierarchical Action Tree (HAT): organizes the access to the system functionality.
  - Level 2 - Interaction Units (IUs): represent the main interactive operations that can be performed on objects. There are three types of IUs: *Service Interaction Unit (SIU)*, which represents a form to execute a service; *Population Interaction Unit (PIU),* which represents a query of instances from a class; *Instance Interaction Unit* (IIU), which represents details of a specific object.
  - Level 3 - Elementary Patterns (EPs): constitute the building blocks from which IUs are constructed. Through these patterns, we can model: masks for text entry fields (*EP Introduction)*; lists of elements (*EP Defined Selection*); groups of widgets (*EP Argument Grouping*); filter criteria (*EP Filter*); set of elements to display in a table (*EP Display Set*); order criteria for lists (*EP Order Criterion*); actions that the user can trigger (*EP Actions*); navigations among interfaces (*EP Navigation*).

  The Concrete Interaction Model specifies how the elements that compose an interface will be displayed. For example, in this model, the analyst decides the widget to display a Defined Selection, which can be a list box or a radio button. The Concrete Interaction Model is defined through Transformation Templates, which specify the structure, layout and style of an interface according to preferences of end-users and the different hardware and software computing platforms. A Transformation Template is composed of Parameters with associated values which parameterize the different design alternatives of interfaces.

In the following, we use *Structured Text Entry* and *Warning* (Table 1) as usability mechanisms to illustrate how our approach works in OO-Method. Structured Text Entry belongs to the FUF called *User Input Error Prevention,* whose goal is *to help the user when the system only accepts inputs in a specific format.* Warning belongs to the FUF called *Feedback*, whose goal is *to inform users about what is happening in the system.* We select both FUFs because their goals are simple enough for presentation in a couple of pages and both mechanisms are used in our experiment. Moreover, Structured Test Entry is partially supported by OO-Method currently, which is useful to illustrate that some primitives used to represent configurable properties can be already supported by the MDD method.

First step of the proposed procedure is **identification of Modes of Use**. We identify three MoUs for the Structured Text Entry and one for Warning. These MoUs have been derived from the requirements elicitation guidelines of the usability mechanisms [17].

Pre-print Version

Table 4 shows the elicitation requirements questions from which the MoUs have been derived, the goal of the MoUs and their names.

Table 4. Structured Text Entry and Warning MoUs

FUF Question	Goal	MoU
Structured Text Entry		
Which is the format of input arguments?	Specify the format of the input widget to help the user	Specify the input widget visualization type (MoU1_STE)
What guidance should the user receive to enter the input in the required format?	Stop the user from entering data that is not in a valid format	Mask definition (MoU2_STE)
What guidance should the user receive to enter the input in the required format?	Provide the user with guidance on which format to use to enter data	Default values (MoU3_STE)
Warning		
Which requested services have irreversible consequences?	Warn the user about the consequences of executing a service	Warning message (MoU1_W)

Table 5.Properties of MoU1_STE, MoU2_STE, MoU3_STE and MoU1_W

Question	Goal	Name
Specify the input widget visualization type (MoU1_STE)		
Which is the format of input arguments?	Define how the user will visualize input arguments	Type of input widget (P1_MoU1_STE)
Mask definition (MoU2_STE)		
Which widgets require a specific format for their data?	Specify the widgets that need a mask	Widget selection (P1_MoU2_STE)
Which is the required format for the widget?	Define the regular expression that defines the mask	Regular expression (P2_MoU2_STE)
Default values (MoU3_STE)		
Which widgets require a default value?	Specify the widgets that need a default value	Widget selection (P1_MoU3_STE)
Which is the required default value?	Define the default value	Default value definition (P2_MoU3_STE)
Warning message (MoU1_W)		
Which tasks require a confirmation?	Specify the services that need a warning before its execution	Service selection (P1_MoU1_W)
When does the system show the confirmation?	Define the condition to display the warning message	Condition definition (P2_MoU2_W)
Which information is provided to confirm?	Define how the user will visualize the warning message	Message visualization (P3_MoU3_W)

Pre-print Version

Next step is **identification of properties** for each MoU. We derive properties from FUF requirements elicitation guidelines. Table 5 shows the properties that we have identified from FUF definition. They are all configurable properties since analysts need to specify what setup users would like. These two first steps of our proposal are independent of the MDD method.

Next step is **definition of conceptual primitives** to identify the required primitives to abstractly represent every configurable property. Whether or not each configurable property is already supported by the MDD method needs to be studied. This task is MDD method dependent. Let us analyze for INTEGRANOVA the properties identified for MoU1_STE, MoU2_STE and MoU3_STE.

Type of input widget (P1_MoU1_STE) is not completely supported by this MDD method. Depending on the argument type, analysts can choose from a restricted list of widgets. For example, if the argument type is a numbered list, analysts can choose between a list box or a text box. However, if the argument is Boolean, the widget will be directly transformed into a check box. But a radio button would be better in some contexts. Therefore, to include *P1_MoU1_STE* property, the OO-Method conceptual model needs to be enriched with new primitives that represent the different widget types. These changes affect the Concrete Interaction Model, which defines visualization features.

Note that the conceptual primitives in the OO-Method Interaction Model (Abstract and Concrete) are defined textually. However, INTEGRANOVA facilitates the definition of these primitives that it displays as widgets to be filled in by the analyst.

Fig 2 shows a prototype modelling the *Type of input widget* property *(P1_MoU1_STE)*. On the left of the window there is a list with all the Service Interaction Units (SIU) defined in the system. Arguments are grouped by the service to which they belong. We select the argument *province* of *Create a client* service as an example. On the right of Fig 2, analysts can choose the type of widget that will visualize the selected argument (Property *P1_MoU1_STE*). The widget types from which analysts can choose depend on the argument type, which should have been defined previously in the existent object model (when classes and attributes constituting the business logic are defined). In the example, the *province* argument type is a numbered list, but this argument type can also be represented by a combo box or radio button.

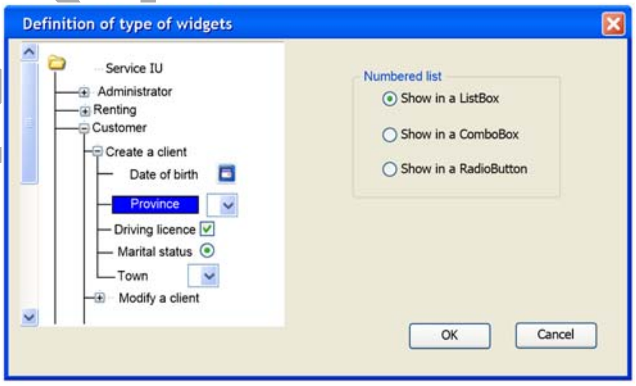
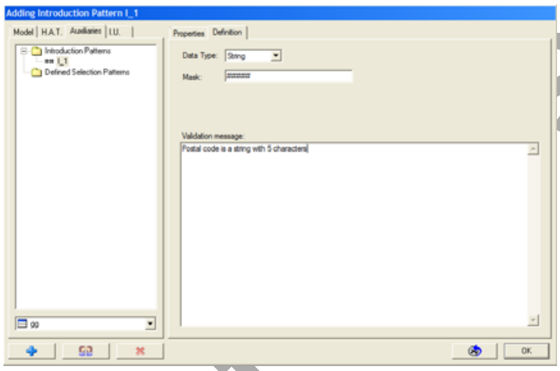


Fig 2. How to model MoU1_P1 with new primitives in the Concrete Interaction Model

Preprint Version

MoU2_STE and *MoU3_STE* are already supported by OO-Method conceptual model and do not therefore require new conceptual primitives. This example is useful to illustrate that some configurable properties can be already supported by the MDD method. In the following, we show how both MoUs are already modelled in INTEGRANOVA with existent primitives. The two properties of *Mask definition (MoU2_STE)* are modelled in the Abstract Interaction Model, where analysts specify the elements of the SIU. The *Regular expression (P2_MoU2_STE)* property is defined in an existent window like Fig 3. In this example, the analyst has defined a mask that accepts a string with only five characters to represent a post code. Next, the analyst has to assign this regular expression to an existing argument. This assignment is the representation of the *Widget selection* property *(P1_MoU2_STE)*.

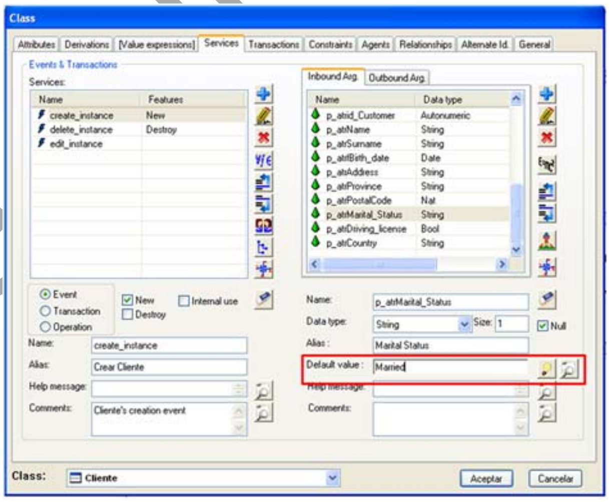


Data Type: String

Mask: #####

Validation message:
Postal code is a string with 5 characters

Fig 3. How MoU2_P2 is already modelled in the Abstract Interaction Model



Services:

Name	Features
create_instance	New
delete_instance	Destroy
edit_instance	

Name	Data type
p_atrid_Customer	Autonumeric
p_atrName	String
p_atrSurname	String
p_atrBirth_date	Date
p_atrAddress	String
p_atrProvince	String
p_atrPostalCode	Nat
p_atrMarital_Status	String
p_atrDriving_license	Bool
p_atrCountry	String

[✓ Event] Transaction Operation

[✓ New] Destroy Internal use

Name: create_instance

Alias: Crear Cliente

Help message:

Comments: Cliente's creation event

Name: p_atrMarital_Status

Data type: String Size: 1 [✓ Null]

Alias : Marital Status

Default value : Married

Help message:

Comments:

Class: Cliente

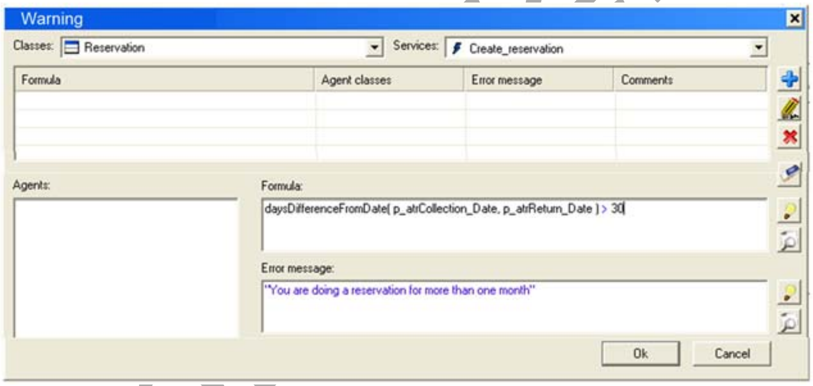
Fig 4. How MoU3_P1 and MoU3_P2 are already modelled in the Object Model

The *Widget selection (P1_MoU3_STE)* and *Definition of the default value (P2_MoU3_STE)* properties are modelled in the existent object model (Fig 4). When

Preprint version

analysts specify the attributes in a class, they can also specify a default value for each attribute. The *P1_MoU3_STE* property is specified by selecting one of the arguments on the right side of Fig 4. After the input argument has been selected, the *P2_MoU3_STE* property is defined in the default value field in Fig 4. The default value must be compliant with the argument type.

Regarding properties of *Warning message (MoU1_W)*, they are not supported by INTEGRANOVA yet. The inclusion of *Service selection (P1_MoU1_W)* involves specifying what services must display a warning message before running. We can add a new primitive within the Object Model to express whether or not each service of a class needs a warning message. The Object Model already supports the definition of services (methods of a class), therefore, it is the most suitable model to define all properties regarding services. The inclusion of *Condition definition (P2_MoU1_W)* needs a primitive to represent when to display a warning message. We propose including a new primitive in the Object Model to define formulas that express when to show the message. Fig 5 shows a prototype modelling P1_MoU1_W and P2_MoU2_W to define a warning message for the service *Create reservation*. The system warns end users before running the service if the period of reservation lasts longer than 30 days.


Classes: Reservation
Services: Create_reservation

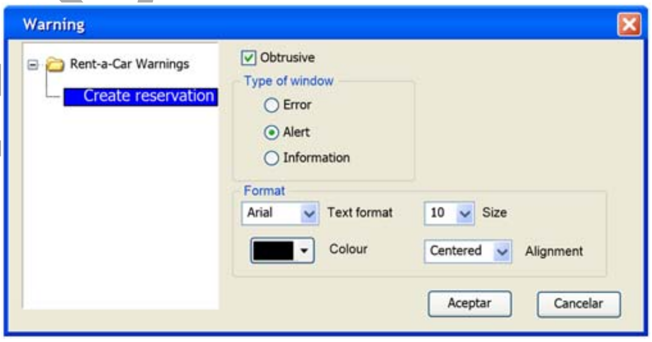
Formula	Agent classes	Error message	Comments

Agents:

Formula: daysDifferenceFromDate(p_atrCollection_Date, p_atrReturn_Date) > 30

Error message: "You are doing a reservation for more than one month"

Fig 5. How to model P1_MoU3_W and P2_MoU3_W with new primitives in the Object Model


Rent-a-Car Warnings
Create reservation

[✓] Obtrusive

Type of window: Error, [✓ Alert], Information

Format: Arial Text format; 10 Size; Colour; Centered Alignment

Fig 6. How to model P3_MoU3_W with new primitives in the Concrete Interaction Model

Preview version

Message visualization (P3_MoU3_W) is focused on display options. The inclusion of this property involves new primitives to represent every visual alternative of the warning message. These new primitives are added to the Concrete Interaction Model, where we can model all display options through design templates. Fig 6 shows an example of prototype that configures visualization alternatives to display the warning message of *Create_reservation.* According to this configuration, the message will be displayed within an obtrusive alert window and the text message will be displayed in Arial, size 10 and centred alignment.

Note that the analyst must specify all the primitives that represent configurable properties before generating the code that implements them. Each primitive is exclusive of a specific configurable property derived from a specific MoU. In order to facilitate the analysts' work, these conceptual primitives can have a default value in case analysts do not want to configure them. Default values should be the most frequently used values. Analysts can change these default values in case they do not satisfy user's requirements.

The solution provided for OO-Method illustrates the type of solution needed for any MDD method. We have generated 47 new specific conceptual primitives (see [27] and [23]) to enable OO-Method to deal with MoU configurable properties. The conceptual model of OO-Method already supported 9 configurable properties for which no new primitives were required.

The last step in the proposed procedure is to proceed with the **changes to the model compiler**. Again, this step is method dependent. The only changes to be made to OO-Method model compiler to support Structured Text Entry are to include *Specify the input widgets visualization type (MoU1_STE),* since the other two MoUs are already supported. The aim of these changes is to generate the code that implements the type of widget specified by means of conceptual primitives.

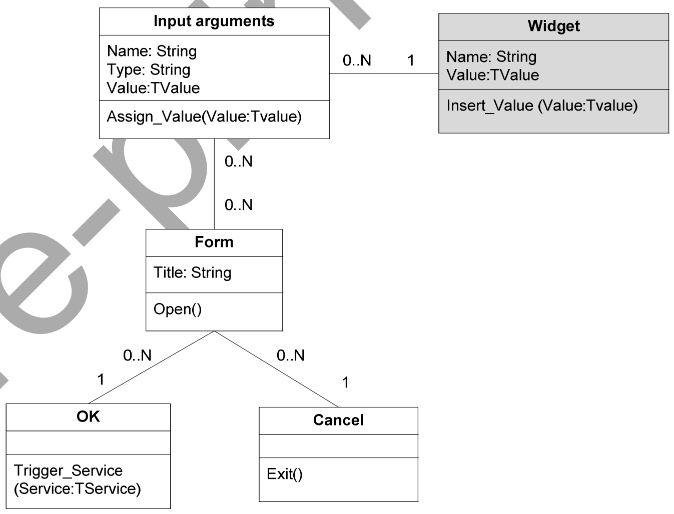


Fig 7. Class Diagram to represent the implementation of MoU1_STE

We use UML class diagrams to represent the changes in the code transformation process. Fig 7 shows every class that is affected by the inclusion of *MoU1_STE*. New software classes required to implement MoUs are shaded grey, classes extended with new

Pre-print version

attributes and methods appear with a background crossed by diagonal lines, whereas unchanged classes appear on a white background. The meaning of each class is as follows:

- **OK**: This class represents the button that the user uses to trigger a service.
- **Cancel**: This class implements a cancel button that goes back to a previous window.
- **Form**: This class implements a window where the user must enter values (SIU). Once the values have been input, the user can trigger the service that requires the arguments (by means of the OK button).
- **Input arguments**: This class represents the arguments required to execute the service related to the form.
- **Widget**: This class represents a widget that is the front-end of an input argument.

Fig 8 shows the classes to implement the properties of *Warning_message (MoU1_W)*. Classes Form, OK and Cancel have the same meaning as in Fig 7.

- **ClassX action:** Each one of these classes represents a class of the Object Model. These classes must be extended with methods to check the condition of the warning message.
- **Service wrapper:** This class connects the end user interface to the system functionality. It must be extended with methods to capture requests of actions that have a warning message related to them.
- **Alert manager:** This class shows warning messages to end users according to visualization alternatives previously defined.

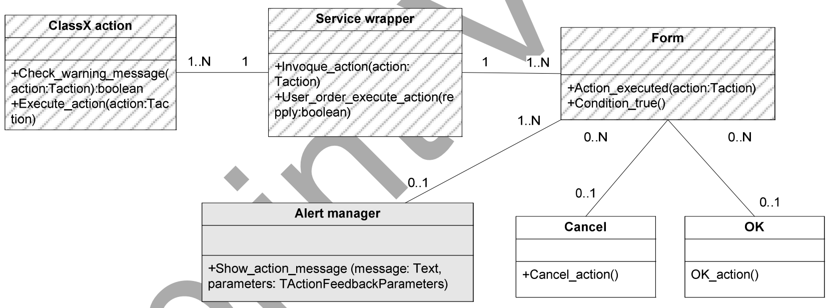

ClassX action
+Check_warning_message(action:Taction):boolean
+Execute_action(action:Taction)
1..N
1
Service wrapper
+Invoque_action(action: Taction)
+User_order_execute_action(repply:boolean)
1
1..N
Form
+Action_executed(action:Taction)
+Condition_true()
1..N
0..N
0..N
0..1
Alert manager
+Show_action_message (message: Text, parameters: TActionFeedbackParameters)
0..1
Cancel
+Cancel_action()
0..1
OK
OK_action()

Fig 8. Class Diagram to represent the implementation of MoU1_W

Changes applied to the conceptual model and to the model compiler need to be MDD method specific. For the 47 new OO-Method conceptual primitives generated in the third step, we have generated 94 new attributes, 76 new services and 11 new classes (see [27] and [23]). Since OO-Method has a model to represent the whole system interface (Interaction Model), FUFs can be more easily included in the method than in most MDD methods that do not count with a model to define all the characteristics of the user interaction (such as [9]). The level of expressiveness to represent the interaction within a model depends on the MDD method. For example, the interaction model of OO-Method already supported *Mask definition (MoU2)* and *Default values (MoU3)* and no change was required. Since most primitives that represent MoU properties are related to interaction, the workload for supporting MoUs will be greater for most MDD methods which have models to represent the interaction with poorer expressiveness than it has been for INTEGRANOVA.

The process to incorporate FUFs in a MDD method will be carried out only once. This effort is worth since once it has been done, analysts will be able to incorporate FUFs in

Pre-print Version

their development and improve the usability of the system by means of abstract primitives. Using these primitives as input, the model compiler automatically will generate the code that implements the MoUs.

5 Laboratory Evaluation

The aim of the evaluation we have carried out is to study whether end users perceive the benefits of including MoUs in the system. If so, the effort to include MoUs in a MDD method will be worthwhile, since MoUs improve end user's satisfaction. Most HCI recommendations (including FUFs) are based on experts' opinion and their usability improvement has not been empirically evaluated.

We have carried out a controlled experiment with 66 subjects using a car rental Web application. We divided the experimental subjects into two sets: subjects that interact with the system without MoUs and subjects that interact with the system including several MoUs. The most common system functionalities are: reserve a car; pick up a car; return a car; register a new customer; create an invoice. This Web application has been fully developed using INTEGRANOVA [6]. One author of this paper has included manually in the generated code the MoUs not supported by INTEGRANOVA (only Mask definition and Default values are currently supported by the MDD tool).

5.1 Experiment Definition

We evaluate two **research questions**:

- **R1:** Is the satisfaction of users who interact with MoUs better than the satisfaction of users who interact without MoUs?
- **R2:** Do users interacting with MoUs record better times than users interacting without MoUs?

We identify the following **null hypotheses** related to research questions R1 and R2:

- **H1$_0$**: Satisfaction for users interacting with MoUs is the same as satisfaction for users interacting without MoUs.
- **H2$_0$**: Time for users interacting with MoUs is the same as time for users interacting without MoUs.

There are two **response variables** [15] in the experiment: user satisfaction level and time to finish the task. *User satisfaction level* indicates whether or not the user is satisfied with the interaction. *Time to finish the tasks* measures how long it takes the user to complete the experimental tasks.

We have defined a **metric** for each response variable:

- **M1:** User satisfaction is measured by means of a five-point Likert-scale questionnaire. To design the questionnaire, we have followed HCI recommendations for questionnaires to evaluate usability [33]. We first identified the usability attributes to which each MoU is related. To do this, we used the list of usability attributes defined in ISO 9126-1 [13], since they are measurable entities. Second, we defined a question for each usability attribute related to the MoUs included in the experiment. Users have to respond to these questions on a five-point Likert-scale. For example, the Specify the visualization type of input widgets (MoU1) MoU is related to three usability attributes: Minimal Actions, Familiarity of Concepts and Error Prevention. Each usability attribute results in a question. Two questionnaire items are generated for each question

Pre-print Version

(a positive and a negative statement) in order to verify user response reliability. Subjects are asked to check the box that best represents their opinion from "I totally agree with the affirmative sentence" to "I totally agree with the negative sentence". Besides, general usability questions are asked after subjects have completed all the tasks. These questions are: Is the system easy to use? Would you recommend this system to other people? Are you generally satisfied with this system?

- **M2:** Time (measured in seconds) to finish the tasks. This time is measured per task and subject through the implementation of a hidden timer. The timer starts when the task is shown to the subject and it stops when the subject indicates that the task is finished.

There is one **factor** [15] in the experiment: *Use of MoUs*. This factor involves studying the Web application with and without MoUs. There is a **blocking variable**: *Previous experience of applications generated with INTEGRANOVA*. For this variable, we divided the subjects into experienced INTEGRANOVA application users and beginners.

The **subjects** were selected out of convenience. There were a total of 66 subjects from different backgrounds. All subjects had interacted with Web applications before and were aged from 21 to 56 years. They were volunteers from different countries that were able to perform the evaluation over the internet as if they were employees of offices all over the world of a rental car company. We have classified users depending on their previous experience with Web applications generated with INTEGRANOVA, since usually, users of Web applications are subjects without any knowledge of computer engineering. Learning how to interact with Web applications generated with INTEGRANOVA might add noise to the evaluation that experienced users do not present. Table 6 shows the design of the experiment.

Table 6. Experimental groups

		Use of MoUs	
	Groups	With MoUs	Without MoUs
Experienced in INTEGRANOVA	G1 (11)	X	
	G2 (11)		X
Inexperienced in INTEGRANOVA	G3 (22)	X	
	G4 (22)		X

The **instruments** used for running the experiment are:

- **A demographic questionnaire**: This questionnaire gathers information about subjects' gender, age, experience of using Web applications and experience of using applications generated with INTEGRANOVA.
- **Tasks:** There are four tasks, each aiming at studying different MoUs. The tasks are the same for all subjects irrespective of whether they interact with or without MoUs. This ensures that all subjects interact with the application in the same way. We timed all subjects as they performed every task. This timer implements metric *M2*.
- **User satisfaction questionnaire**: After performing each task, the subjects fill in a questionnaire that captures satisfaction. The questionnaire includes a question for each usability attribute of the MoUs studied in the task (this questionnaire can be seen in [23]). This questionnaire implements metric *M1*.

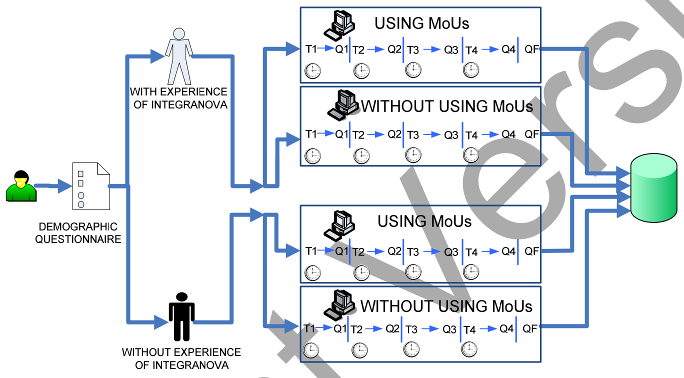
The instruments were posted on a Web page available over the internet [41]. We refer to this page as the ***Guide Page***, because it guides subjects through the experiment. The Guide

Pre-print Version

Page is not to be confused with the rent-a-car system on which the subjects perform the tasks.

5.2 Experiment Procedure

Fig 9 shows the experimental process. The experiment starts with the demographic questionnaire. After the subjects have filled in this questionnaire on the Guide Page, they record their experience on INTEGRANOVA applications. Depending on their background knowledge, the Guide Page automatically assigns the subject to the group of experts or beginners. Next, the Guide Page alternately assigns subjects to the with or without MoUs group. This procedure ensures that the groups of subjects with and without MoUs are balanced.


DEMOGRAPHIC QUESTIONNAIRE
WITH EXPERIENCE OF INTEGRANOVA
WITHOUT EXPERIENCE OF INTEGRANOVA
USING MoUs
WITHOUT USING MoUs
USING MoUs
WITHOUT USING MoUs
T1 Q1 T2 Q2 T3 Q3 T4 Q4 QF
Fig 9. Experiment operation

After subjects have been assigned to a group, the Guide Page shows the first task to be performed using the rent-a-car system (*T1* in Fig 9). The Guide Page automatically times subjects as they perform each task. When subjects finish the task, they have to fill in the satisfaction questionnaire on the Guide Page. This questionnaire includes a question for each usability attribute related to the MoUs of Task 1 (*Q1* in Fig 9). Next, the Guide Page shows Task 2 (*T2* in Fig 9). This process is repeated for each task. After subjects have finished all tasks, there is a short questionnaire on the Guide Page with three questions about the general usability of the system (*QF* in Fig 9).

Table 7 shows the relationship among experimental tasks, MoUs and usability mechanisms included in the experiment. From 12 usability mechanism and 22 MoUs, 4 usability mechanisms and 7 MoUs were relevant for the car rental system (MoU2_STE and Mou_STE3 are already supported by INTEGRANOVA). These MoUs include a total of 16 configurable properties and 1 non-configurable property. The other 15 MoUs have not been included in the experiment since they are not relevant in the context of the car rental system. We have discarded MoUs that are derived from usability mechanisms that are especially useful for systems with much input data (Step by Step), for systems whose actions last for several seconds (Interaction Feedback, Progress Feedback, Abort Operation), for systems with critical actions (Global Undo) and for systems where end users interact with the same system repeatedly during a long period of time (Preferences, Personal Object Space, Favourites). The application of all these mechanisms is not useful for the car rental system, where end users provide a few arguments, actions are simple,

Pre-print Version

actions last for a few milliseconds and end users do not interact with the same system repeatedly.

Table 7. Relationship between experimental tasks and modes of use

Task	Mode of use	Mechanisms
Create a car	Inform about service execution success or failure (MoU1_SSF)	System Status Feedback
	Specify the input widgets visualization type (MoU1_STE)	Structured Text Entry
	Default values (MoU3_STE)	Structured Text Entry
Create a bank account	Mask definition (MoU2_STE)	Structured Text Entry
	Dynamic help (MoU1_MH)	Multilevel Help
Reserve a car for rental	Warning message (MoU1_W)	Warning
Put up a car for sale	Show the action state (MoU3_SSF)	System Status Feedback

5.3 Data Analysis

We analysed the data using three methods: comparison of means, univariate general lineal model, and box and whisker plots. In the following we detail these three analyses.

The **comparison of means** is shown in Fig 10. The y-axis represents subject satisfaction. The smaller the value, the better satisfaction is. Value 1 means that the subject is completely satisfied and value 5 means that the subject is completely dissatisfied. The x-axis represents the MoUs studied in the experiment (MoU acronyms were described in Table 7). From Fig 10, the users that interact with MoUs appear to be more satisfied. This rule does not hold for MoU3_SSF (Default values).

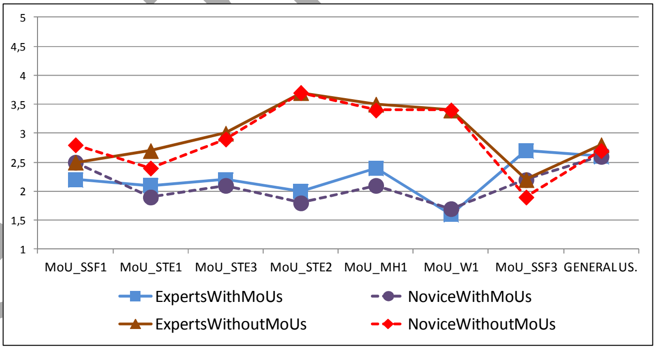


Fig 10. Average satisfaction per MoU

Fig 11 shows the average time spent on a task. The y-axis represents minutes and the x-axis represents the four tasks in the experiment. Experts appear to take less time to complete a task than beginners, which makes sense.

Preprint version

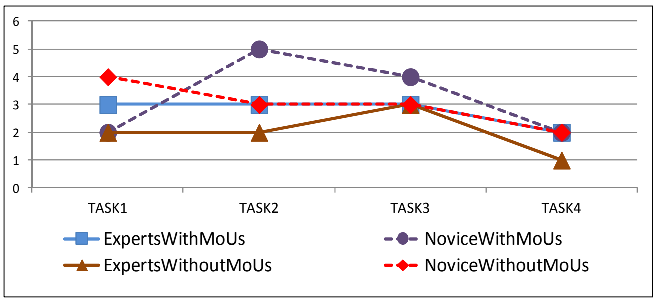


Fig 11. Average minutes per task

Univariate General Lineal Model (GLM) can only be applied in these three assumptions: residuals are independent of each other, residuals must be normally distributed, residuals should have the same variance for all values of the independent variables (homoscedasticity assumption). We ensured that all these assumptions were satisfied. All the residuals obtain a value close to 2 using the Durbin-Watson tests, which means that residuals are uncorrelated. All the residuals obtain a p-value higher than 0.05 with K-S test, which means that residuals are normally distributed. All the residuals obtain a p-value higher than 0.05 with Levene's test, which means that residuals have the same variances for each independent variable.

Table 8 shows the GLM for User satisfaction level with the Use of MoUs factor. The last column (Sig.) in Table 8 shows that subject satisfaction strongly depends on the use of MoUs, except for Inform about service execution success or failure (MoU1_SSF), Show the action state (MoU3_SSF) and General usability.

Table 8. Univariate GLM for User satisfaction level

Response variable	Type III Sum of Squares	Mean Square	F	Sig.
MoU1_SSF	6.68	6.68	0.911	0.344
MoU1_STE	44.182	44.182	5.802	0.019
MoU2_STE	458.727	458.727	50.047	0.000
MoU3_STE	45.833	45.833	12.77	0.001
MoU1_MH	94.561	94.561	22.571	0.000
MoU1_W	427.636	427.636	41.406	0.000
MoU3_SSF	21.879	21.879	1.822	0.182
General	2.97	2.97	0.529	0.47

Table 9. Univariate GLM for Time to finish the tasks

Response variable	Type III Sum of Squares	Mean Square	F	Sig.
Time_Task1	23221.879	23221.879	2.402	0.126
Time_Task2	62930.97	62930.97	3.464	0.067
Time_Task3	3136.742	3136.742	1.31	0.718
Time_Task4	858.242	858.242	0.076	0.784
Total_Time	33773.47	33773.47	0.247	0.621

Preprint Version

Table 9 shows the GLM for Time to finish the tasks with the Use of MoUs factor. We have timed each task and the addition of all of them. The last column (Sig.) in Table 9 shows that the time to finish the task is not related to the use of MoUs, which might make sense since not always a higher usability involves making tasks faster. Sometimes the improvement of other usability criteria different from efficiency (such as learnability or satisfaction) may involve a decrease in efficiency.

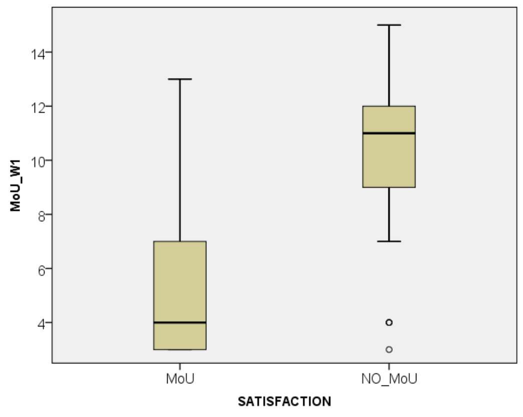


Fig 12. Box and whisker plot for User satisfaction level with and without MoU1_W

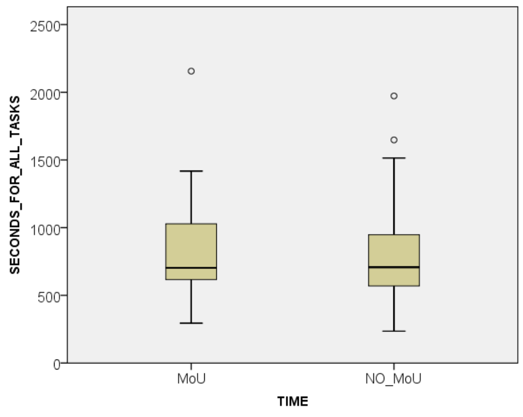


Fig 13. Box and whisker plot for Time to finish the tasks with and without MoUs

Box and whisker plots illustrate the median and quartile for both response variables (User satisfaction level and Time to finish tasks). Fig 12 shows the plot that compares User satisfaction level with and without Warning message (MoU1_W). The x-axis represents the use of the MoU1_W factor and the y-axis represents the sum of all the questions that measure the User satisfaction level of MoU1_W. According to Fig 12, there is a sizeable difference between the medians of subjects that do and do not interact with MoU1_W. The satisfaction value for subjects that interact with MoU1_W is better (the lower the value on the y-axis, the better satisfaction is). Also, the median for subjects with MoU1_W is positively skewed, whereas the median for subjects without MoU1_W is negatively

Pre-version

skewed. All MoUs have a similar trend, except for MoU1_SSF, MoU3_SSF and General usability, where the difference of medians between subjects that do or do not interact with MoUs is not so clear.

Fig 13 shows the box and whisker plot for Time to finish the tasks with reference to the Use of MoUs factor. The median of subjects that interact with and without MoUs is identical, although it is more positively skewed for subjects that interact with MoUs.

5.4 Results Interpretation

We can state that MoUs generally improve user satisfaction independently of the experience in the use of applications developed with the MDD method. Consequently, we reject hypothesis $H1_0$ (*satisfaction for users interacting with MoUs is the same as satisfaction for users interacting without MoUs*).

There are two exceptions: inform about service execution success or failure (MoU1_SSF) and show the action state (MoU3_SSF). An explanation for this result might be that these two MoUs were not implemented in the best possible way for the type of systems used in the evaluation. Both MoUs showed a message to explain whether or not the actions had been completed successfully and why the actions had been disabled, respectively. After the evaluation, some subjects commented that these messages threw them.

Another finding from the experiment is that the General usability of the system improves very little. It is noteworthy that, unfortunately, there are no studies (in either the HCI field or SE) about the degree of usability improvement that each specific mechanism, recommendation, heuristic or guideline provides. Usability benefits are evaluated after applying several improvements (typically suggested by a usability expert in HCI field). Therefore, there is no knowledge yet about the specific gain in usability when one or a small set of features is included in a system. Our experiment has incorporated 3 out of 6 FUFs, 4 out of 12 mechanisms and only 32% of MoUs. It seems that our approach is promising, since an improvement of general usability is still appreciated even for such a small incorporation of usability mechanisms.

With regard to the time hypothesis, the analysis shows that time is independent of interaction with or without MoUs. Moreover, there is no difference between the time taken by experts in applications developed with the MDD method and beginners. Consequently, we accept hypothesis $H2_0$ (*Time for users interacting with MoUs is the same as time for users interacting without MoUs*). For some tasks, like Task 2 (Create a bank account), time taken by users that interact with MoUs is even worse than time taken by users that interact without MoUs. Notice that users who interact without MoUs are not notified about mistakes made during the task. We observed that, very often, they did not take as long to complete the task because they performed the task incorrectly. For example, Task 2 forced the user to make a mistake that only subjects who interacted with MoUs noticed. In this task, the user had to insert a bank account number randomly. This value should have a specific 16-digit format (according to real cards). However, most subjects that interacted without MoUs inserted the wrong number of digits. Subjects that interacted with MoUs had a mask and a default value that indicated the correct number of digits. Therefore, these subjects noticed and spent time fixing the mistake. Reviewing the task outcomes, we can also state that MoUs help to improve user effectiveness (completing the task satisfactorily).

From the results of our experiment we can extract some relevant conclusions. First, end users' satisfaction improves after including MoUs in a system. This statement justifies the

Pre-print Version

enhancement of MDD methods to support MoUs. Second, the improvement of satisfaction does not depend on the level of end users' experience, which means that novice users also appreciate an improvement in satisfaction. Third, interaction time is not reduced significantly through the inclusion of MoUs, which means that the use of usability features is not suitable to reduce end users' effort.

5.5 Threats to Validity

We have used the classification of threats defined by Wohlin [43] to identify threats. Next, we discuss how we have dealt with those issues that threaten the validity of the experiment:

Subjects of random heterogeneity: This threat appears when there are subjects with more experience than others. In our experiment, all subjects had lengthy experience in Web applications. This was confirmed by the demographic questionnaire.

Maturation: This is the effect of subjects reacting differently to treatments as time passes. We dealt with this threat by designing an experiment that takes only 15 minutes.

Instrumentation: Even though tasks and questionnaires are the same for all subjects, these can be interpreted differently by each subject. In order to minimize this threat, we ran a pilot test with 4 subjects. This pilot test was useful for detecting ambiguous and hard-to-understand instructions and questions. All detected defects were fixed before carrying out the real experiment.

Hypothesis guessing: This threat accounts for cases where subjects guess the aim of the experiment and act conditionally upon that goal. This threat has been minimized by concealing the aim.

Interaction of selection and treatment: This is an effect of having a subject population that is not representative of the population that we want to generalize. This threat is minimized by blocking the number of subjects with and without experience of INTEGRANOVA applications. Moreover, we have studied subjects aged from 21 to 56 years, with different professions and from several countries.

Next, we describe threats that we did not manage to avoid due to the characteristics of our experiment:

Mono-MDD tool bias: This is the effect of studying our approach only with one MDD method. The application of our approach to INTEGRANOVA demonstrates that the proposal can work with a real tool but this fact does not involve that using other MDD tools, results would be the same. Each MDD method has its own model to represent the interaction, which hinders the generalization of our results to any other MDD method.

Experiment expectancies: This threat appears when participants can bias the results unconsciously due to expectations for specific results. Our experiment suffers from this threat since we implemented manually unsupported MoUs for the experiment and we also defined the approach to include MoUs in a MDD method.

Restricted generalizability: the results of our experiment are only valid for the car rental system, although the findings might be a clue for other systems that deal with management operations. To generalize the results to other systems, Web applications from different domains need to be used. However, our research is not aiming to gain empirical evidence on usability improvement through the incorporation of FUFs in a system. HCI recommendations for improving usability have been routinely followed during years without experimental evidence. This experiment just aims to collect some empirical data to illustrate that including usability features into a MDD method might worth.

Pre-print Version

6 Manual Versus MDD Design of Usability Features

Most MDD methods have mechanisms to represent the interaction with the end user. For example, WebRatio [2] includes a Presentation Model to express the layout and graphic appearance of pages, independently of the output device and of the rendition language. This model is based on an abstract XML syntax. UWE [19] enables the definition of the front-end interface by means of a Hypertext Model. It defines pages and their internal organization in terms of components for displaying content. This model also supports the definition of links between pages and content units that support information location and browsing. Components can also specify operations; such as, content management or user's login/logout procedures. NDT [9] has an abstract interface to represent the interaction with the user. This model is based on a set of evaluated prototypes, where the analyst and the users must choose the best one for the developing system. OO-Method [29][28] has two models to represent the interaction: the Abstract Interaction Model and the Concrete Interaction Model. The Abstract Model represents the interface independently of platform features and the Concrete Model represents the interface for a specific platform.

However, all these MDD methods (among others) do not provide enough expressiveness in their conceptual models to support usability features. Nowadays, analysts that work with MDD methods need to enhance manually the generated code to include such features. Next, we compare the effort of programming usability features manually versus the effort of modelling them through conceptual primitives. This comparative is performed using OO-Method. The object used in the comparison is the software system used in the laboratory evaluation. The subjects are two of the authors of this paper, who are experts in OO-Method and INTEGRANOVA. These subjects did not participated in the definition of OO-Method (1992) or in the implementation of INTEGRANOVA (2002) but they are developing software systems with INTEGRANOVA from 2006. The choice of experts is because of our approach requires analysts that already work with MDD methods and aim to deal with usability features through conceptual primitives, in the same way as they deal with functional or behaviour features. Analyst1 and Analyst2 have a wide experience developing software systems with INTEGRANOVA and they are already familiar with the architecture of the generated code. Moreover, Analyst2 is an expert in the development of Web applications using programming languages such as PHP or C#.

As we have commented in the evaluation section, the 5 MoUs studied in the experiment were implemented manually by Analyst1, since INTEGRANOVA does not yet support their code generation. We have considered the effort spent in this implementation as the data of Analyst1 to manually include usability features in the code generated from INTEGRANOVA. Analyst2 has replicated the development of the same system used in the experiment to analyze possible differences between efforts of both analysts. Table 10 shows effort of Analyst1 (A.1) and Analyst2 (A.2) to manually implement unsupported MoUs in terms of time and number of lines of code. Mask definition (MoU2_STE) and Default values (MoU3_STE) have not been included since they did not require manual implementation in INTEGRANOVA. Remember that both MoUs are already supported by INTEGRANOVA and they are modelled using the primitives shown in Fig 3 and Fig 4. As a result, Analyst1 needed 14 hours and Analyst2 needed 11.5 hours to manually implement the 5 MoUs. This time includes the time taken to debug the code. The source code of more than three classes was modified for each MoU, which is an added difficulty for the analyst.

Pre-print version

The reason why effort (both time and number of lines) obtained with Analyst2 is better than Analyst1 might be due to his great experience in the development of Web applications.

These data align with previous information on FUF design effort [16]. Table 11 shows this previous information focused on the FUFs used in our lab evaluation. The table includes information about the difficulty of implementing FUF functionality, the number of classes affected by the FUF, the complexity of the new methods that implement the FUF and the amount of interaction between new and existing methods. Feedback has the biggest impact on design even if not many classes are needed. At the other end of the scale, Help is the easiest FUF to implement since it does not require much functionality or many methods. However, it took us a long time to implement this FUF because it appears in all the system interfaces, since each interface has its own help.

Table 10. Time taken to implement each MoU manually in INTEGRANOVA

FUF	MoU	Time		Lines	
		A. 1	A.2	A.1	A.2
Feedback	Inform about service execution success or failure (MoU1_SSF)	2 h	2h	60	12
	Show the action state (MoU3_SSF)	2 h	5h	26	22
	Warning message (MoU1_W)	3 h	1.5h	94	16
User Input Error Prevention	Specify the input widgets visualization type (MoU1_STE)	3 h	1.5h	35	22
Help	Dynamic help (MoU1_MH)	4 h	1.5h	108	23

Table 11. Difficulty of including FUFs manually [16]

FUF	Functionality	Class	Methods	Interac.
Feedback	High	Low	Medium	High
User Input Error Prevention	Medium	Low	Medium	Low
Help	Low	Low	Low	High

Let us move now on measuring the workload of the analyst working with conceptual primitives (MDD approach). The 2 MoUs that are currently supported by INTEGRANOVA (Mask definition and Default values) are measured using existing INTEGRANOVA interfaces (see Fig 3 and Fig 4). The other 5 MoUs that are not supported currently by INTEGRANOVA have been calculated using interface prototypes (such as Fig 2). Table 12 shows the number of clicks and the approximate seconds that Analyst1 (A.1) and Analyst2 (A.2) needed to model MoUs used in the experiment. Note that comparing both analysts, we notice that Analyst1 took less effort (both time and number of clicks) than Analyst2.

Pre-print Version

The reason might be that Analyst1 is working in the proposal to represent usability features through conceptual models from the beginning, and he knows the prototypes perfectly. Data of Table 12 has been extracted without considering default values in INTEGRANOVA. These numbers can decrease sharply if the required and default values match in the interfaces to model MoUs.

Comparing Table 10 and Table 11 with Table 12, we find that the workload of the analysts using conceptual primitives to develop usability features is clearly smaller. Analysts needed around 12 hours to implement modes of use manually while they needed around 10 minutes to model modes of use through conceptual primitives.

With our proposal, the MDD designer needs to spend time improving the conceptual model with new conceptual primitives and to change the model compiler in order to incorporate usability features into the MDD method. However, this is a one-off workload. Once the new primitives have been enabled in the MDD method, analysts find it straightforward to include MoUs in just a few clicks.

Table 12. Number of clicks to model each MoU in INTEGRANOVA

FUF	MoU	Time		Clicks	
		A.1	A.2	A.1	A.2
Feedback	Inform about service execution success or failure (MoU1_SSF)	300 sec	350 sec	34	36
	Show the action state (MoU3_SSF)	5 sec	8 sec	2	2
	Warning message (MoU1_W)	10 sec	12 sec	8	9
User Input Error Prevention	Specify the input widgets visualization type (MoU1_STE)	7 sec	10 sec	4	5
	Mask definition (MoU2_STE)	7 sec	12 sec	5	5
	Default values (MoU3_STE)	6 sec	12 sec	6	6
Help	Dynamic help (MoU1_MH)	280 sec	400 sec	30	32

7 Related Work

In the literature, there are many works related to User Interface Design Patterns (UIDPs) and interaction patterns that propose solutions for well-known and frequent user interface problems. The major UIDP libraries include Tidwell [40], Perzel et al. [31] and van Welie et al. [42]. Tidwell represents UIDPs graphically in such a way that users can participate in architecture design. Perzel et al. describe a set of interaction patterns targeting web environments. Van Welie et al. have defined interaction patterns focused on the user's perspective.

A shortcoming of these patterns is that each author defines the patterns with a different notation and a different syntax. There are as many notations to represent UIDPs as authors

Pre-print Version

working in this area. Analysts need to be familiar with a huge amount of patterns expressed in different notations in order to take advantage of all of UIDPs. Some proposals try to overcome this problem by using a formal notation to represent patterns. Henninger et al. [12] use Semantic Web concepts to formally describe UIDPs in a way that computers can understand and that can be converted into a human-readable form.

UIDPs deal only with interface visual elements, i.e., a list of elements or a navigation button. Interface visual elements are not the only type of usability features, there are usability features strongly related to system architecture (as Folmer [11] et al. and Bass et al. [4] state). This type of usability feature cannot be represented using UIDPs. For example, a UIDP can specify that a progress bar is needed in an interface, but this pattern does not deal with the internal services needed to be executed for the progress bar to work.

There are very few works dealing with usability features in a MDD method. Moreover, when they are discussed, very few precise details are given. This makes it difficult to understand how these approaches could work correctly in practical settings. Tao [38] proposes to model usability by means of state transition diagrams. Each diagram can be used to represent an interaction between the system and the user. Paternò et al. [30] have defined a method for the development of user interfaces for applications based on Web services. The method starts from a task model and it is refined with an abstract and a concrete model. In order to guide the analyst, the process to specify interfaces is supported with usability guidelines. Both state transition diagrams and tasks models are not able to deal with all types of usability subcharacteristics; they are only able to represent interactions.

Sottet et al. [37] investigate MDD mappings for embedding both usability description and control. In this research, a user interface is defined as a graph of models describing the interface from different perspectives ranging from user tasks to deployment in the context of use. Transformations between different abstraction levels are performed by means of mappings. These mappings describe and control system usability. Raneburger et al. [32] propose improving system usability by MDD transformations. Raneburger's proposal focuses on minimizing navigation and scrolling in interfaces for small devices. Both Sottet and Raneburger define usability features inside transformation rules. This approach requires know-how to define transformations with usability.

There are works focused on measuring usability in conceptual models. Fernandez et al. [10] propose a usability model to evaluate system usability from conceptual models. According to Fernandez, evaluation performed at the conceptual model level produces a platform-independent usability report that provides feedback to the system analysis stage. Molina et al. [21] propose defining usability features from the early stages of the MDD development process. This approach focuses on navigational models provided by a tool that offers automatic support for all the activities. But most of the usability features are subjective and cannot be evaluated automatically without taking into account the user. For instance, features related to the attractiveness subcharacteristic cannot be measured by means of conceptual models. Therefore, the result of early usability evaluation is a prediction of sorts, but it cannot be considered trustworthy.

Summarizing, there are some proposals for dealing with usability in a MDD method. But few propose modelling usability features by means of conceptual models, which is a software artefact strongly related to producing quality systems. Moreover, we found no work that defines specific conceptual primitives to represent usability features in a MDD

Pre-print Version

method. Usability is an important feature of systems, therefore MDD methods should provide a mechanism to abstractly represent this characteristic.

8 Conclusions

We aim to enrich MDD methods with enough expressiveness to support usability features. This paper presents a procedure to extract properties of existing functional usability features and represent them with conceptual primitives. Next, these primitives can generate the code that implements the usability features thanks to a model compiler. Our proposal brings us a step closer to conceptual models where the models represent not only functionality, behaviour or persistence, but also usability features.

In a MDD context, we have found no other research proposing conceptual primitives to abstractly represent usability features. Other authors suggest dealing with usability by means of models, but do not define how to build such models. In general, in any MDD method, usability features are manually implemented once the system has been generated from a conceptual model.

Our approach needs to be partly independent and partly dependent on the MDD method. The modes of use and properties obtained in this research are applicable to any MDD method. The conceptual primitives and the changes to the model compiler are MDD method dependent, since the conceptual model and model compiler are exclusive to the MDD method. However, our work on the OO-Method shows that our approach works and is useful for guiding designers through the changes that should be made to other MDD methods. The application of our proposal to other MDD methods depends on the expressiveness of their conceptual models. OO-Method has an interaction model, which facilitates the inclusion of new conceptual primitives to represent interaction features. However, MDD methods with less expressiveness to deal with interaction would require adding more conceptual primitives to represent MoUs.

By means of an experiment, we have observed that our approach improves user satisfaction. This means that we are getting better user satisfaction by incorporating MoUs in a system. We have also compared the workload required to introduce MoUs by means of conceptual primitives versus manually. Once the primitives representing MoUs have been incorporated into the MDD method, there is a sizeable reduction in analyst workload with respect to manual implementation.

Pre-print Version

Acknowledgments

This work was developed with the support of the Spanish Ministry of Science and Innovation project SMART ADAPT (TIN2013-42981-P), TIN2011-23216 and was co-financed by ERDF. It also has the support of Generalitat Valenciana-funded ORCA project (PROMETEO/2009/015) and UV (UV-INV-PRECOMP13-115032).

References

[1] S. Abrahão, E. Iborra, and J. Vanderdonckt, "Usability Evaluation of User Interfaces Generated with a Model-Driven Architecture Tool," in Maturing Usability, E. Law, et al., Eds., ed: Springer, pp. 3-32, 2008.

[2] R. Acerbis, A. Bongio, M. Brambilla and S. Butti, WebRatio 5: An Eclipse-Based CASE Tool for Engineering Web Applications, Lecture Notes in Computer Science, vol. 4607, pp. 501-505, 2007.

[3] N. Aquino, J. Vanderdonckt, F. Valverde and O. Pastor, Using Profiles to Support Model Transformations in

the Model-Driven Development of User Interfaces, presented at 7th Int. Conf. on Computer-Aided Design of User Interfaces CADUI'2008, Albacete, Spain, pp. 35-46, 2008.

[4] L. Bass and B. John, Linking usability to software architecture patterns through general scenarios, The journal of systems and software, vol. 66, pp. 187-197, 2003.

[5] J.M. Bastien and D. Scapin, Ergonomic Criteria for the Evaluation of Human-Computer Interfaces, Rapport technique de l'INRIA, pp. 79, 1993.

[6] INTEGRANOVA http://www.integranova.com

[7] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, Non-Functional Requirements in Software Engineering. London: Kluwer Academic Publishing, 2000.

[8] T. Coram and L. Lee. A Pattern Language for User Interface Design. http://www.maplefish.com/todd/papers/experiences/Experiences.html, 1996

[9] M.J. Escalona and G. Aragon, NDT. A Model-Driven Approach for Web Requirements, IEEE Transactions on Software Engineering, vol. 34, pp. 377-390, 2008.

[10] A. Fernández, E. Insfrán, and S. Abrahão, Integrating a Usability Model into Model-Driven Web Development Process, presented at Web Information Systems Engineering - WISE 2009, pp. 497-510, 2009.

[11] E. Folmer and J. Bosch, Architecting for usability: A Survey, Journal of Systems and Software, vol. 70 (1), pp. 61-78, 2004.

[12] S. Henninger and P. Ashokkumar, An Ontology-Based Infrastructure for Usability Design Patterns, presented at Semantic Web Enabled Software Engineering (SWESE), Galway, Ireland, pp. 41-55, 2005.

[13] ISO/IEC 9126-1, Software engineering - Product quality - 1: Quality model, 2001.

[14] ISO 9241-11, Ergonomic requirements for office work with visual display terminals - Part 11: Guidance on Usability, 1998.

[15] N. Juristo and A. Moreno, Basics of Software Engineering Experimentation, Springer, 2001.

[16] N. Juristo, A. M. Moreno and M. I. Sánchez, Analysing the impact of usability on software design, Journal of Systems and Software, vol. 80, pp. 1506-1516, 2007.

[17] N. Juristo, A. M. Moreno and M. I. Sánchez, Guidelines for Eliciting Usability Functionalities, IEEE Transactions on Software Engineering, vol. 33, pp. 744-758, 2007.

[18] S. Kent, "Model Driven Engineering," presented at the Proceedings of the Third International Conference on Integrated Formal Methods, pp. 286-298, 2002.

[19] N. Koch, A. Knapp, G. Zhang and H. Baumeister, "UML-Based Web Engineering, an Approach Based on Standards, Web Engineering, Modelling and Implementing Web Applications, Springer, pp. 157-191, 2008.

[20] S.J. Mellor, A. N. Clark and T. Futagami, Guest Editors' Introduction: Model-Driven Development, in IEEE Software, vol. 20, pp. 14-18, 2003.

[21] F. Molina and A. Toval, Integrating usability requirements that can be evaluated in design time into Model Driven Engineering of Web Information Systems, Advances in Engineering Software, vol. 40, pp. 1306-1317, 2009.

[22] P.J. Molina, S. Meliá, and Ó. Pastor, JUST-UI: A User Interface Specification Model, presented at Computer Aided Design of User Interfaces (CADUI'2002), Valenciennes, France, 2002.

[23] List of changes: http://hci.dsic.upv.es/FUF/ChangesList.html

[24] Y. I. Ormeño and J. I. Panach, "Mapping study about usability requirements elicitation," presented at the Proceedings of the 25th international conference on Advanced Information Systems Engineering, Valencia, Spain, pp. 672-687, 2013.

[25] J.I. Panach, N. Juristo and O. Pastor: Including Functional Usability Features in a Model-Driven Development Method Computer Science and Information Systems (ComSIS) vol. 10, 999-1024, 2013.

[26] J,I. Panach, N. Juristo and O. Pastor: Introducing Usability in a Conceptual Modeling-Based Software Development Process, presented at 31st International Conference on Conceptual Modeling (ER), Vol. 7532. Springer, Lecture Notes in Computer Science, Florence, Italy, 525-530, 2012.

Pre-print Version

[27] J.I. Panach. Incorporating Usability Mechanisms in MDD Development. PhD Dissertation. Universidad Politécnica de Valencia, 2010

[28] O. Pastor, J. Gómez, E. Insfrán and V. Pelechano. The OO-method approach for information systems modelling: from object-oriented conceptual modelling to automated programming, Information Systems, vol. 26, pp. 507-534, 2001.

[29] O. Pastor and J. Molina, Model-Driven Architecture in Practice, Springer, 2007.

[30] F. Paternò, C. Santoro and L.D. Spano: Engineering the authoring of usable service front ends, Journal of Systems and Software vol. 84, 1806-1822, 2011.

[31] K. Perzel and D. Kane, Usability Patterns for Applications on the World Wide Web, presented at PloP'99 Conference, 1999.

[32] D. Raneburger, R. Popp, S. Kavaldjian, H. Kaindl, and J. Falb, Optimized GUI Generation for Small Screens, Model-Driven Development of Advanced User Interfaces, vol. 340, Springer, pp. 107-122, 2011.

[33] J. Sauro and J. R. Lewis, Quantifying the User Experience: Practical Statistics for User Research, Morgan Kaufmann, 2012.

[34] C. D. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer. 25-31 Available: http://doi.ieeecomputersociety.org/10.1109/MC.2006.58, 2006.

[35] B. Selic, The Pragmatics of Model-Driven Development, IEEE software, vol. 20, pp. 19-25, 2003.

[36] S. Sendall and W. Kozaczynski, Model Transformation: The Heart and Soul of Model-Driven Software Development, IEEE Software, vol. 20, pp. 42-45, 2003.

[37] J.S. Sottet, G. Calvary, J. Coutaz, and J.-M. Favre, A Model-Driven Engineering Approach for the Usability of Plastic User Interfaces, presented at Engineering Interactive Systems joining Three Working Conferences : IFIP WG2.7/13.4 10th Conference on Engineering Human Computer Interaction, IFIP G 13.2 1st Conference on Human Centred Software Engineering, DSVIS - 14th Conference on Design Specification and Verification of Interactive Systems, pp. 22-24, 2007.

[38] Y. Tao, An Adaptive Approach to Obtaining Usability Information for Early Usability Evaluation, presented at International MultiConference of Engineers and Computer Scientists (IMECS), pp. 1066-1070, 2007

[39] UML: http://www.uml.org/

[40] J. Tidwell, Designing Interfaces, O'Reilly Media, 2005.

[41] Web used in the experiment: http://hci.dsic.upv.es/TareasEvaluacion

[42] M.v. Welie and H. Traetteberg, Interaction Patterns in User Interfaces, presented at 7th. Pattern Languages of Programs Conference, Illinois, USA, 2000.

[43] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Springer, 2012.

Pre-print Version