Dealing with Usability in Model Transformation Technologies[1]

Jose Ignacio Panach[1], Sergio España[1], Ana M. Moreno[2], Óscar Pastor[1]

[1] Technical University of Valencia
Department of Information Systems and Computation
Camino de Vera s/n, 46022, Valencia, Spain.
{jpanach,sergio.espana,opastor }@dsic.upv.es
[2] Technical University of Madrid
Computing Science School
28660 Boadilla del Monte, Madrid, Spain
ammoreno@fi.upm.es

Abstract. Nowadays, the concept of Model Transformation Technology (MTT) is widely accepted in the Software Engineering community. These technologies have the capability of generating software code (solution space) from a conceptual model that specifies the system abstractly (problem space). Most MTTs disregard interaction modelling (and specifically usability modelling), even though usability is as important as functionality to produce high-quality software. The issue of ensuring usability has been researched from several perspectives. One of these perspectives is based on elaborating the information to be discussed with the user to gather usability needs and the modifications to be done in software design to support those needs. We adopt this perspective by using guidelines to capture usability requirements and architectural usability patterns. The main contribution of this paper is to propose a strategy to include existing usability features inside a complete Model Transformation Technology, from abstract modelling to code generation. In order to reach this goal, new conceptual primitives have to be defined using as a source the description of the usability features. The analyst uses these primitives to model the functionality of the usability features. Once the strategy is defined in general terms, it is applied to a specific Model Transformation Technology: the OO-Method.

1. Introduction

If we look back on software development history from a global perspective, the abstraction level has been continuously rising from the solution space to the problem space. At the beginning, software systems were built in a low-level, machine-understandable code. Then, new programming languages got progressively closer to the developer's cognitive models and provided a higher abstraction level, with the objective of improving efficiency and understandability. According to this evolution, modern Software Engineering (SE) is interested in providing strategies based on

[1] This work has been developed with the support of MEC under the projects SESAMO TIN2007-62894. co-financed by FEDER and TIN2005-00176.

sound Model Transformation Technologies, where the main idea is to obtain the final software product by means of a transformation process. Model Transformation Technology bridges the gap between the models at different abstraction levels. In essence, Model Transformation Technologies take a model as input and generate another model as output. Model Transformation Technology is a part of the Model Driven Development (MDD) approach. MDD is simply the notion that we can construct a model of a system that we can then transform into the real thing [16]. In many MDD approaches, the system is modelled by means of a conceptual model. The modelling language that supports the conceptual model offers a set of conceptual primitives[2]. A model compiler is an automated tool that receives the conceptual model and generates the software system code. This idea is represented by different proposals, such as the MDA standard [15], the Conceptual-Schema Centric-Development challenge [22], and the Extreme Non-Programming approach [18]. More importantly, tools have started to enter the game with industrial solutions (OlivaNOVA [5], AndroMDA [1]).

In this work, within this context of elevating the abstraction level in software development, we are interested in the study of a basic aspect for software quality [11]: *usability*. ISO 9241-11 [10] defines usability as "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". Usability benefits have been pointed out by several authors [3][8]. However, in the SE community, the main focus is generally placed on data and functional modelling, disregarding usability aspects [3].

Usability is a very wide concept. Human-Computer Interaction literature provides many different recommendations to improve the usability of a software system. In [12], authors present three groups of recommendations:

1. Usability recommendations with impact on the user interface (UI). These recommendations refer to presentation issues with slight modifications of the UI design (e.g. buttons, pull-down menus, colours, fonts, layout).
2. Usability recommendations with impact on the development process. These can only be taken into account by modifying the whole development process. For example, those that intend to reduce the user cognitive load require involving the user in the software development.
3. Usability recommendations with impact on the architectural design. These involve building certain functionalities into the software to improve user-system interaction. These set of usability recommendations are referred to as **Functional Usability Features** (FUFs). Examples of these FUFs are providing cancel, undo and feedback facilities. A big amount of rework is needed to include these features in a software system, unless they are considered from the first stages of the software development process [12]. User needs related to FUFs can be gathered by means of requirements elicitation guidelines [14] and the architectural design that they involve can be described by means of design patterns (aka architectural usability patterns) [13].

[2] In the context of this paper, a conceptual primitive is an element of the modelling language that allows to abstractly represent some aspect of the system. For instance, in a class diagram, the class is the main conceptual primitive; furthermore, we also consider conceptual primitives the class attribute and the class service.

In this paper, we will focus on FUFs. We are interested in studying how to incorporate FUFs in Model-Driven Development (MDD) approaches. In order to incorporate the FUFs, their corresponding usability requirements guidelines and architectural usability patterns have been studied. As a result, a set of changes to extend the modelling language with new conceptual primitives and to modify the model compiler have been identified.

One of the most remarkable benefits of using an MDD approach to address FUFs is the ease with which the system usability is improved: whenever a usability defect is found once the software system has been generated, the developer does not need to change the architecture of the system nor fix the defect in the source code, as in classical approaches. In our approach, the defect can be fixed by changing the conceptual model; that is, using the usability-related conceptual primitives. Then, the model compiler will generate the software again, now including components that fulfill the desired usability features. In this way, architectural usability patterns and the functionality that supports the business logic are appropriately intertwined in the code.

This paper takes the OO-Method [23] as an example of Model Transformation Technology. The OO-Method is an MDA-compliant, object-oriented software production method that generates computerised information systems automatically. We have chosen it because an industrial tool called OlivaNOVA [5] supports the method, thus allowing us to perform challenging experiments in practice.

The paper is structured as follows. Section 2 reviews the literature on usability modelling. Section 3 explains the MDA paradigm. Section 4 describes our approach to include usability modelling in a MTT. Section 5 shows a practical application of this approach to the OO-Method. Finally, section 6 shows the conclusions of this work.

2. Usability Modelling in the Literature

As far as authors know, there is no model transformation-based software development method that treats features directly in a Usability Model that is independent of the rest of the models that make up the conceptual model. Normally, methods deal with usability indirectly, via the models that represent the user-system interaction. Some modelling tools offer a model to represent the interaction that leads to improve particular usability features. Many modelling techniques and tools follow this trend and propose the task model as an abstract interaction model from which an abstract interface model is derived. DiaTask [28] derives a dialog graph from a task model. The dialog graph is composed by views and transitions. Each view is an abstraction of a single subdialog of the described user interface. A transition is a directed relation between an element of a view and a view. An interface prototype that reflects the navigational structure is generated from the dialog graph. UI Pilot [26] enables designers to create the initial specifications for the screens of website, desktop or mobile applications. UI Pilot is based on the use of wireframes (simple annotated descriptions of interface elements). Wireframes have proven effective in communicating requirements between design and engineering teams. UsiXML [30] is an XML-based interface description language. Their authors propose a task model as a primary interaction

model that is used to derive interface models later. The UsiXML suite of tools allows interface sketching and generation.

Many tools support the Concur Task Tree (CTT) notation [24] for interaction modelling. The UsiXML suite supports CTT models. TERESA (Transformation Environment for inteRactivE Systems representAtions) [19] is a tool that supports CTT modelling and generates interfaces for different types of devices. In turn, SUIDT (Safe User Interface Design Tool) [2] is a tool that automatically generates interfaces using several interrelated models, some of which are based on the CTT notation. The above mentioned tools are only focused on interface modelling, disregarding the modelling of the software system functionality. CTT notation is widely employed in the Human Computer Interaction community, but it does not support functionality modelling. We advocate integrating three axes of system descriptions, as the OO-Method does: system memory, system reaction and user-system interaction.

Another tool that is worth mentioning is VAQUITA [4], which is aimed at web environments. The tool uses mapping rules to reverse-engineer an HTML page and obtains an interface model. Then, the interface model can be modified in order to improve usability. However, the holistic perspective of modelling and a precise model transformation-based, complete software production process is missing.

Finally, there are several UML-based approaches where interaction and functionality modelling have been integrated. This is the case of UMLi [29] and WISDOM [21]. UMLi is a set of user interface modelling primitives that extend UML to provide greater support for UI design. This way, some usability attributes can be improved. However, UMLi models are so detailed that the modelling turns out to be very difficult, thus hampering its industrial application. WISDOM is a software engineering approach that enriches UML with the necessary stereotypes to allow user-centred development. It also has a detailed user interface design. Three of its models are concerned with interaction modelling at different stages: the Interaction Model (analysis), the Dialog Model and the Presentation Model (both in design). The WISDOM notation simplifies the application of UML with regard to UMLi. However, neither of the two methods considers the generation of full functional systems.

3. MDA Environments

In 2001, the Object Management Group proposed an increasingly popular paradigm: the Model Driven Architecture (MDA) [15]. This de facto standard defines how to apply Model Driven Development. Three viewpoints were proposed:

1. A Computation Independent Model (CIM) focuses on the environment and the requirements of the system.
2. A Platform Independent Model (PIM) focuses on the operation of the system, which stays constant across any possible technological platform.
3. A Platform Specific Model (PSM) aims to provide the platform-dependent viewpoint with those features that are specific to a platform.

As defined by the Object Management Group, a model transformation is the process of converting one model to another model of the same system [15]. Commonly, the target model is in a lower abstraction level than the source model and, therefore, it

is closer to the final implementation. By means of consecutive transformations, we end up with an executable model of the system: the Code Model.

Transformations can be applied manually, with computer assistance, or automatically. Transformation rules have to be unambiguously specified using some language, regardless of the degree of automation. Again there is a wide choice, ranging from natural language descriptions to QVT [17] specifications. Among the several model transformation approaches that can be used, the Metamodel Transformation deserves our special attention (see Figure 1). The definition of transformation rules is a hard task but it benefits analysts in many ways:

- Complete support for the software life cycle from requirements to maintenance.
- Reduction of software development costs. Analysts put their main effort in the analysis stage. Subsequent stages are facilitated by automatically deriving initial models that are refined by the analysts. This saves time and resources.
- Quality improvement. Code generation reduces the possibility of error.
- The same model can be transformed into code for several programming languages.

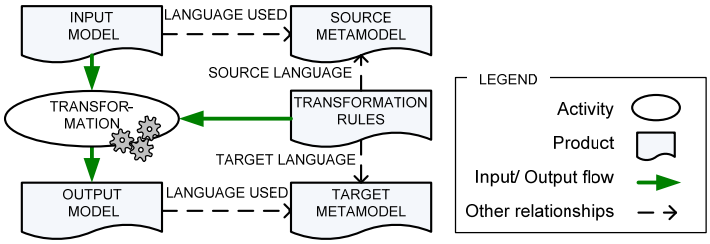

Fig. 1. Metamodel Transformation

The advantages of using a model transformation approach have more weight than the effort required to define transformation rules. For this reason, several Model Transformation Technologies have recently appeared in order to generate code from conceptual models [22][6]. We could say that "conventional" SE focuses on system structure and system behaviour, but it does not do a good job from the interaction modelling perspective in general, and from the usability point of view in particular. Some proposals such as [27][6] aim to integrate the usability engineering process with the SE process. These proposals show that these two perspectives (interaction and SE) do not often understand the goals and needs of others, pointing at several integration problems: lack of coordination; lack of provision for change; lack of synchronization of development schedules; lack of communication among different developer roles; lack of constraint mapping and dependency checks.

In the next section, we provide a concrete proposal that contributes to solve these problems and to accomplish the integration of interaction modelling and SE.

4. Projecting Usability to an MDA-based Method

This section explains a strategy for adapting an MDA-based software development method in order to address usability. Our proposal is focused on Functional Usability

Features (FUFs) because they have a wide impact on design [12]. At the requirements stage, analysts use requirements guidelines to elicit user needs regarding FUFs [13]. At the design stage, architectural usability patterns help developers to support the FUFs [14]. All in all, the rationale behind our approach is the following:

- Usability requirements guidelines contribute to define the different configuration possibilities of a specific usability feature (the details of the user needs). Feature configuration has to be modelled by means of conceptual primitives. Consequently, usability requirements guidelines have been studied to define conceptual primitives that represent FUF configurations.
- Architectural usability patterns offer an abstract design solution to include, in the system architecture, the components that support the usability feature. This proposal can be used to define the code generation strategy of the MTT.

As Figure 2 shows, we have proposed a four-step strategy to embed these FUFs in an MDA-based method. The first step is to study how the usability features can improve the usability of the generated systems. To fulfil this goal, we study the usability requirements guidelines to identify Ways of Use (WoU in Figure 2). The same feature may have several applications in the system. We define **Way of Use** as a specific application of a FUF in the final interface. We take the *Structured Text Entry* FUF as an example. It allows the specification of restrictions on data entry. Three Ways of Use are defined for this FUF: (1) this FUF can specify the widget type to enter data with a specific format (checkbox, radiobutton, listbox, etc.); (2) this FUF can define a mask that specifies the required format of an input text; (3) also, this FUF can define default values in order to help the user to enter information.

The second step is the definition of one or several usability properties for each Way of Use. **Usability properties** are options of the FUF that are used to adapt it to the user's requirements related to usability. For example, in the first Way of Use of the *Structured Text Entry* FUF, we can define a usability property for selecting the type of widgets and another one for organizing the widgets on the screen. We consider two types of usability properties:

- **Non-configurable usability properties** have the same value in all generated systems. It is unnecessary to configure these properties because they do not offer any alternative. For example, if commonly accepted usability guidelines determine that each action should be accompanied by a progress bar to indicate when it will finish, the analyst's decision is not involved. This is a non-configurable usability property of the *Progress Feedback* FUF.
- **Configurable usability properties** with different configuration alternatives that depend on the analyst's decisions. For example, in the *Structured Text Entry* FUF, the analyst decides the type of the widget and its organization on the screen. Therefore, this FUF has two configurable usability properties.

The third step for including usability features in an MDA environment is to specify which models (or views) have to be modified in order to support each usability property. The conceptual model may be composed of several views. Each view models the system from one perspective. For instance, in a given MDA-based method, one view may be used to model persistence and another view to model interaction. Usability properties are modelled in these views by means of conceptual primitives. Each configurable usability property requires the definition of new conceptual primitives. In other words, the source metamodel of the MDA-based method has to be enriched with

new conceptual primitives. For instance, the configurable usability property of the *Structured Text Entry* which represents the type of the widget is represented by means of a conceptual primitive. This primitive specifies the widget type in the view which represents the interaction of the system. Non-configurable usability properties do not need new conceptual primitives because the knowledge to generate code to support their corresponding functionality can be embedded in the model compiler.

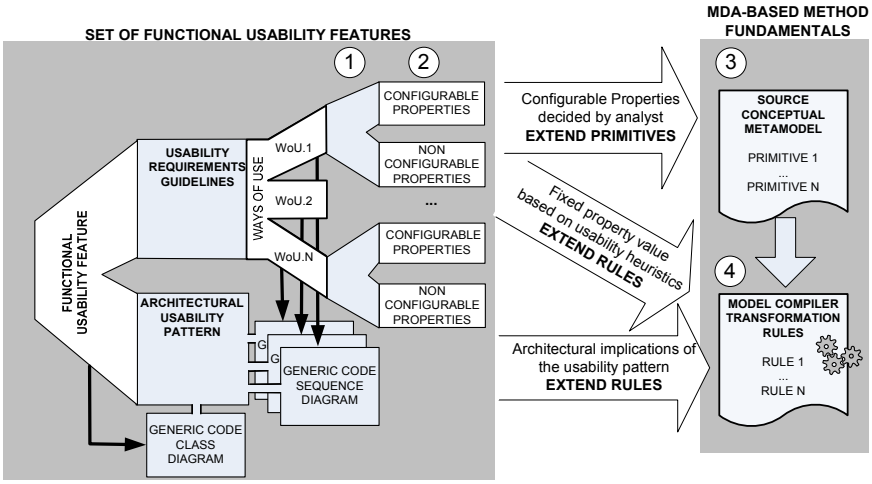

Fig. 2. Strategy to include Functional Usability Features in a MTT

The fourth and last step is to improve the transformation rules of the model compiler in order to ensure that it can generate code that supports the functionality of the usability features, taking the conceptual model as input. For that aim, architectural usability patterns can be helpful. The incorporation of usability features involves adding new classes and services to the generated code. Both types of properties imply changes in the model compiler. With regard to configurable properties, new transformation rules have to be defined to map the new conceptual primitives to programming language code. With regard to non-configurable properties, the model compiler will also generate their corresponding code. For instance, the non-configurable property of the *Progress Feedback* which states that all actions should have a progress bar implies including code to calculate the remaining time of the action execution.

We can conclude that the incorporation of usability features concerns the entire code generation process of an MDA environment. Once the usability features are addressed, all the code that implements the functionality of these features together with the business logic can be generated automatically by means of the model compiler.

5. A Practical Application in the OO-Method

This paper applies the Functional Usability Features to the OO-Method. The main advantage of the OO-Method is that an industrial implementation of the method (Oli-

vaNova [5]) provides a model compiler that generates fully functional systems from the OO-Method conceptual model. Moreover, the OO-Method conceptual model is abstract and platform-independent. These characteristics make it the most appropriate MDA environment to illustrate our proposal.

5.1 The OO-Method, an MDA Environment

This section argues that the OO-Method [23] is MDA [15] compliant. Following the MDA paradigm, the OO-Method is based on the creation of abstract models and the application of model transformations. The equivalence between the OO-Method and the MDA models is the following:

- The OO-Method conceptual model corresponds to the MDA Platform Independent Model. The OO-Method conceptual model is composed of four views:
  - The *Object Model* specifies the system structure in terms of classes of objects and their relations. It is modelled as an extended UML class diagram.
  - The *Dynamic Model* represents the sequences of events that can occur to a class of objects and the interaction between object classes.
  - The *Functional Model* specifies how events change object states. The behaviour of the system is modelled by the Functional and Dynamic Models working together.
  - The *Interaction Model* models the interaction between the system and the user. This model has two views: the Abstract Interaction Model and the Concrete Interaction Model [25]. The *Abstract Interaction Model* defines the interface without taking into account concrete aspects of visualization. It represents the interface independently of the types of interaction and the peculiarities of the platform. The *Concrete Interaction Model* specifies details of the interface. It is a user-interface model that specifies the interface representation in terms of elements that can be perceived by the end user. The Concrete Interaction Model is not supported yet by OlivaNOVA because it is currently under research.
- The architectural knowledge embedded in the model compiler corresponds to the MDA Platform Specific Model.
- The MDA Code Model corresponds to the generated code that supports the system.

5.2 Dealing with Usability in the OO-Method

This section shows an instantiation of the strategy to include FUFs in the OO-Method Model Transformation Technology. To achieve this goal, it is necessary to adapt the FUFs to the OO-Method. We explain this adaptation, following the steps defined above: (1) define the Ways of Use of the FUFs; (2) define the usability properties of each Way of Use; (3) extend the conceptual model with new conceptual primitives in order to model configurable usability properties; (4) extend the model compiler to support the new conceptual primitives and non-configurable usability properties.

For the sake of brevity, this section focuses on the changes implied by a usability feature called *Warning*. This feature is used to specify which information needs to be elicited and specified in order to ask for user confirmation in case the action requested has irreversible consequences. This FUF contributes to prevent user errors. The re-

quirements guidelines of the FUF elicits the information necessary to identify the actions where this feature should be applied and how to prevent the user for the consequences of those actions (generally by means of a message asking the user to accept or reject the action execution). The architectural usability pattern proposes to the developer a set of software components to include such feature in a software design.

5.2.1 Defining the Ways of Use of the FUF

Studying the requirements guidelines of the Warning FUF we can state that this feature only has one Way of Use: action warning - to notify the user before executing a potentially erroneous action. Some business rules recommend advising the user before executing an erroneous or irreversible action whenever a certain condition is satisfied. For example, in an invoicing system, the system should advise the user if an invoice with a total amount greater than 10.000 € is going to be emitted; this amount is infrequent, so it could be an error. This Way of Use is not currently supported by the OO-Method.

5.2.2 Defining the Usability Properties of the Way of Use

After studying the corresponding requirements guideline, the following usability properties have been defined for the action warning Way of Use: the business service associated to the warning, the condition, the text to show, and a set of format options for text visualization (see next step for more detail).

5.2.3 Extending the Conceptual Model with New Conceptual Primitives

All the usability properties of this Way of Use are configurable by the analyst. Therefore, the OO-Method conceptual model needs to be extended to support these properties. Two views are affected by the Warning FUF: the Object Model and the Concrete Interaction Model.

- Object Model: this view is extended with new conceptual primitives to model the following configurable usability properties:
  - The service in which the FUF is applied.
  - The condition that should be satisfied to show the warning message.
  - The text that will be shown to the user when the condition holds true.
- Concrete Interaction Model: this view needs new conceptual primitives to model the following configurable usability properties:
  - Whether or not the window is obtrusive[3].
  - The window type: alert, information or error.
  - Text font.
  - Size.
  - Colour.
  - Alignment.

In order to facilitate the analyst's work, these conceptual primitives should have a default value in case the analyst does not want to configure them. Default values

[3] The term obtrusive is used to define a window that does not allow any other user interaction until the window is closed. Sometimes this is referred to as 'modal'.

should be the values that are most frequently used for each conceptual primitive. The analyst can change these default values to adapt the conceptual primitives to the user's requirements. By default, the Warning feature is implemented by an obtrusive window of the alert type, with Arial font, size 10, black colour and centred alignment.

The inclusion of these conceptual primitives in the OO-Method implies changes in OlivaNOVA [5], the industrial tool that implements the OO-Method. Changes related to the *Warning* FUF affect the Object Model and the Concrete Interaction Model:

- Object Model. This view should allow specifying which services have an associated warning message, the condition, and the text for this message. Figure 3 shows a window prototype[4] where the analyst can model these usability properties.
  - Define the condition that, if fulfilled, triggers the warning message. The condition is edited using a wizard and the following elements can participate: (1) class attributes; (2) arguments of the service or transaction related to the warning message; (3) user functions defined by the analyst; (4) standard functions that are already included in OlivaNOVA to work with basic data types such as boolean, numeric, string and date types; (5) operators for basic data types.
  - Define the warning message that the system shows if the condition holds true.
  - Define explanatory commentaries for the developing team.

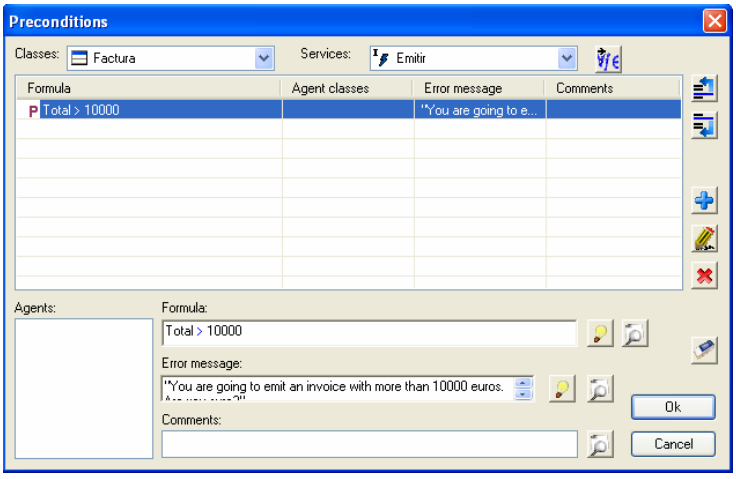

Fig. 3. Prototype to model the Warning usability feature in the Object Model

- Concrete Interaction Model. Once the analyst has modelled the functionality of the Warning FUF in the Object Model, the next step is to model by means of the Concrete Interaction Model how the warning message is shown to the user. Figure 4 shows a non-functional prototype with the conceptual primitives used by the analyst to model the warning visualization. The tree view on the left-hand side shows the services that have an associated warning message. For each warning message, the analyst can change the primitives on the right-hand side.

[4] These non-functional prototypes are only meant to illustrate the changes needed in the OlivaNOVA tool to support the Warning FUF.

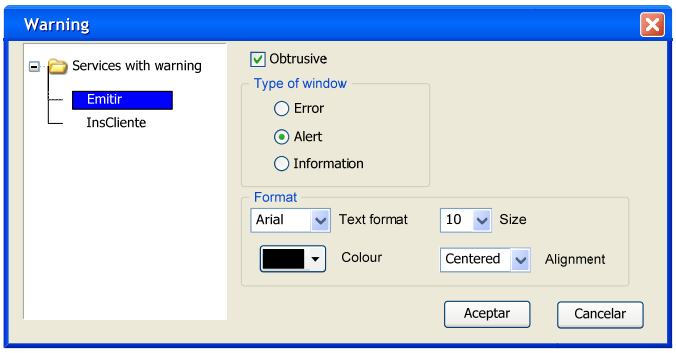

Fig. 4. Prototype to model the Warning usability feature in the Concrete Interaction Model

5.2.4 Extending the Model Compiler

Every conceptual primitive that composes the OO-Method conceptual model at the problem-space level is mapped to a piece of software code that represents it at the solution-space level. The architectural usability pattern can help in this task. The new conceptual primitives that are included to deal with the usability features should be related to a set of transformation rules in order to generate code that implements their functionalities. Although non-configurable properties do not have associated conceptual primitives, they need transformation rules too. Both facts imply changes in the automatic code-generation strategy. In order to abstractly represent these changes, we have used two types of diagrams (Figure 2): *Class Diagram* and *Sequence Diagram*. Class diagrams are used to represent the (software) classes associated to an architectural usability pattern, the relations among them, and the class methods that will implement the functionality offered by the usability feature. Sequence diagrams are used to express the sequence of actions that are going to be carried out by the classes that appear in the class diagram. The functionality of each FUF is represented by a single class diagram, while each Way of Use is represented by a sequence diagram.

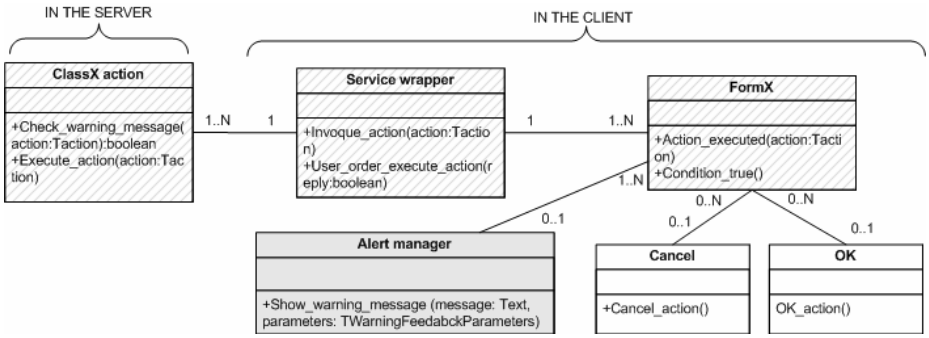

Fig. 5. Class diagram for Warning usability feature

OlivaNOVA can generate applications in C# and Java. We have focused on C#. To illustrate our approach, we offer the diagrams for the *Warning* feature (Figures 5 and

6). New (software) classes needed to implement the feature appear with grey background. Classes with some methods that have been modified to add the usability feature appear with a background crossed by diagonal lines. Finally, those classes that do not change appear with white background. These classes are:

- ClassX action. For each class of the Object Model (at the analysis stage), the model compiler creates a class of this type (at the implementation stage). We have informally used the letter X to abstractly represent the set of all these classes. This class checks the condition associated to the warning message.
- FormX. There is a class of this type for each service modelled in the conceptual model. The class *FormX* represents the set of all the forms used by the user to execute services. This class receives a notification from *ClassX action* when the condition holds true, and it invokes *Alert manager* to show the warning to the user.
- Alert manager. This new class is created from scratch to show a window in which the user can decide whether or not she/he wants to execute the action.

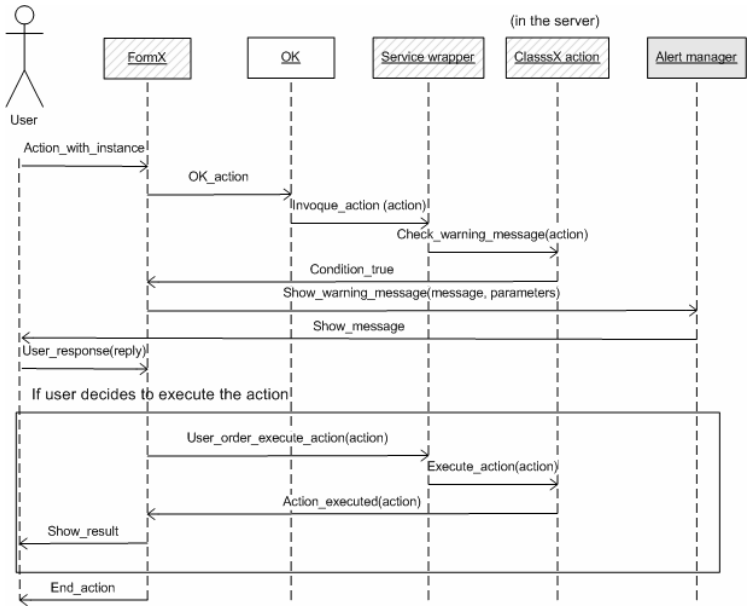

Fig. 6. Sequence diagram for Warning pattern

Figure 6 shows the sequence diagram (which represents the Way of Use of the Warning feature) whenever the condition to show the message holds true. When a user wants to execute an action, the *Service wrapper* class receives the request. This class launches the request to the *ClassX action* (which is implemented in the server). This class is in charge of verifying whether any warning message associated with the action exists; if so, it checks whether the condition is satisfied. When the condition is satisfied, the class *Alert manager* asks the user whether or not she/he wants to execute the action. If the user decides to execute the action, the class *FormX* will trigger the action. These changes in the model compiler make it possible to generate code to support the functionality of the Warning feature.

6. Conclusions

This paper discusses a strategy for including usability features with functional implications in a Model Transformation Technology. Our proposal is based on the use of already existing Functional Usability Features (FUF) that contain the information needed to specify and design such features. This method is based on the idea of abstracting the information contained in such usability features in order to include it in a conceptual model. Thanks to this abstraction, a model compiler can automatically generate the code for a specific programming language taking as input the conceptual model. This code supports the functionality of the usability features together with the functionality of the whole system. This automation reduces the development rework in case a usability defect is found in the implemented system. Notice how this proposal clearly helps to provide practical support to the existing tendency of addressing the treatment of usability in the early stages of the software development process. Therefore, it represents a starting point to improve the integration of usability and SE.

The proposed strategy consists of four steps that have to be taken for each FUF: (1) to study the FUF usability requirements guideline and to identify Ways of Use: each Way of Use is a particular application of the feature in a system; (2) to identify usability properties in each Way of Use: briefly, each property is an option of the FUF; (3) to extend the conceptual model with new conceptual primitives to represent configurable usability properties abstractly; (4) to change the transformation rules of the model compiler to support the code generation related to the new primitives and the non-configurable usability properties, using as input the architectural usability pattern.

This approach has been applied to the OO-Method using the Warning feature. The OO-Method is chosen as an example of Model Transformation Technology due to the high level of abstraction of its conceptual model. Taking as input the OO-Method conceptual model, a model compiler generates fully functional software systems. Both the OO-Method conceptual model and the model compiler need to be modified to incorporate the functionality of the usability features. The paper discusses the inclusion of the Warning FUF. Moreover, the changes to support such feature in OlivaNOVA (a industrial suite of tools that supports the OO-Method) are detailed.

As future work, this strategy should be applied not only to the Warning but also to the rest of the Functional Usability Features in order to incorporate them in the OO-Method. The inclusion of all these usability features should improve the usability of the systems generated with the OO-Method. To assess this issue, an empirical evaluation will be made. It will consist of usability tests with end users.

References

[1] AndroMDA, http://www.andromda.org/. Last visit: March 2008.

[2] Baron M., G. P. (Romania 2002). "SUIDT: A task model based GUI-Builder." Task MOdels and DIAgrams for user interface design (TAMODIA): 64-71.

[3] Bias, R.G., Mayhew D.J. (2005). Cost-Justifying Usability. An Update for the Internet Age.

[4] Bouillon, L., Vanderdonckt, J., Souchon, N. (2002). Recovering Alternative Presentation Models of a Web Page with VAQUITA. CADUI 2002, France.pp. 311-322.

[5] CARE Technologies S.A. www.care-t.com . Last visit: March 2008.

[6] Ceri, S., Fraternali, P., Bongio, A. (2000). Web Modeling Language (WebML): a modeling language for designing Web sites. WWW9, Amsterdam,.pp. 137 - 157.
[7] Chrusch M. (2000). Seven Great Myths of Usability. Interactions. pp. 13-16.
[8] Donahue G.M. (2001).Usability and the Bottom Line. IEEE Softwa, vol. 18(11) pp 22-30.
[9] Ferré, X., Juristo, N., Moreno, A. (2005). Framework for Integrating Usability Practices into the Software Process 6th International Conference on Product Focused Software Process Improvement (PROFES'05). Lecture Notes in Computer Science (LNCS) 3547.
[10] ISO 9241-11 (1998): Ergonomic Requirements for Office work with Visual Display Terminals. Part 11: Guidance on Usability.
[11] ISO/IEC 9126-1 (2001): Software engineering - Product quality - 1: Quality model.
[12] Juristo N., Moreno A.M., Sánchez-Segura M. (2007) Analysing the Impact of Usability on Software Design. Journal of System and Software. Vol. 80(9). pp:1506 – 1516
[13] Juristo, N., Moreno A.M., Sánchez-Segura, M. (2007) Guideliness for Eliciting Usability Functionalities. IEEE Transactions on Software Engineering, vol 33 (11). pp. 744-758
[14] Juristo, N., Lopez, M., Moreno, A. Sánchez-Segura. M (2003). Improving Software Usability Through Architectural Patterns. ICSE Workshop "Bridging the Gaps Between Software Engineering and Human-Computer Interaction". Portland, USA, pp. 12-19
[15] MDA Guide V1.0.1: http://www.omg.org/docs/omg/03-06-01.pdf, Last visit: March 2008.
[16] Mellor, S. J., Clark, A. N., Futagami, T. (2003). Guest Editors' Introduction: Model-Driven Development. IEEE Software. 20: 14-18.
[17] MOF QVT: http:// www.omg.org/cgi-bin/apps/doc?ptc/05-11-01, Las visit: March 2008.
[18] Morgan,T. (2002). Business Rules and Information Systems-Aligning IT with Business Goals.
[19] Mori, G., Paterno, F. and Santoro, C. Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions. *IEEE Transactions on Software Engineering*.
[20] Nielsen J..(2003) Return on Investment for Usability. Alertbox,. Http://www.useit.com
[21] Nunes, N. J. y J. F. e. Cunha (2000). "Wisdom: a software engineering method for small software development companies." Software, IEEE 17(5).pp. 113-119.
[22] Olive, A. (2005). Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. Proceedings of the 16th Conference on Advanced Information Systems Engineering, LNCS Springer-Verlag, Porto ,Vol. 3520, pp. 1-15.pp. 1-15.
[23] Pastor, O., Molina, J. (2007). Model-Driven Arquitecture in Practice. Valencia, Springer.
[24] Paternò, F. (2004). ConcurTaskTrees: An Engineered Notation for Task Models. The Handbook of Task Analysis for Human-Computer Interaction. D. Diaper, N. Stanton and N. A. Stanton. London, United Kingdom, Lawrence Erlbaum Associates.pp. 483-501.
[25] Pederiva, I., Vanderdonckt, J., España, S., Panach, I., Pastor, O. (2007). The Beautification Process in Model-Driven Engineer-ing of User Interfaces. INTERACT 2007, Brasil.
[26] Puerta, A., Micheletti, M., Mak, A. (2005). The UI pilot: a model-based tool to guide early interface design, San Diego, California, USA, ACM Press.pp. 215-222.
[27] Pyla , P., Pérez-Quiñones ,M., Arthur ,J. , Hartson, H. (2003). Towards a Model-Based Framework for Integrating Usability and Software Engineering Life Cycles. INTERACT 2003, eprint arXiv:cs/0402036.
[28] Reichart, D., Forbrig, P., Dittmar, A. (2004). Task models as basis for requirements engineering and software execution. Conference on Task models and diagrams, Prague, Czech Republic, ACM Press.pp. 51-58.
[29] Silva, P. P. d. and N. W. Paton (2003). "User Interface Modeling in UMLi." IEEE Software 20(4).pp. 62-69.
[30] Vanderdonckt, J., Q. Limbourg, et al. (2004). USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces. Proceedings of W3C Workshop on Multimodal Interaction WMI'2004, Sophia Antipolis, Greece.