www.ietdl.org

Published in IET Software
Received on 13th April 2011
Revised on 24th April 2012
doi: 10.1049/iet-sen.2011.0060

ISSN 1751-8806

Model and software tool for automatic generation of user interface based on use case and data model

I. Antović S. Vlajić M. Milić D. Savić V. Stanojević

Faculty of Organizational Sciences, Software Engineering Laboratory, University of Belgrade, 154 Jove Ilića, 11000 Belgrade, Serbia
E-mail: ilijaa@fon.bg.ac.rs

Abstract: The aim of this study is to identify the correlations between the use case model, data model and the desired user interface (UI). Since use cases describe the interaction between the users and the system, implemented through the user interface with the aim to change the state of the system, the correlation between these three components should be taken into account at the software requirements phase. In this study, the authors have introduced the meta-model of software requirements developed on the identified correlations. Based on this meta-model, it is possible to create a model of concrete software requirements, which enables not only the design and implementation of the user interface, but also the automation of this process. In order to prove the sustainability of this approach, they have developed a special software tool that performs the transformation of the model to an executable source code. They have considered different ways of user interaction with the system, and consequently, they have recommended the set of most common user interface templates. Thus the flexibility of the user interface is achieved, as the user interface of the same use case could be displayed in several different ways, while still maintaining the desired functionality.

1 Introduction

The starting point of creating any software product is the software requirements phase, and the results of this phase represent the basis of all other phases in the software life cycle. For this reason, the requirement specification, which is commonly represented by the use case model, should be detailed enough to provide the information necessary for the design and implementation of all elements of the software system; starting with the user interface, throughout application logic that encapsulates domain structure and system behaviour, to data storage. However, the requirement specification often misses the information required to design certain aspects of the system. In such cases, it is left to a programmer to interpret requirements and find out solutions, which can often lead to results which do not meet the user's needs, for example, when considering the user interface, one will find that the requirement specification rarely provides information about its appearance, content, types of components to be used and so on.

Hence use cases describe the interaction between the users and the system implemented through the user interface with the aim of changing the state of the system, the correlation between these three components should be taken into account at the software requirements phase.

This paper will present the results of research whose goal was to develop a model for the software requirement specification semantically rich enough to enable the automation of the user interface implementation process.

Considering the problem of designing user interfaces, we noticed different relationships between user requirements specification, data model and desired user interface.

First, we observed relationships between the desired user interface and user requirements. Looking at the existing techniques for gathering and specification of user requirements, use cases have been imposed as the most appropriate technique for their specification, especially if we want to achieve that certain parts of the software are created automatically. In this sense we have analysed the actions executed by an actor in the use case main scenario and the impact of these actions on the user interface.

By analysing user interfaces and their underlying data models, we noticed certain relationships that can be established between them. In this context, we have analysed entities, their attributes, relationships between them and their influence on the user interface.

Besides the influence of elements of use cases and data model on the user interface, we noticed that for the same use case we can design a variety of user interfaces, while still maintaining the desired functionality. This fact should not be ignored when automating the process of generating the user interface, because it would thereby lose flexibility in designing the user interface. The user should be able to choose the characteristics they want to apply on their user interface. These characteristics are not related only to the choice of graphical components, but to the way these components are organised, as well as to the additional functionality that are often seen on a user interface. To recognise these characteristics, we considered user interfaces

© The Institution of Engineering and Technology 2012

of several different software systems with different business domains and we defined several user interface templates. Consequently, flexibility is achieved, and the interested parties can try out different templates or the combination of templates during user requirement specifications to decide which one best suits their specific use case.

The meta-model defined in this paper represents the input for the software tool that enables generating the user interface source code. We will present the main characteristics of the generator as well as the calculations derived from its application on the real-life project. In doing so, we have taken into account both saved time and the proportions of requirements implemented by its application compared with those developed manually.

Also, we performed a comparison of this approach with existing approaches from the field and we presented the most important results of the comparison.

2 Background

From the user's point of view, the difference between the user interface layer and other layers of the software system is not clearly distinguishable. In fact, it is often completely unseen. For most users the software system represents a means to complete certain tasks. To them, the user interface is not simply part of the software system, but the system itself [1].

On the other hand, the system designer perceives the user interface as the realisation of the software's input and/or output. There are many different methods for software system development, whose mutual characteristic is dividing between the design of the system's functionality and the design of the way that the user interacts with the system [1].

The phases included in the software life cycle are the collection and specification of user requirements, analysis, design, implementation and testing. The design of the user interface starts only after the requirements phase has been completed or even after the design of application logic.

Contrary to that, there is another approach to development – one which suggests that the process of designing the user interface should be done parallel with the requirement specification. This approach is based on the way the user sees the software system, which is summarised by saying that 'the user interface is the system!' [1]. The reason for this is that the user is not familiar to the structural details and the ways in which the system is implemented. The requirement specification explains how the user wishes to execute a certain task using the software system. The execution of such a task entails an interaction between the user and the system, which forms a particular structure. According to this approach, the user interface should follow this structure so as to attain a higher level of software system usability. The early phases of creating a user interface often do not result in a fully functional user interface, but a prototype whose main goal is to emphasise the questions and presumptions that might not have been clarified using other approaches in requirements engineering [2].

A very important aspect in software development is the cost of a completion of software project. Given that the production of software is a series of intellectual and technical activities performed by highly-educated engineers, it becomes clear that the cost of the system greatly depends on the time and effort required for its production. The development of the user interface is a significant part of such a process [3]. A survey [4] found that 48% of a software system's programming code is related to the user interface, with about 50% of the entire development time being used for its implementation. Other research in this area has produced similar conclusions [5], regardless of the fact that tools which hasten this process and make it easier are utilised in the development process.

Owing to the fact that the user interface represents a direct link between the user and the system, the user often alters the requirements regarding its appearance and structure during the development process. These alterations occasionally cause changes to be made to the already implemented system and often result in the discarding of the interface developed until that point. This means that the entire process must be started from the scratch. However, the user may find that even the new solution fails to meet their needs, as they often find it difficult to precisely define the requirement specifications.

All these facts impose a strong need for the automation of the process with a goal is to devise a solution to speeding up the development of the user interface and reducing its costs. The resulting user interface must be based on user requirements. This tool should make the requirement specification process more effective. The user should then be able to identify quickly deficiencies in the requirement specification and remove them, enabling the system to meet their needs as closely as possible.

3 Software requirements: the basis of developing a good user interface

The significance of software requirements in software development is best illustrated by the research [6] which shows that half of the factors that influence the success of a software project are linked specifically to software requirements. Thereafter, this paper pays a great attention to the process of collecting and specifying the software requirements.

An important factor in the collection and specification of software requirements is the techniques employed in this process. The method chosen for software development often dictates which technique is to be employed, and the way in which the results of the applied technique are to be interpreted.

The most widely used techniques for the collection of user requirements today are use case specification [7, 8], and user stories [9]. The use case specification and user stories share certain characteristics (they are based on the scenarios and are oriented towards user's aims), but also differ in a number of ways (they vary in structure, the range of requirements covered, integrity, feasibility and how detailed they are) [7, 9, 10]. The goal of both techniques is to specify the user's intentions, that is, what the user wishes to achieve through a desired functionality. Considering the two aforementioned techniques for the collection and specification of user requirements, we can conclude that the use case specification presents a more acceptable solution than the user stories, especially when we wish to achieve automated development of certain software elements. The reason behind this is the use case's structure, integrity, as well as the possibility to include additional details significant to different system aspects. The literature often suggests the use cases as the best choice when using the model driven architecture approach [11].

3.1 Problems in specifying software requirements

Choosing a technique for the software requirements specification does not solve all the problems that can occur

© The Institution of Engineering and Technology 2012

doi: 10.1049/iet-sen.2011.0060

Table 1 Time required to make changes in different phases (in hours) [14]

Requirements collection	Design	Implementation	Testing (during development)	Acceptability test	Working system
1	3–6	10	15–40	30–70	40–1000

during this process, but can later have significant consequences. One of the most common problems that occur during the use case specification process is the existence of ambiguities in use cases. The literature identifies multiple reasons for ambiguities and gives recommendations on how to lessen the chance of their occurrence. The reasons are complex logic, negative requirements, omissions, boundaries and ambiguous terms [12].

All ambiguities in software requirements must be identified and removed as early as possible in order to avoid additional effort and save time (Table 1). Should ambiguity only occur once the implementation phase has been already reached, the necessary changes must be made, which is often very complex, time-consuming and hard to trace [13].

3.2 Connection between software requirements and data models

If ambiguous requirements were the enemy, the domain model was the first line of defence! [15].

Numbers of software development methods suggest designing the domain model of the system after having employed the use case specification.

In the object-oriented applications, states of the systems are encapsulated in the domain objects, which, along with relationships and constraints make the domain model of the software system. A domain model describes a static structure of the software system, that is, the basic concepts of the business problem domain being developed. This model is also called the ‘conceptual model’ and is constructed in the early, analysis phase of the software life cycle. The domain model represents the basis for the further design of the data model that will form the foundations for designing the data storage.

In order to avoid the ambiguities in user requirements and all their inherent perils, this research will recommend a slightly different approach to the use case specification and to the development of the domain model. According to this approach, the use case specification will be based on a previously developed domain model, instead of vice versa. Therefore before employing the use case specification it is necessary to construct an initial version or ‘sketch’, of the domain model. The domain model will become the starting point of the use case specification. This does not mean that the starting domain model will remain unchanged until the end of the requirements phase. On the contrary, when the new details of user requirements are identified during the use case specification, which were not included in initial domain model, they should be added to it. The domain model will be the foundation for designing the static elements of the system, while the use case specification will be used as a basis for designing dynamic elements of the system. The static elements of the system represent its structure, whereas dynamic parts determine the way the system functions. This will enable the static (domain model) and dynamic (use case) parts of the system’s model to be connected from the very beginning of requirements phase, which will preserve their consistency. This fact is crucial for the next phases of development if they are to relies directly on the use case specification (which is the case with all use case driven methods). In this context the domain model functions as a vocabulary of terms which will ensure their consistency when describing the problem domain. The literature [16] suggests this kind of integration of the use case and the domain model as a way of ensuring validity of software requirements.

3.3 Use case structure

As mentioned earlier one of the advantages of the use case specification over other techniques for the collection and specification of requirements is its structure. This structure is not strictly defined, and different authors present the use case specification in different forms.

Today’s literature lists hundreds of different formats or templates for use case specification. These templates are a result of the need to standardise the use case specification, in order to ensure that the specification is complete and that the communication between team members in charge of the specification is comfortable. Each of these templates is made for a specific purpose, so it is difficult to speak of a single template that could be qualified as universally acceptable. Still, the template chosen for a specification has a great impact on the usability of a use case specification during the design phase. It has also impact on the completed software. As far as the user interface design is concerned, using certain templates makes the process of constructing a good user interface easier, while other templates provide no useful information for such development or might even make it more difficult [17].

The use case specification templates describe structural elements that each use case should contain. However, none of the these templates describes in detail the structure of the main scenario or the structure of scenario steps, except for some templates which recommend the numeration of the steps or the separation between the actions performed by the user and the actions executed by the system.

The literature [7, 15] suggests that all steps in a scenario should be written in active voice, and in sentences with a ‘noun–verb–noun structure’. The first noun signifies the doer – the user or the system as the subject of the sentence (depending on whether the action is performed by the user or the system), the verb represents the predicate of the sentence and denotes the action being executed, while the second noun marks the object of the action being performed.

As we mentioned earlier in this paper there is a need to link the domain model and the use case specification, and we have concluded that the use case specification should be based on the previously developed domain model. When studying the steps in the use case scenario (use case scenario is also called a use case instance [8, 18]) in greater detail, one can notice that the object of the action (the second noun in the noun–verb–noun structure) is represented by a domain model entity or the attribute of an entity, depending on the level of detail used when employing a use case specification. This is how a connection between use cases and the domain model is made, that is, the data model to be created based on the domain model. In this way, each use

© The Institution of Engineering and Technology 2012

Table 2 Action types in use case scenario steps

User actions	preparation of input data
	input
	selection
	sending of requests and data to the system
System actions	execution of operation changing the internal state of the system, whose prerequisite is the execution of data validation
	displaying of results of the executed operation to the user

case can clearly be linked to an entity set to which it relates. Each entity set represents a sub-set of the domain model entities, which clearly defines the boundaries of use cases from the aspect of system structure.

The literature [8, 19] often explains the action performed in a use case scenario step as a transaction in the communication between the user and the system. Jacobson *et al.* [8] note four types of actions used to describe a scenario step. Owing to a need for better requirement specification from the aspect of user interface, we will introduce further two types which relate to the preparation of input data, which makes the complete list of actions appear in this way (underlined) (Table 2).

4 Connection between use case specification and the user interface

In the literature, we can often find the recommendation to omit information about the user interface from the use case specification [18]. As one of the most common mistakes in the use case specification, authors mention existence of details related to the user interface. This recommendation comes from the legitimate concern that details related to the user interface can distract the interested parties from the core of the problem being analysed when specifying software requirements.

Information about the user interface is of no importance when describing the goal and the aim of the user, while the user's goal and aim are essential for use cases. However, the fact that information about the user interface is so often found within a use case specification testifies to the necessity of establishing clear relations between the user interface design and the interaction between a user and a system as described by use cases [20].

By analysing use case specifications and the user interface designed for these specifications, we have identified certain connections between the two.

The first connection relates to the execution order of user-initiated steps in the use case scenario. The user interface is meant to enable the communication between the user and the system in a way described by the use case specification. When considering this, it seems only natural that the ordering of the components on the user interface should be arranged in such a way that the user is provided with a logical sequence for operation execution; one which matches the desired description illustrated in the use case specification.

The second connection between the elements of the user interface and the use case specification relates to the very type of action in each step of the main scenario executed by the user. The different types of actions performed by the user, as well as the actions executed by the system, have already been mentioned in this paper. User actions pertain to the preparation of input data (where we noticed two types of actions: 'input' and 'selection'), as well as 'sending requests and information to the system' (the third

Table 3 Graphical components which match the different types of scenario actions

Action type	Graphical component type
input	text field
	check field
	date entry field
	table
selection	drop-down list
	table
	radio buttons
	list
	tree
sending requests and information to the system	button
	menu or menu entry
	event defined for a component

type of user actions). Table 3 shows the graphical components that are usually used for the realisation of different user actions. It is important to emphasise that a particular action can be performed using other components and the same time presented components can be used in a different way, that is, different actions. Typical example is the table component that can be used not only for data entry or selection but also for grouping other components that can then be used for realisation of any kind of actions.

The connection between the elements of the use case specification and the user interface elements is illustrated in Fig. 1.

5 Connection between the data model and the user interface

By analysing data models, as well as the user interface designed for these models, we have identified certain connections between them. While observing data models, we considered the effects of the basic data model elements (entities, their attributes and the relationships between them) on the characteristics of the designed user interface.

5.1 Attributes

The attributes of the data model elements in terms of designing a user interface should not be observed independently from use case specification. The data model is designed with the goal of enabling permanent storage of data, so that the information significant to the user can be stored. Besides, additional information is also kept in order to ensure the consistency of data or to keep track of the changes to the data. All of the information which is not covered by the use case specification should be hidden from the user, as it can cause confusion without presenting anything useful to the user when executing the use case.

© The Institution of Engineering and Technology 2012

doi: 10.1049/iet-sen.2011.0060

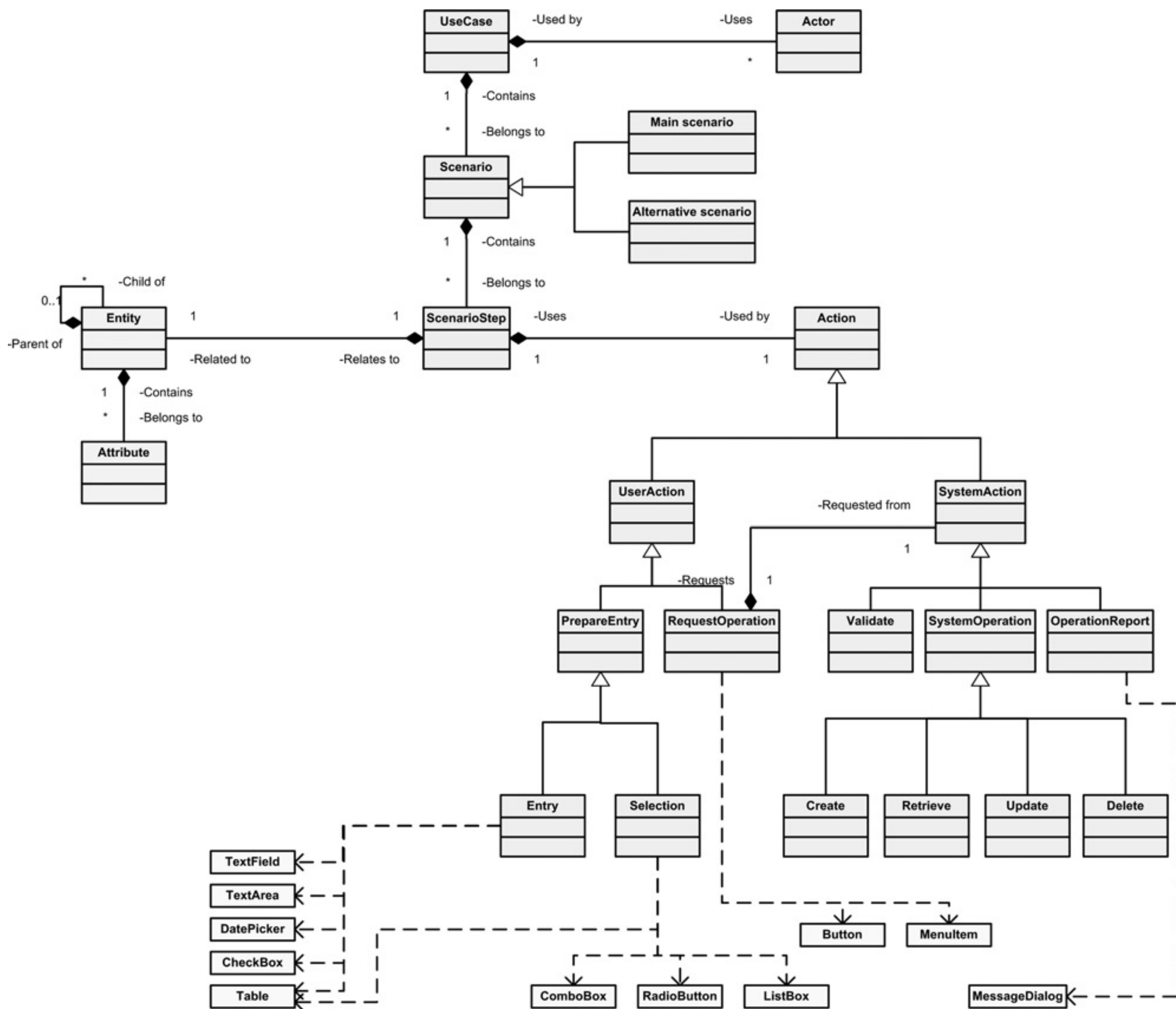

Fig. 1 *Connection between the elements of use case specification and the user interface*

5.2 Attribute types

Regarding the attribute types, it is also possible to establish connections with certain components of the user interface. Some components are, to a lesser or greater degree, suitable for the output or input of a specific type of data. For example, in the case of logical data ('Boolean'), the use of a 'text field' where the user is expected to enter a value (true/false, yes/no, 0/1 and so forth) is rare. A component designed for this purpose (a 'checkbox', a group of 'radio buttons', a 'combo box' with preset values) will be displayed instead. In the case of text data, a 'text field' is the most commonly used component (with one or more lines of text, depending on the size of the data), while attributes which represent dates have their own components designed for exactly those purposes [different kinds of components for the input or selection of date ('date-picker')].

5.3 Relationships

Data model relationships have also a significant role in designing a user interface. A *1*–* relationship (one-to-many) means that for the instance of *1* type of entity, there exists a number of attached instances of entities from the * side. Let us look at the example with the 'Invoice' and 'InvoiceItem' entities, where the use case for the entry of Invoice implies the entry of InvoiceItem as well. Often, the user interface will display entries for the * side (the InvoiceItem side) as a separate sub-form to the form created when Invoice entries are made. The sub-form can be displayed in the same window or a separate one, taking care that a means to synchronise data with the initial window should be insured.

This sub-form should enable the entry of multiple instances of InvoiceItem for a single Invoice, so a graphical component which shows data in the form of a table can be used for that purpose, with each row in the table presenting a different instance of InvoiceItem for the currently active Invoice, displayed in the main form. This relationship is often called the parent–child relationship, where the 'parent' can have more 'children', while each 'child' has only one 'parent' (Fig. 2).

However, the child does not always have to be displayed in the form of a table. The one-to-many relationship can be realised in such a way that in the sub-form, for data input for the child, instead of a table, it has text fields, date fields or combo boxes, which are used to enter data for one child. If the sub-form is realised in this way, besides the graphical components used for entering data regarding a single child, it is necessary to set up a way for the user to navigate

© The Institution of Engineering and Technology 2012

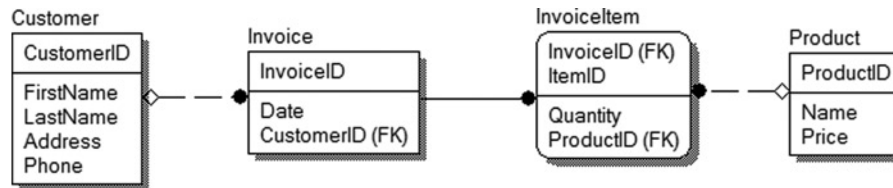

Fig. 2 *Graphical example of the parent–child relationship between entities*

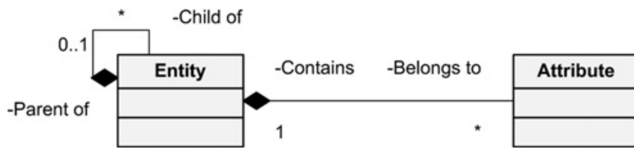

Fig. 3 *Example of a 1–* relationship between the Invoice and InvoiceItem entities, as well as a *–1 relationship between the Customer and Invoice entities*

throughout the existing list of children. In the example of Invoice–InvoiceItem, the user needs to be able to access all of the InvoiceItem for the active Invoice. Most cases this navigation implies buttons for shifting to the next, previous, first and last instance of InvoiceItem. Providing access to a specific 'item' should also be possible.

When the model includes a *–*1* relationship (many-to-one; in a relational model, an attribute from side *1* is treated as a foreign key to a relation from side *), the entity on side *1* is most often displayed in some kind of list (combo box, table, and a group of radio buttons). If we consider the previous example, and add to it by saying that each invoice is given to a specific 'customer', and *1* Customer can be issued with multiple Invoices, then the relationship between Invoice and Customer is a *–*1* relationship (Fig. 3). When entering data in Invoice, it is necessary to choose one of the existing Customers to whom the Invoice is issued. Then the list from which the Customer is chosen will be populated with data belonging to the Customer entity, where each of the items in the list is one concrete instance of the Customer entity. Besides, use case specifications often predict a situation where the Customer list contains no Customer to whom an Invoice is to be given or where the aim is to allow the user to gain insight into Customer details, usually not visible when showing Customers in the list. Therefore it is useful to enable user to change these fields (by adding new, altering and deleting existing Customers) or to review the data on a Customer, selected from the list. This creates a relationship between different use cases (as the manipulation of data about Customers is most often presented by a separate use case).

6 Identifying the user interface templates

By analysing the user interfaces of several different software systems, we have noticed recurring characteristics which are independent of the concrete domain for which these systems were designed. These characteristics are not related only to the types of graphical components being used (the type of components to be used will mainly be determined by the type of action from a use case specification, and the data model elements, in the previously explained way), but to the way these components are organised in the user interface. We have particularly focused on the way of placing the graphical components, grouping components in logical units and noticing the relationships between these units, and then identifying the additional functionality that is commonly present, such as navigation between components, the possibility of more detailed view of certain data on the user interface etc. To obtain these common characteristics, we considered user interfaces of several different software systems with different business domains. In order to reach objectivity of results, we have considered not only the software systems that we designed, but also the systems of independent authors that have been seconded to us for the research purpose. We have analysed the system related to public administration domain (e-auctions), information system of a manufacturing company (Perihard inzenjering), a medical institution, system for monitoring the scientific work of teachers (Faculty of Organisational Sciences (FOS)) and master students service (FOS). Besides these systems, many characteristics were gathered from MS access [21] user interface wizard, which we once used to develop our prototypes with. The identified characteristics were basis for defining several user interface templates: field-form, field-tab, table-form, table-tab and kind of (un)normalised graphical user interface (KNGUI) template.

A single use case can be realised through different templates. By using the tools for automatic user interface generation, created for the purposes of this research, it is possible to obtain quickly and easily a completely different user interface for the same use case by making simple adjustments of the selected template in the model. In this way, the interested parties can try out different templates during user requirement specification phase, and decide which one best suits their specific use case. The templates are designed in such a way that they can be combined, which means that different templates can be used for both a single use case and its parts.

In the previous section we have explained that the *1–** relationship between entities in the data model is displayed by using a basic 'parent' form for the entity on side *1*, while the entities on side * are displayed in sub-forms. Each sub-form can be realised through a different template. The following diagram displays the relationship between different use cases, data models, templates and the graphical components used by templates (Fig. 4).

If the use case processes an entity with multiple levels of depth (the parent has children, while the children have their own children and so forth), it is possible to use a different template on each level for each child. It is advised to be cautioned when introducing new levels of depth, because the user interface tends to become too complex in those cases, which can confuse the user. Of course, this recommendation does not pertain to the user interface templates, but the use case specification, while the problem of the over complexity user interface is simply a consequence of a bad use case specification. The cause of a bad specification is not necessarily the project analyst. Instead, the reason is most often lays on interested party that demands certain functionality. In these cases, the analyst can point out the potential problem, but can also

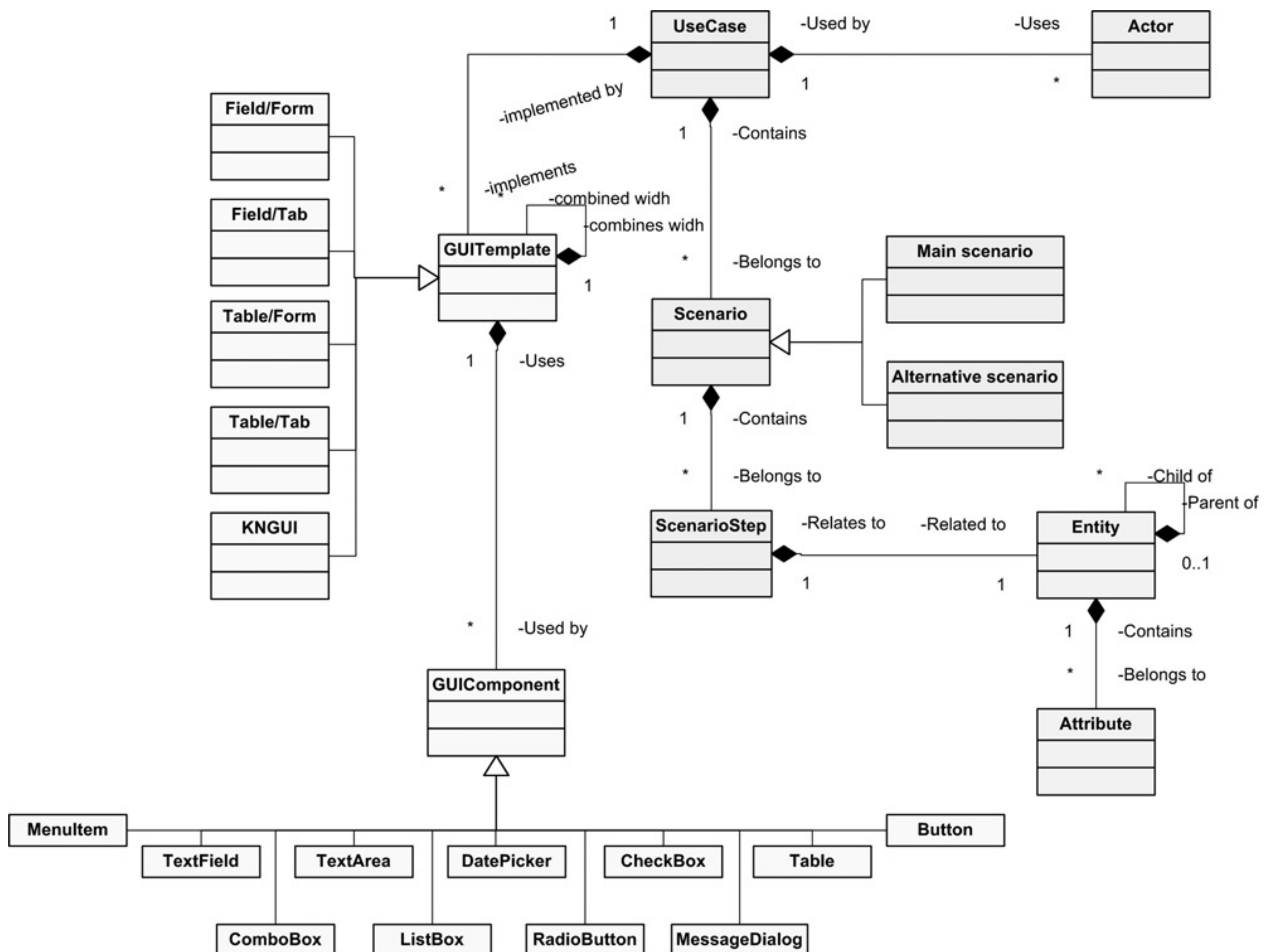

Fig. 4 *Relationship between different use case elements, data models, templates and the graphical components used by templates*

suggest that a different template combination can be used, that is, one which can lessen the complexity of the user interface.

The following part of this paper will provide an example of combining different templates into multiple depth levels. This example demonstrates one of the solutions, attained by automatically generating a user interface by utilising a developed code generator for a highly complicated use case, which processes the entity in four levels of depth. The example shown is a part of the user interface designed for the project Kostmod 4.0, which was implemented for the needs of the Royal Norwegian Ministry of Defence. Even after informing them on the potential problem, the customer decided not to deviate from their original plan. In the end, the problem was solved by combining a number different user interface templates (Fig. 5). The diagram shows a simplified data model, that is, the part of the data model to which the aforementioned example refers to (Fig. 6). The characteristics of each defined template, their links to the use case specification and the data model entities they refer to are explained in the following sections.

6.1 Field-form template

The field-form template was devised in such a way as to enable the input and output of information related to a specific entity through the use of fields, that is, graphical components used for the entry or selection of individual data related to the attributes of the entity being processed, while the entity's children are displayed in separate windows. Each child can be displayed through a different template. In this template, the following components can occur: 'Label', 'Text-field' (consisting of one text line; known as a 'text-area' if it contains more than one text line), 'combo-box', 'navigation' buttons (in the form of a 'toolbar' or regular 'buttons'), buttons for insert/update/delete options, 'date-picker', 'check-box' and 'option buttons' that are used for displaying the windows containing information about children.

Which of these components will be displayed depends on the types of actions specified in the use case scenario, as well as on the data model itself, which was already mentioned in the previous sections. Fig. 7 illustrates a user interface for a simple use case which includes information pertaining to one entity (without children), which is realised using a field-form template. The figure shows one of the ways for displaying navigation buttons, as well as options for inserting, updating and deleting ('toolbar' at the bottom of the window).

The insert operation (inserting a new record) is realised by bringing up a new window in which the user can enter new data, and then save it or cancel the entire process.

This simple example shows that the data 'User right' is shown as a combo box. This is a consequence of the action type 'choose' from the use case specification, which is related to the $*-1$ relationship of the data model, that is, related to a foreign key in a relational model. The simplified data model that demonstrates the relationship between these entities is illustrated in the following figure (Fig. 8).

© The Institution of Engineering and Technology 2012

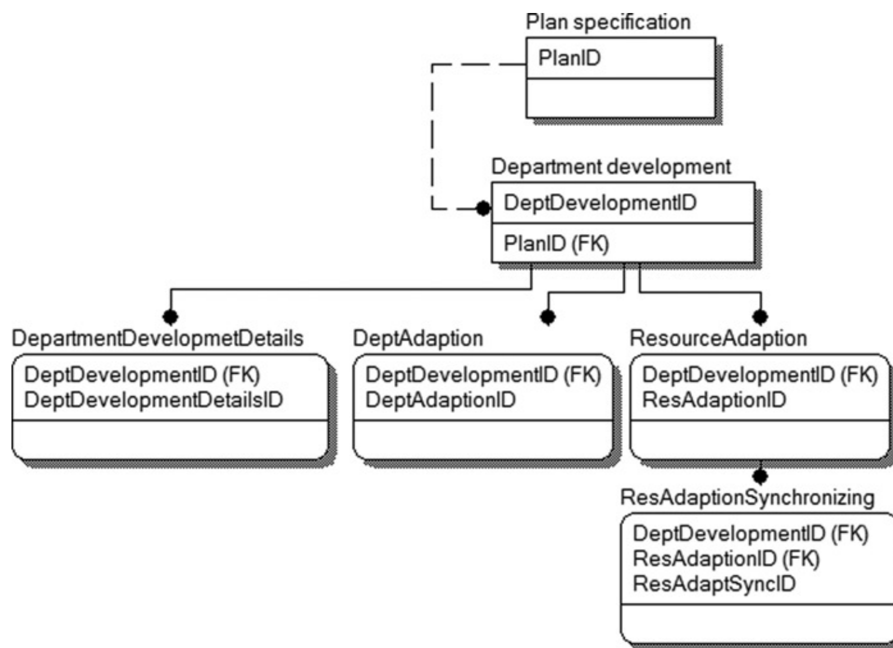

Fig. 5 *Example of combining different user interface templates with four levels of depth*

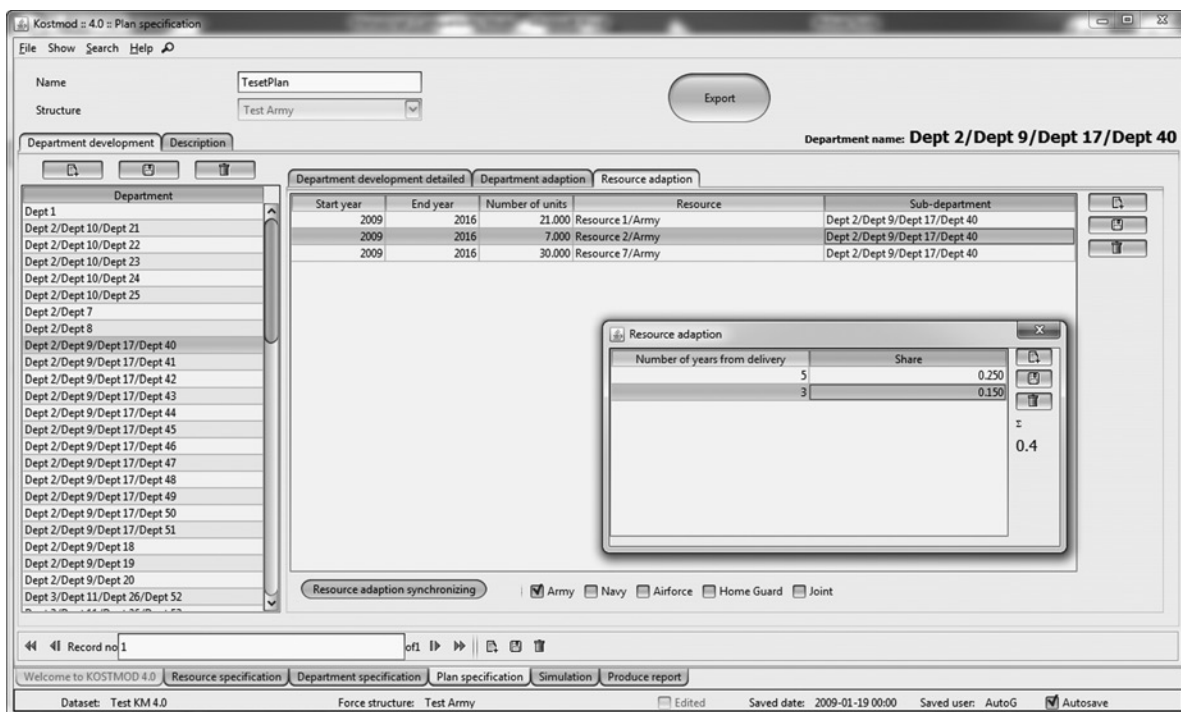

Fig. 6 *Part of the data model illustrating the example of combining different UI templates*

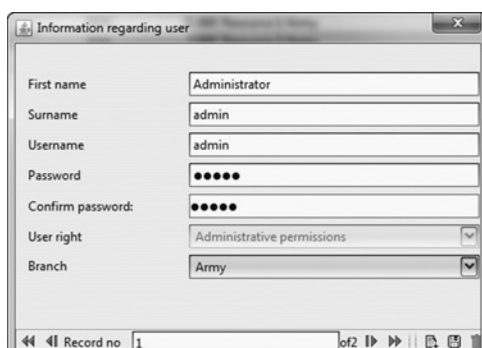

Fig. 7 *Field-form template: one entity without children*

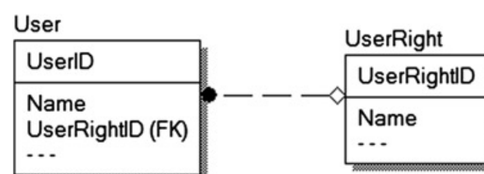

Fig. 8 *$^{*}-1$ relationship, which is one of the prerequisites for using a combo box*

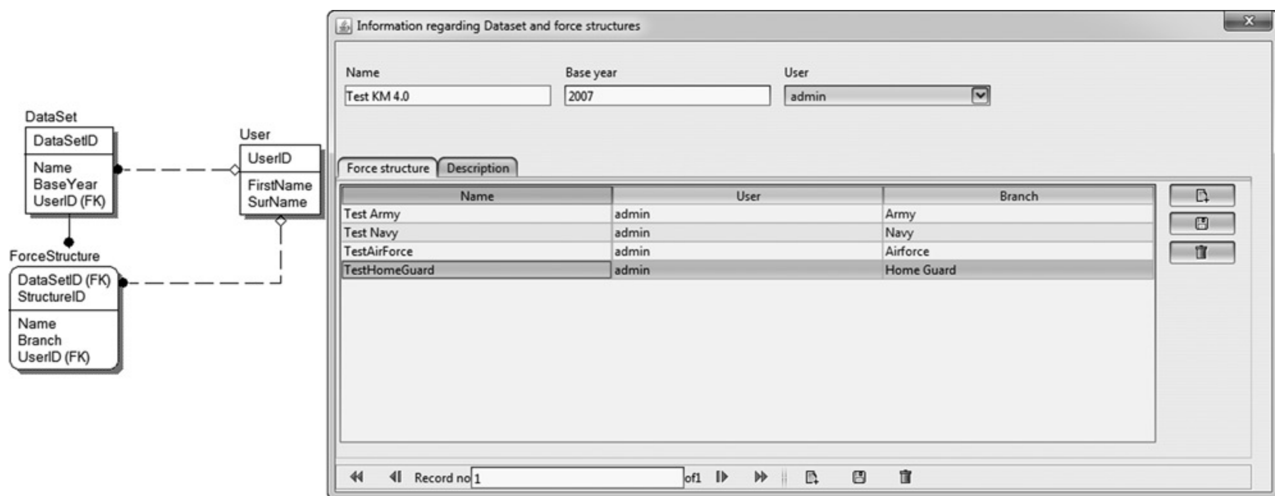

Fig. 9 *Part of the data model and a user interface to display the field-tab template*

Should the user find that the combo box does not contain the required piece of information when inserting or updating (or wish to see more detailed information about one of the elements), it is possible to enable the user to enter a new element into the list or see the details of the selected element. This function is executed by right-clicking on the combo box and selecting the option Show details. However, it is necessary to define the desired behaviour in the user interface generation model.

Inserting, updating or deleting of data attributed to children in this template is enabled in the individual windows designated for each child. These windows are brought up by marking a specific ‘option button’. As long as the button is marked, the window containing the data about the child will be displayed.

6.2 Field-tab template

The field-tab template, much like the field-form template, was devised in such a way as to enable the input and output of information related to a specific entity through the use of fields, that is, graphical components used for the entry or selection of individual data related to the attributes of the entity being processed, while the entity’s children are all displayed in the same window, but in different tabs. Each child can be displayed through a different template, which is specified in the user interface generation model.

Graphical components used in this template, apart from the ones used in the field-form template, include the ‘Tabbed-Pane’ component which is used for displaying panels that contain information about children. By navigating through the basic entity using navigational buttons, the data in the tabs are refreshed, in order to reflect the child data for the currently selected parent entity.

Fig. 9 illustrates a simplified data model and, following that, an example of a user interface realised by using the field-tab template for the selected data model. This example shows a further functionality, that is, the primary keys, if assigned auto-incrementally, can be hidden from the user. All that is needed is for this functionality is that this option is selected in the user interface generation model.

The field-tab is different from the field-form template in the way in which a certain entity’s child information is displayed. If the use case contains just one element, without children, it does not make a difference which of these two templates is chosen in the user interface generation model.

6.3 Table-form template

The table-form template is devised in such a way to enable the input and output of information about a specific entity in the form of a table, while the children of the entity are displayed in separate windows by pressing the appropriate option button. Each child can be displayed using a different template.

This template uses a ‘table’ as the basic graphical component, and is suitable in situations where the user wishes to have an overview of information about the multiple instances of a single entity. Field-form and field-tab templates can only display information about a single instance of an entity at the time, while other instances of the same entity are only available through the use of navigation buttons. In the table-form template, data are shown in a table that enables the user to navigate easily by simply selecting a row in the table. When selecting a row, information in child windows is refreshed in order to reflect the selected parent entity.

Besides displaying data in a table form, entry of information and updating is enabled through the use of a table. Changes are made directly by selecting one of the rows, and data is saved either by calling the save option or selecting another row. At this point, the user is notified that changes have been made and will be prompted to either save the changed or revert to the data in its previous state (if the auto-save option is turned-off in the user interface generation model). If the data are invalid the system will not permit switching to another row until the data are corrected or reverted to their previous state.

When inserting or updating, the appropriate graphical component will be displayed, which depend on the action type specified in the use case step or the type of attribute. This is realised by different ‘cell editors’ that are adjusted to use different graphical components for editing data.

The function which enables reviewing and updating the detailed information about an element of a combo box (show details) is also present in this template. It is executed by right-clicking on the cell containing the combo box.

6.4 Table-tab template

The table-tab template, like the table-form template, is devised so as to allow the input and output of information about a certain entity in a table form, while the children are

© The Institution of Engineering and Technology 2012

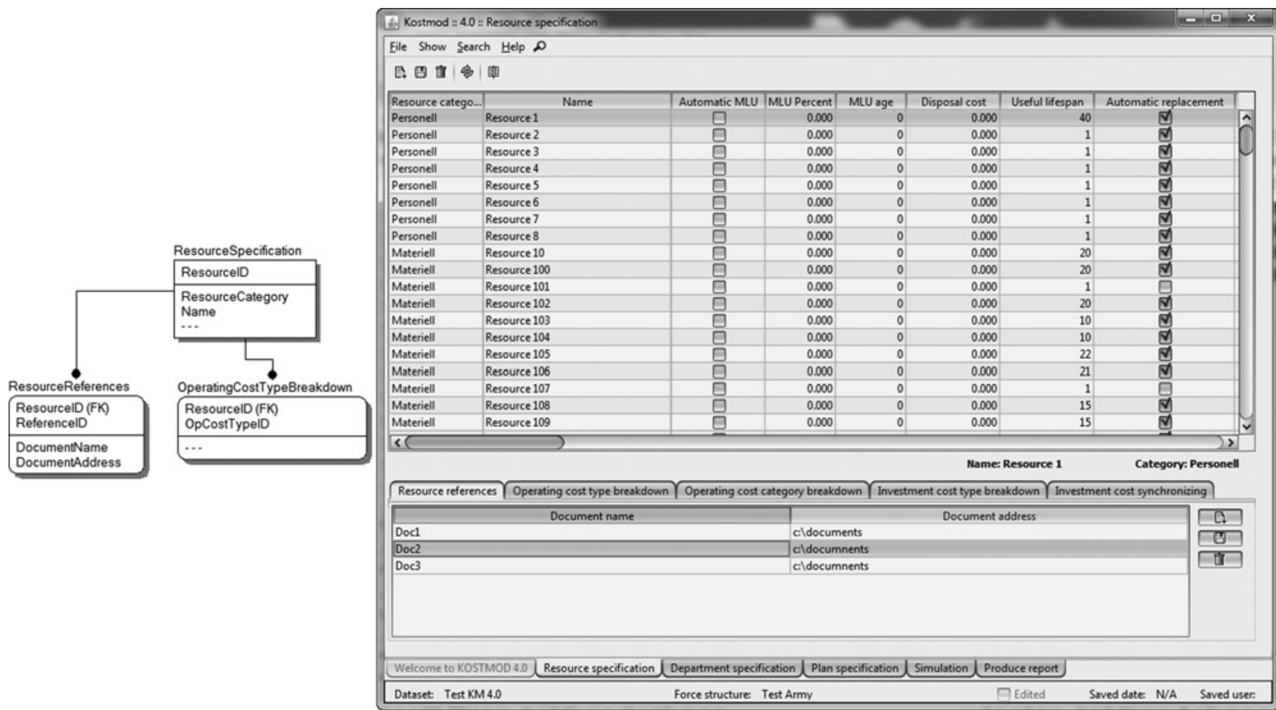

Fig. 10 *Part of the data model and a user interface demonstrating the table-tab template*

displayed in the same window, but in different tabs. Each child can be displayed using a different template, which is specified in the user interface generation model.

The graphical components used in this template, apart from those used in the table-form template, include the 'Tabbed-Pane' component used to display information pertaining to children. When navigating through the basic entity by selecting a specific row in the table, information displayed in the tabs is refreshed, so that the displayed children data reflects the selected parent entity.

Fig. 10 illustrates a simplified data model and an example of a user interface realised using the table-tab template for the displayed data model. This example also shows the utilisation of a cell editor that is adjusted for logic data type.

The table-tab is different from the table-form template in the way in which a certain entity's child information is displayed. If the use case contains only one entity without children, it does not make a difference which of the two templates will be chosen for the user interface generation model.

Fig. 10 shows a different way of displaying options for inserting, updating and deleting by the use of a toolbar component. This manner of display is produced by specifying the corresponding parameter in the user interface generation model. It is suitable for the situations in which components for data display (in this case – tables) take a great amount of space on the user interface. The possibility of displaying options in this way is not limited to this template alone, but can be used in all other templates as well. The appropriate template can be chosen for each case individually.

6.5 KNGUI template

The KNGUI template is devised so as to enable the entry and output of information about a specific entity in table form. The model enables possibility to select children to be displayed in the same table as parent entity. The remaining children can be displayed either in tabs or separate windows, using various templates.

This template is suitable in situations where the children of the entity represent the aggregation of that entity with other. In these cases, the table – whose columns show all the attributes of the entity being processed – is dynamically populated by columns representing all the instances of all aggregated entities. A cell which is created by such a merging will display the value of an attribute on the aggregated entity, the attribute specified by the model. The table generated in this way and displayed on the user interface completely transforms the data model which exists in the database and, in fact, represents the merging of at least three tables of a relational data base.

This template uses the table as the basic graphical component and is suitable in situations in which the user wishes to have an overview of information related to multiple instances of a single entity, and provides the possibility enter data about the entity's children easily, without taking additional space on the user interface. In order to display data pertaining to children, table-form and table-tab templates require panels to be shown in either the initial or separate windows. Also, the user is not required to move between panels in order to enter different data. Instead, all data are entered into the same table, which significantly speeds up the process.

This template is not limited to processing only one child, but can instead include an arbitrary number of entities which represent the children of the initial entity. Caution is advised when choosing how many children will be included in the display, as this can cause the table to become too wide, which reduces visibility. Should the table used for aggregation contain a large number of records, the issues of visibility may arise.

A similar manner of displaying data was made possible by using the cross-tab or pivot instruction within the SQL query itself, within the Microsoft SQL Server database management system. The pivot is defined as an order which enables the

© The Institution of Engineering and Technology 2012
IET Softw., 2012, Vol. 6, Iss. 6, pp. 559–573
doi: 10.1049/iet-sen.2011.0060

displaying of the row values in a table as columns in the resulting SQL query table [22]. It is most commonly used for displaying the results of an aggregating function (SUM, COUNT etc.) as values in a table.

The KNGUI template is different from the previously described approach because it enables not only the transformation of table data within a relational database with the goal of displaying these data differently, but also allows the use of operations that make changes to the displayed data that are then reflected on the data in the database. Each operation then transforms the displayed data into a set of objects whose altered state is kept in the database by the use of various SQL queries, which depends on the changes made.

7 Software requirements specification model which enables the automation of the user interface implementation process: the user interface generation model

The goal of this research was to develop a model for the specification of software requirements that would be semantically rich enough to enable the automation of the process of implementing the user interface.

The basis for defining such a model are the connections between the use case model, the data model, and the desired user interface which were all noted in the previous sections. Considering the way in which use cases describe the interaction between the user and the system, which is realised by the user interface with the goal to change the state of the system, the connection between these three components must be represented appropriately in order to enable the automation of the user interface implementation.

When defining this model, one must make sure that the model follows good practice in the specification of software requirements, which means that it should provide a frame for the specification of requirements that will be understandable to all interested parties, from engineers to end users. Furthermore, ambiguities in requirements should be kept to minimum in order to prevent numerous problems described in previous sections.

The software requirement specification is supposed to contain information about the user's goals and aims in terms of executing system functions, and should not include implementation details, such as details about the user interface or the realisation of other parts of the system. In order for the specification to remain completely in the field of the problem, and not in the field of the solution, it is necessary to split these two fields through a model, and make the problem independent from the solution that can be usually achieved in one or more ways of realisation.

As the development of the model is based on the connections observed between the use case elements, the data model and the graphical user interface, a meta-model has been developed. The meta-model considers all relevant elements and the connections between them (Fig. 11).

Fig. 11 *Meta-model representing the connection between the elements of use case specifications, data model and user interface*

© The Institution of Engineering and Technology 2012

In order to enable the automatic generation of user interface through a model based on the illustrated meta-model, this model must be defined in a form which is suitable for manipulation by a Java program that will be used to realise a generator, but which is also readable and understandable by all interested parties in the process of requirement specification. XML is usually listed as the most suitable format for this purpose, as this language with a simple syntax is legible for both humans and computer programs.

7.1 Model specification: UseCaseModel.xml and UseCaseGuiExtension.xml files

The main objective of this paper is to present a requirement specification meta-model that enables automation of the process of generating the user interface. The way of specifying a model for the generation was beyond the scope of this paper. During the research, the specification was done mostly manually or using a tool made for research purposes, which facilitates the specification. The basic idea was first to clearly define relationships between use cases, data model and user interface, and to define the meta-model on the basis of observed elements and their interrelations, and only after that to shift focus on how to obtain the input specification, and thus to define the entire process of how to make and maintain applications by proposed approach.

In order to separate the specification of software requirements from the information pertaining to the user interface, the user interface generation model is specified by two separate XML documents. The first XML document represents the use cases, and part of the data model to which the use case relates, and these data are housed in the 'UseCaseModel.xml' file. The second XML document contains information about the elements of the graphic user interface, and every element of this document refers to a particular element from the defined 'UseCaseModel.xml' file. The content of the second document is specified in the 'UseCaseGUIExtension.xml' file. The existence of the second document is not mandatory. When the second document is not created, the default user interface elements (such as user interface templates and graphical components) will be used for displaying the use case described within 'UseCaseModel.xml' file. If the requirement is made that the user interface differs from the implied settings, these settings are specified in the 'UseCaseGUIExtension.xml' file.

8 Automation of the transformation process of the suggested model into executable programming code

In order to prove the sustainability of this approach, we have designed a software tool that conducts the transformation of the defined model into executable programming code. This will enable the creation of a user interface in the early phase of the software's life cycle. It gives the possibility to validate and verify user requirements, and drastically reduces the time and effort required to implement the user interface. This software tool enables generating of executable programming code for the interface of a Java desktop application.

During the design of generator we have taken into consideration all advantages and disadvantages of this approach in application development. The authors [23, 24] point out as the greatest advantages: reduction of programming errors, time reduction, cost reduction, standardisation of implementation, quality, customer satisfaction, patterns, code reusability and productivity. Disadvantages are usually related to not always applicable, application generators are difficult to build, recognising where an application generator can be used, influence on development process, startup cost, longer design phase, configuration management, process compatibility impacts, impacts to environment and other tools.

Nakatani *et al.* [25] emphasised that the business application systems carry out information storage, retrieval, updating and deleting. Owing to these characteristics they point out that most use cases can be put into several category groups: data storage, data retrieval, data updating, data list creation and documents creation and document transmission. In their research 53 of 58 use cases were categorised into these six groups. They developed tools [26] that provide basic use case frameworks: for example, Create, Read, Update, Delete and Making-reports, to help developers write down use cases and manage relationships between use cases and domain objects in order to keep consistency in requirements. These frameworks are expected to cover over 80% simple use cases in business application.

We obtained the similar results in previously mentioned Kostmod 4.0 project. We calculated that user interface for ~75% of total number of use cases were completely implemented using our approach and other 10% needed minor manual development adjustments. For other 15% of use cases some parts of UI were also generated but they required much more manual development adjustments. The whole project was released in 4 months, which was much faster than usually estimated 12 months.

8.1 Generator structure

The generator structure is made up of a collection of classes and interfaces weighted with different types of responsibility in the process of generating a programming code. The structure of the generator is shown in Fig. 12.

The user interface generator accepts the 'UseCaseModel.xml' and, if required, the 'UseCaseGUIExtension.xml' files as input. It interprets the content and performs the transformation of the specified model into an executable Java programming code.

8.2 Generated programming code

Following the transformation of the user interface generation model, the generator produces a programming code, and ultimately creates an application. The application is entirely specific to the problem area. In that sense, the central graphic component which is used as a container component for all other graphical components is the 'JPanel' class. One or more panels were created which depended on the chosen set of templates. In that way, the resulting application is not strictly connected to the way in which graphical components are displayed. 'JFrame' and 'JDialog' classes are most commonly used to display forms. However, the 'JPanel' class represents a generic, lightweight container component that can be transferred from one container to another with ease. Therefore, after the generation, JPanel components should be added to JFrame of JDialog container in order to be displayed. Currently, this part of user interface implementation process is not included in our code generation tool, because our main intention was to create a clear connection between use case model, data

© The Institution of Engineering and Technology 2012

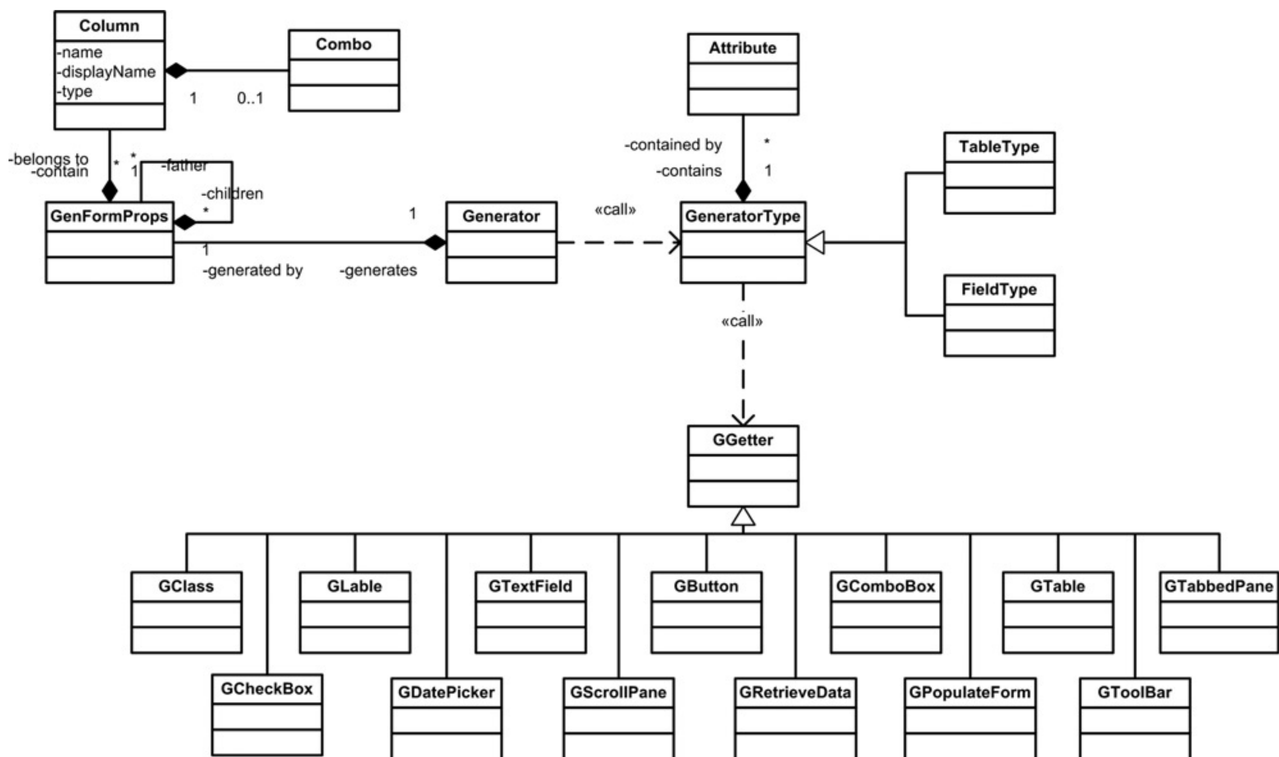

Fig. 12 *Class diagram representing the generator structure*

model and user interface. Furthermore, in most cases one needs to add user interface to already existing projects, with already defined policy for displaying graphical components. Still, this will be one of the directions for our further research.

As mentioned before, we have defined several templates or ways to display information on the form. Predefined templates are used when generating programming code to which graphical components are added. Based on this, we can conclude that identical data can be displayed in different ways (within text fields, inside tables, in separate windows etc.), with the graphical components found on the JPanel component. Panels created in this way can be opened with a graphical tool that supports visual editing of user interface.

The generated user interface enables invocation of business logic operations for the execution of basic operations on the database (adding a new record, changing a record, deletion of a record and selecting a record). These operations should be implemented in the business logic layer. In addition, most cases require definition of business logic that will execute some other operations (e.g. calculating certain sub-scores, complex rules of validation etc.). Also, the user interface and business logic can change over time, which leads to a need to alter the generated programming code. In this case, it is not recommendable to apply changes directly to a generated class (or generated panel). Instead, the generated class should be inherited, and care should be taken to situate all the specifics within that class. This will enable the forms to be re-generated, while the specific logic will remain intact.

User interface classes for each defined use case in the model will be created in individual Java package, into which all 'java' files are stored.

The generated forms possess no information on domain objects, which ensures the independence of the presentation layer of the application from the domain model.

In order to achieve this, we have used data transfer object (DTO) [27] pattern to make application layers as independent as much as possible. General idea was to make a class capable to carry heterogeneous data through layers. The DTO class uses HashMaps to store the data. Key-value pairs are attribute name–attribute value. Every top parent form has a method that maps data from graphical components to DTO object (retriveData()) and vice versa (populateForm()). It is important to note that complete structure which represents the data model displayed in the forms is stored in the DTO.

This enables a form to be independent from a domain class. In our research we have identified all necessary services that business logic has to provide in order to create a completely functional application (services necessary for create, read (retrieve), update, delete (CRUD) operations). These services are defined in the form of interface, which we implemented for the research purpose, in the same way as we implemented the persistence layer. Using this implementation is not mandatory and therefore user can provide different implementation. Details of how we implemented these layers are beyond the scope of this paper.

8.3 Data validation on the presentation layer

Using the 'UseCaseGUIExtensionl.xml' file enables generating the code responsible for the execution of validation on the user interface layer. Regardless of the validation on the user interface layer, validation rules are also often checked on the application logic layer. Consistency and data integrity checks are performed on the persistent layer. As a result of this, it is possible to block the processing of data in several locations if they do not meet the defined validation rules. Using validation on the user interface layer is useful due to the fact that the workload on the application logic is lessened, and the application logic is called only when all data are valid.

Validation rules refer to the data processed by the system. When speaking about the relational data model, validation

© The Institution of Engineering and Technology 2012

Table 4 Comparation of Apache Isis, Metawidget, WebRatio, AlphaSimple, BizAgi BPM Suite and SilabUI on defined criteria

	Apache isis	Metawidget	WebRatio	AlphaSimple	BizAgi BPM suite	SilabUI
input specification	domain model	domain model	WebML or BPMN model	textUML	BPMN	use cases and domain model
defining input specification	manual	manual	wizard	manual	wizard	manual
dependency on domain model	yes	yes	yes	yes	yes	yes
generates UI source code or generates UI on runtime	runtime	runtime	source code	source code	runtime	source code
template selection	no	no	yes	no	no	yes
generated source code can be changed	no	no	no	yes	no	yes
separation of UI from other layers	no	yes	yes	yes	yes	yes

can refer to the relation attributes, tables, as well as a number of linked tables. This means we can identify five types of validation rules: attribute type, attribute value, co-dependence of a single relation's attributes, co-dependence of different relation's attributes and structural limitations.

When generating a user interface, it is possible to generate a code responsible for validation, where rules that relate to the type of the relation's attribute can be automatically applied, based on the suggested model. All other validation rules can be added by expanding validation methods.

9 Related work

In order to compare our approach to existing work in the field, we conducted a survey in which our tool (with the working title SilabUI) is reviewed in relation with five existing tools. First, we have defined a set of comparison criteria that are important for the automation the user interface process, and then selected some of today's popular tools and approaches to compare our approach with. These tools are Apache Isis [28], Metawidget [29], WebRatio [30], AlphaSimple [31], BizAgi BPM Suite [32]. In order to achieve the objectivity of the comparison of selected tools, we have defined specific user requirement (whose domain model was shown in Section 5.3). This requirement with underlying domain model includes the operations and relationships between objects that we can usually meet in business applications. The result of the case study is six software applications, each developed using different tool. Then we carried out the evaluation of those tools according to defined comparison criteria, and main results are shown in Table 4 [33, 34].

In addition, there is another interesting approach that uses a tool for modelling UMPLE to generate executable user interface prototypes [35]. UMPLE allows you to create class diagrams and state machines specification. User interface prototypes are generated based on these models that can be specified both graphically and textually. The main advantage of this tool is the simplicity of making the input specifications, while the main disadvantage is the inability of choosing different user interface templates. It would be interesting to consider connecting our approach and one using UMPLE, especially the part of input specifications, in order to take advantages of both approaches.

10 Conclusion

This research considered the observed relationships between the elements of the use case specification, the data model and the user interface. Based on these perceived relationships, a meta-model of software requirements has been developed, the one which considers the elements of all three aforementioned system aspects. Using this meta-model, we can create a software requirement model which enables not only the design and implementation of the user interface but the automation of this process as well. Potential further research in this area could include (i) the examination of the applicability of the meta-model to various technologies and platforms such as desktop and web applications in different technologies etc.; (ii) the enrichment of the meta-model and the generator in order to ensure support for the realisation of various architectures of the resulting application; (iii) the enhancement of the model with the possibility to specify more complex validation rules; and (iv) ensuring the adaptability of the user interface.

11 References

1 Van Vliet, H.: 'Software engineering: principles and practice' (John Wiley & Sons Ltd, Chichester, West Sussex, England, 2008, 3rd edn.)
2 Pfleeger, S.L., Atlee, J.M.: 'Software engineering theory and practice' (Prentice-Hall, 2006, 3rd edn.)
3 Kennard, R., Leaney, J.: 'Towards a general purpose architecture for UI generation', *J. Syst. Softw.*, 2010, **83**, (10), pp. 1896–1905
4 Myers, B., Rosson, M.: 'Survey on user interface programming'. ACM: Human Factors in Computing Systems, Proc. SIGHI, 1992, pp. 195–202
5 Kivistö, K.: 'A third generation object-oriented process model, roles and architectures in focus'. Academic thesis, Faculty of Science, University of Oulu, Oulu, 2000
6 Johnson, J.: 'Turning chaos into success', *Softw. Mag.*, 2000, **19**, (3), pp. 30–39
7 Cockburn, A.: 'Writing effective use cases' (Addison Wesley Longman Publishing Co. Inc, Boston, 2000)
8 Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.: 'Object-oriented software engineering – a use case driven approach' (Addison Wesley Longman Publishing Co. Inc, Boston, 1993)
9 Cohn, M.: 'User stories applied: for Agile software development' (Addison Wesley Longman Publishing Co. Inc, Boston, 2004)
10 Stellman A.: 'Requirements 101: user stories vs. use cases', available at http://www.stellman-greene.com/2009/05/03/requirements-101-user-stories-vs-use-cases/, Building Better Software: Jennifer Greene and Andrew Stellman weblog, published: 3. 05. 2009, accessed: 1 July 2009
11 Fatolahi, A., Somé, S.S., Lethbridge, T.C.: 'Towards a semi-automated model-driven method for the generation of web-based applications from use cases'. Proc. Fourth Model-Driven Web Engineering Workshop, Toulouse, 2008, pp. 31–45
12 Wiegers, K.E.: 'More about software requirements: thorny issues and practical advice' (Microsoft Press, 2006)
13 Lin, L., Prowell, S.J., Poore1, J.H.: 'The impact of requirements changes on specifications and state machines', *Softw., Pract. Exp.*, 2009, **39**, (6), pp. 573–610
14 Frost, A., Campo, M.: 'Advancing defect containment to quantitative defect management', *J. Def. Softw. Eng.*, 2007, **20**, (12), pp. 24–28
15 Rosemberg, D., Stewphens, M.: 'Use case driven object modelling with UML: theory and practice' (Apress, Berkeley, 2007)
16 Cox, K., Phalp, K.T.: 'Practical experience of eliciting classes from use case descriptions', *J. Syst. Softw.*, 2010, **80**, (8), pp. 1286–1304

© The Institution of Engineering and Technology 2012
IET Softw., 2012, Vol. 6, Iss. 6, pp. 559–573
doi: 10.1049/iet-sen.2011.0060

17 Constantine, L., Lockwood, L.: 'Structure and style in use cases for user interface design'. Second Int. Conf. Usage-Centred Design, New Hampshire, 2003
18 Larman, C.: 'Applying UML and patterns – an introduction to object-oriented analysis and design and the unified process' (Prentice-Hall, 1998, 2nd edn.)
19 Ochodek, M., Nawrocki, J.: 'Automatic transactions identification in use cases'. Balancing Agility and Formalism in Software Engineering: Second IFIP Central and East European Conf. Software Engineering Techniques CEE-SET 2007, 2008, (*LNCS*, **5082**), pp. 55–68
20 Sinnig, D., Chalin, P., Rioux, F.: 'Use cases in practice: a survey'. Proc. CUSEC 05, Ottawa, Canada, 2005
21 http://www.office.microsoft.com/en-us/access/, accessed April 2012
22 Cunningham, C., Galindo-Legaria, C.A., Graefe, G.: 'PIVOT and UNPIVOT: optimization and execution strategies in an RDBMS'. Proc. Thirtieth Int. Conf. Very Large Data Bases, 2004, vol. 30, pp. 998–1009
23 Singh, R., Serviss, J.: 'Code generation using GEODE: a CASE study', in Cavalli, A., Sarma, A. (Eds.): 'SDL'97: time for testing', (Elsevier, 1997), pp. 539–550
24 Cleaveland, J.C.: 'Building application generators', *IEEE Softw.*, 1988, **5**, (4), pp. 25–33
25 Nakatani, T., Urai, T., Ohmura, S., Tamai, T.: 'A requirements description metamodel for use cases'. Eighth Asia-Pacific Software Engineering Conf. (APSEC'01), 2001
26 Urai, T., Ohmura, S., Nakatani, T.: 'Use case oriented requirements analysis technique and its support environment'. Proc. Object-Oriented 2001 Symp., Kindai-Kagaku-Sha, 2001, (in Japanese), pp. 17–24
27 Fowler, M.: 'Patterns of enterprise application architecture' (Addison-Wesley Professional, 2002, 1st edn.)
28 http://www.incubator.apache.org/isis, accessed April 2012
29 http://www.metawidget.com, accessed April 2012
30 http://www.webratio.com, accessed April 2012
31 http://www.alphasimple.com, accessed April 2012
32 http://www.bizagi.com, accessed April 2012
33 Buzejic, S.: 'Komparativna analiza alata za generisanje Java korisničkog interfejsa'. MSc thesis, Faculty of Organizational Sciences, University of Belgrade, 2011 (in Serbian)
34 Cirovic, I.: 'Komparativna analiza CASE alata za generisanje aplikacija na osnovu modela'. MSc thesis, Faculty of Organizational Sciences, University of Belgrade, 2011 (in Serbian)
35 Forward, A., Badreddin, O., Lethbridge, C.T., Solano, J.: 'Model-driven rapid prototyping with Umple', *Softw. Pract. Exp.*, 2012, **42**, (7), pp. 781–797

© The Institution of Engineering and Technology 2012