UML Profile for Specifying User Interfaces of Business Applications

Gordana Milosavljević, Igor Dejanović, Branko Perišić, and
Branko Milosavljević

Faculty of Technical Sciences, University of Novi Sad, Serbia
{grist,igord,perisic,mbranko}@uns.ac.rs

Abstract. This paper presents a domain-specific language (DSL) in the
form of a UML profile that facilitates efficient modeling and generation
of fully-functional UIs of enterprise business applications. DSL is based
on an internal HCI standard that defines layout and behaviour of coarse-
grained objects for business applications. Being a regular UML extension,
this language can be used in any general-purpose UML modelling tool
and can easily be integrated with other UML-based models of the appli-
cation.

Key words: user interface, code generation, MDA, UML profile

1   Introduction

Various aspects of model-based development of user interfaces (UIs) are the
subject of intensive research efforts. However, the majority of presented solutions
is hardly applicable to development of real-world information systems because
too much time and effort is spent on developing and synchronisation of different
types of user interface models (for example, presentation model, content model,
navigation model, interaction model), the complexity of sharing the knowledge
embedded in different models, the lack of support in development tools, and the
lack of consensus over which types of models best describe UIs [23]. Most tools
for modelling user interfaces use its own set of notations, thus impeding the
integration with other application models [26]. This is especially the problem in
developing business applications that require tight integration of UI models with
models that specify business logic.

In order to overcome the problem of integration and to facilitate the exchange
of information among different tools, UML can be used to model all aspects of
an application, including the user interface [3]. Although very powerful, UML
without extensions is not suitable for modelling UIs [26, 3, 19].

This paper presents a UML extension in the form of a UML profile for speci-
fying UIs of business applications named EUIS (*Enterprise User Interface Spec-
ification* profile). EUIS is developed in order to enable rapid user interface mod-
elling at a high level of abstraction. EUIS is based on an internal HCI (human-
computer interaction) standard of a business application that defines functional

and presentational features of coarse-grained building blocks thus enabling the generation of a fully functional UI, without the need for defining a multitude of models used for developing UIs in the general case.

It is important to note that the UI model *is not* a model of an application (from the implementation standpoint); it defines the structure of the application using building blocks at a high abstraction level (different types of screen forms, reports, procedures) and their relationships. Depending on the development platform, the intended application architecture, and the implementation of a code generator, one class from the UI model may be mapped to one or more classes or modules of an application, or may even be not mapped to the program code at all but to application repository data instead, if a data-driven application architecture is used (for example, see [13, 14]).
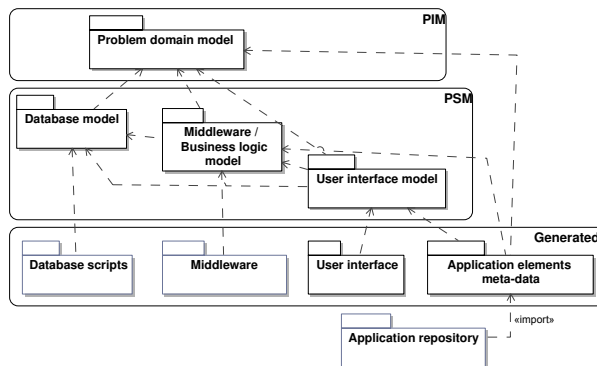
Fig. 1. Model transformations

The development of a whole business application using the EUIS profile comprises the following activities (see Figure 1):

- The development of PIM (platform independent model) of a problem domain by means of class diagrams in a general-purpose UML modelling tool.
- The automatic transformation of a PIM to PSMs (platform-specific model): database schema model, user interface model, and the middle-tier model (in the case a three-tier architecture is chosen).
- Automatic generation of artifacts needed for implementation based on PSMs: database schema creation or alteration scripts, middle-tier implementation artifacts (such as EJBs), fully functional application UI (depending on the target architecture of the client application), and atomic "CRUD" transactions implementing creation, retrieval, update, and deletion of every entity in the persistence layer.

The rest of the paper is structured as follows. Section 2 describes the basics of the internal HCI standard. Section 3 presents the EUIS profile. Section 4 reviews the related work. The last section concludes the paper and outlines some directions of future work.

2   The HCI standard

Our internal human-computer interaction (HCI) standard is aimed at defining functional and visual features of course-grained application components. Its goals include the following: simplicity of use, quick user training, and the automation of user interface construction.

The papers [16, 17] define a number of types of screen forms. For this discussion, the following types are relevant:

- standard data management form,
- standard panel,
- parent-child form, and
- many-to-many form.

Standard form is designed to display data and all available operations so the user can choose a data item and invoke an operation on it without memorising commands (the object-action approach [24]). Standard operations common to all entities are represented by buttons/icons at the top of the form, while specific operations (if they exist) are represented by links/buttons at the right hand side. The standard form layout is presented in Figure 2.
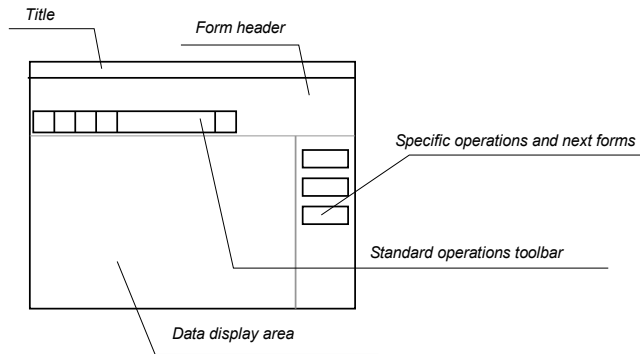
Fig. 2. Standard form layout

Operations common to all entities include search (query by form), display, addition, update, removal, copying, data navigation and view mode toggle (grid view or single record view). Specific operations include complex data processing procedures associated with the given entity (transactions), invocation of related (*next*) screen forms, and invocation of reports. The standard mandates that the specific operations always use the currently selected (viewed) record.

Standard panel has the appearance and the behaviour of the standard form but, instead being shown in its own window, it is used as an element of a complex form. Standard panels are regularly used for parent-child and many-to-many forms.

A **parent-child form** is used for data that have hierarchical structure, where each element in the hierarchy is modelled as an entity in the database. Each element in the hierarchy is represented by a standard panel, where a panel at

the n-th hierarchy level filters its content according to the selected data item at the level $n - 1$.

The **many-to-many form** is used for intensive management of data belonging to entities connected by "many-to-many" relationships, with or without associate classes. Its layout is presented in Figure 3. This screen form is used as follows:

– A number of desired records are selected in the upper panel. These records are "dragged" to the lower panel by clicking the button with the downwards arrow. If a record is dragged by mistake, it can be revoked back by clicking the upwards arrow button.
– The values of non-key attributes of a record selected in the lower panel may be changed.
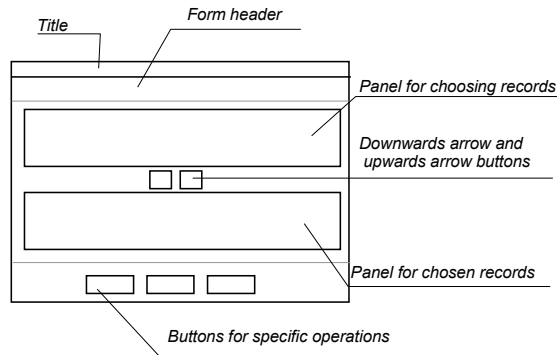
Fig. 3. Many-to-many form layout

Relationships among screen forms are represented by three mechanisms: *zoom*, *next*, and *activate*. The **zoom** mechanism represents the invocation of the form associated with the given entity where the user can choose a data item and "drag" it (pick its values) to the fields of the previously viewed form.

The **next** mechanism, invoked from the form associated with the current entity, displays the form associated with the child entity with its data filtered so that only connected objects are displayed. The key or a representation of the parent entity is displayed in the form header, so the user easily recognises the current context. A *next* can be invoked by menu items, buttons, or links.

The **activate** mechanism enables direct invocation of a form by another form, without restrictions on the data displayed. The invoked form does not need to be related to the current one.

3   The EUIS Profile

The EUIS profile extends the following metaclasses from the *UML::Kernel* package: *Element*, *Class*, *Property*, *Operation*, *Parameter*, *Constraint*, and *Package*. It is complementary to the profile for modelling persistent data that is available

in a majority of modelling tools (see Figure 4). Therefore, EUIS is independent on modelling tools, persistence layer and the database of choice. Profile that models persistent data comprises only the stereotypes present in the majority of modelling tools (possibly under a different name): persistent class, persistent property, persistent data type, and persistent operations (methods implemented in the persistence layer) – see Figure 5. When using the EUIS profile, these stereotypes are replaced with concrete stereotypes of the chosen modelling tool.
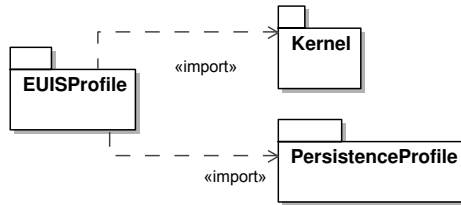

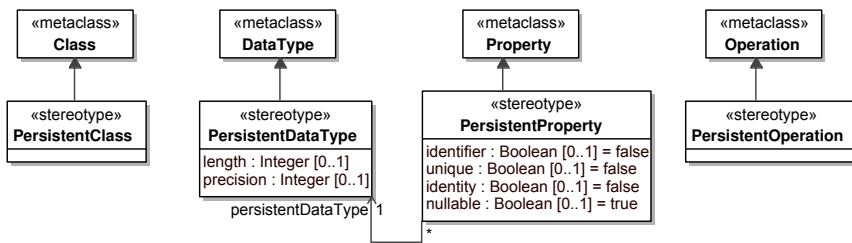**Fig. 4.** Profile structure

Fig. 5. Persistence profile

In order to specify additional information needed for transforming a problem domain model to a user interface model, another profile is developed (see Figure 6) that provides the following: defining a set of one or more properties as a business key – the *BusinessKey* stereotype [4], designation of a method as a complex business procedure – the *Transaction* stereotype, and the designation of a method as a report – the *Report* stereotype.
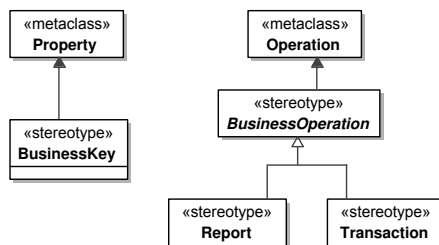

Fig. 6. Profile used in the problem domain model

Stereotypes and enumerated types of the EUIS profile are organised in the following categories:

− a visible element: extension of *Element* metaclass

- visible classes (panels): extensions of *Class* metaclass
- visible properties: extensions of *Property* metaclass
- visible methods: extensions of *Operation* metaclass
- visible parameter: extension of *Parameter* metaclass
- a group of elements: extension of *Property* metaclass
- visible association ends: extensions of *Property* metaclass
- validators: extension of *Constraint* metaclass
- a business subsystem: extension of *Package* metaclass

Due to space constraints, the rest of the section presents only the most important stereotypes and tags. Validators, business subsystem and formal OCL constraints are not presented.

3.1   Visible Elements

Stereotype *VisibleElement* (see Figure 7) represents a model element that is mapped to a user interface element in the generated application. Since *Element* metaclass is a common superclass of all UML metaclasses, this facilitates the representation of all model elements with a UI component and a label, where applicable.
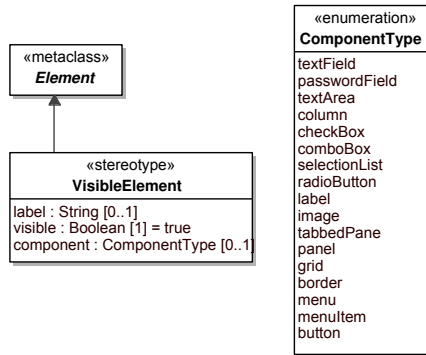
Fig. 7. Visible element

The enumerated type *ComponentType* defines a set of available UI component types. The set of components is designed to be platform-independent. Mapping these values to particular UI components of the chosen development platform is performed in the application generator.

3.2   Visible Classes

Stereotype *VisibleClass* (see Figure 8) represents a class that is mapped to a panel (a regular or a tabulated panel) in the application UI. If a panel is associated to an empty window or a web page, it becomes a screen form that can be independently activated (opened).
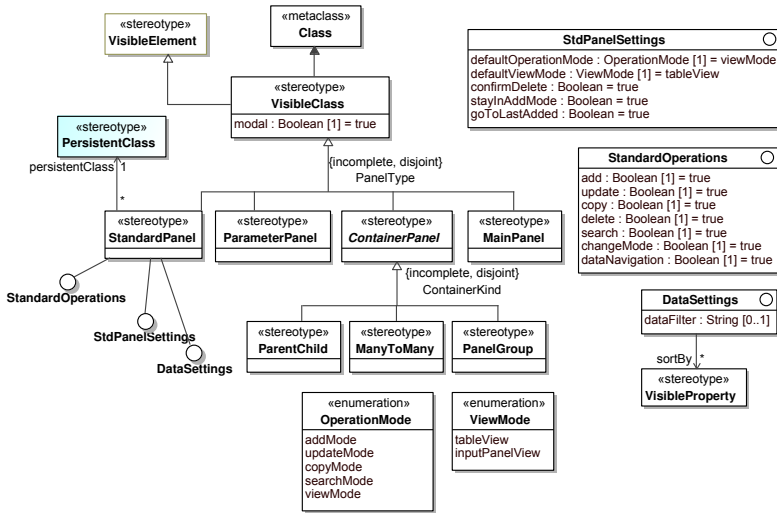
Fig. 8. Visible classes

Stereotype *VisibleClass* is not abstract because of the possibility of modelling specific panels that are not comprised by the HCI standard, but still occur rarely enough that there is no need to extend the HCI standard with a new element.

The inherited tag *label* is used as a window title or a label that explains the purpose of the panel if displayed within a complex panel.

Stereotype *StandardPanel* denotes that the given persistent class is associated with a standard panel whose layout and behaviour are defined by the internal HCI standard. The standard panel implements three interfaces: *StandardOperations* – operations defined by the HCI standard: add, update, copy, delete, search, change mode, navigate data; *StdPanelSettings* – settings that define panel's runtime behaviour; and *DataSettings* – defines data filtering and sorting.

Stereotype *ParameterPanel* represents a class that is mapped to a panel for entering parameters for a visible method (see *VisibleOperation* stereotype) that is invoked by a button or a menu item. Since the majority of parameter panels in an application is created implicitly, as a result of a visible method and its parameters, classes with this stereotype rarely occur. It can be used in situations where a user successively invokes a number of methods with the same set of parameter values.

The *ContainerPanel* is an abstract stereotype that represents a complex panel that can contain other panels (simple or complex), as well as a number of properties and methods. It defines additional attributes, methods, and constraints for its descendants (*ParentChild*, *ManyToMany*, and *PanelGroup*). The layout and behaviour of *ParentChild* and *ManyToMany* panels is defined by the HCI standard, while their relationship to the contained panels is defined by hierarchical relationships (associations with ends having the *Hierarchy* stereotype). For details on associating panels, see section 3.7.

The layout and behaviour of a *PanelGroup* is not defined by the HCI standard. It is used for modelling special-purpose complex panels. The class with a *PanelGroup* stereotype defines only the contained elements, while their relationship is implemented in application code.

Classes with the *MainPanel* stereotype are used for modelling the main form of a business subsystem
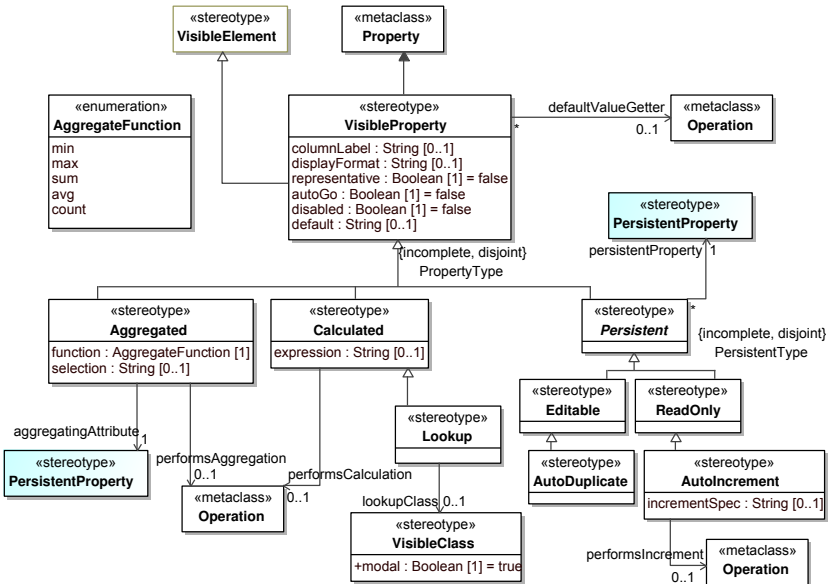
3.3   Visible Properties

Fig. 9. Visible properties

Stereotype *VisibleProperty* (see Figure 9) is a property of a "visible" class and is mapped to a UI component contained in the panel associated to the class. Its tags provide customisation of appearance and behaviour of the UI component, or the table column in the case of tabular display of data (*label*, *columnLabel*, *dataFormat*, *disabled*), default values in the UI component (*default*, *defaultValueGetter*), and automatic focus traversal (*autoGo*). Tag *default* contains an OCL expression that defines the initial value, while *defaultValueGetter* contains the reference to the method used for fetching the default value (in cases when OCL expression cannot be used). Tag *representative* indicates that the given property can be used to represent the whole class from the users' point of view (for example, company name, first name + " " + last name).

Aggregated represents an aggregated property, whose value is calculated using one of the aggregation functions (min, max, sum, avg, count) over the selected property (*aggregatingAttribute*). The set of values being aggregated is specified by an OCL expression (*selection*) or by a method (*performsSelection*).

Stereotype *Calculated* represents a property whose value is calculated according to the given formula over the values in objects of this or some other class. Calculation method can be specified by an OCL expression (*expression*) or by a method (*performsCalculation*).

Abstract stereotype *Persistent* represents a property that is mapped to a persistent property in the problem domain model. Its descendants include *Editable* (enables editing the value of the persistent property in the UI component) and *ReadOnly* (disables editing). Editing values is allowed if the user has appropriate permissions.

Editable has an *AutoDuplicate* descendant that represents a persistent property where the value entered in the UI component is kept as default when entering a new record. It is usually applied to properties whose values are repeated across many records, so the user is spared some effort while entering data.

ReadOnly has an *AutoIncrement* descendant that denotes a persistent property whose value is automatically incremented with each new record entered. Contrary to identity columns or database sequences, this property allows the counter value to be reset if a condition is met (using an OCL expression in *incrementSpec* or a method in *performsIncrement*).

Stereotype *Lookup* describes a property whose value is formed from property values of referenced objects, directly or indirectly. Direct reference means that there is an association with the class that provides the data; indirect reference means that such class can be reached by traversing a series of associations. Properties forming a lookup can be specified as an OCL expression (*expression*) or by specifying the class that provides the data. In the latter case, the representative property of that class is used.
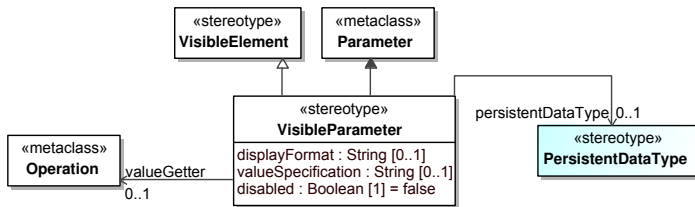
3.4 Visible Parameters

Fig. 10. Visible parameters

Stereotype *VisibleParameter* (see Figure 10) denotes a parameter of a visible method (having the *VisibleOperation* stereotype) that behaves as follows. If it is an input or an input/output parameter, then

- it enables entering parameter values by means of a UI component contained in the parameter panel associated with a visible method, or
- it defines the way of fetching the parameter values in the case when the user is not supposed to enter its value (using tag *valueSpec* contains an OCL expression that calculates the value, or tag *valueGetter* that specifies the method for calculating the value).

If it is an output parameter or a method result, it enables the display of its value by means of a UI component contained in the parameter panel associated with a visible method.
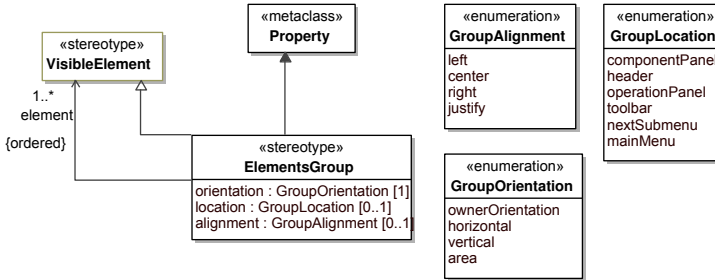
3.5   Groups of Elements

Fig. 11. Groups of elements

Stereotype *ElementsGroup* (see Figure 11) represents an attribute of a class with the *VisibleClass* stereotype used for grouping its elements (properties, methods, associations), thus forming semantic groups that map to groups of UI components in a panel associated with the class. Each group can define the following: an ordered collection of contained elements (tag *element*), the UI element orientation in layout (*orientation*), the location of the group in the panel (*location*), and the alignment of elements in the group (*alignment*).

The inherited tag *label* represents a label displayed in a UI component associated with the group (frame title, panel title, name of the menu item that opens a submenu).

3.6   Visible Methods

Stereotype *VisibleOperation* (see Figure 12) denotes the method of a visible class that has an associated UI component (a button or a menu item) that enables its invocation by the user. If the method has input parameters, they must have the *VisibleParameter* stereotype (see section 3.4).

Abstract stereotype *BusinessOperation* represents a method that is mapped to an activity in the problem domain. Its descendants are *Report* and *Transaction*. *Report* describes a method that invokes a report created by one of the reporting tools. *Report*'s tags enable specifying the report name, and the filtering and sorting criteria. *Transaction* represents a complex business transaction that is implemented as a stored procedure in the database or a method in a middle tier. Its tags enable specifying the UI behaviour immediately before and after its invocation (requesting the confirmation from the user, display refresh mode, error display mode, etc).
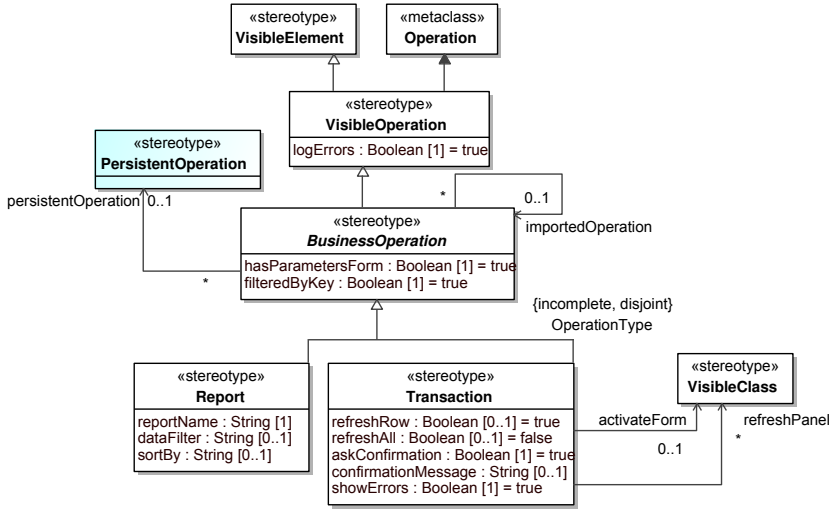
Fig. 12. Visible methods

3.7   Visible Association Ends

Abstract stereotype *VisibleAssociationEnd* (see Figure 13) is applied to a property belonging to a binary association between two visible classes. It defines the relationship between the panel belonging to the class that owns the property (activation panel) and the panel belonging to the class at the other end (destination panel). The nature of the relationship is determined by this stereotype's descendants. *VisibleAssociationEnd* only introduces common properties and constraints that enable destination panel to adjust its layout and behaviour to the context it is used in. For this purpose, *VisibleAssociationEnd* implements the following interfaces: *StandardOperations*, *DataSettings*, *StdPanelSettings*, and *PanelAdjustment*. Tag values specified by *PanelAdjustment* can be set for all types of panels (*VisibleClass* and its descendants), while tag values specified by *StandardOperations*, *DataSettings*, and *StdPanelSettings* can be applied to standard panels only (stereotype *StandardPanel*, see section 3.2).

If tag values are not defined at the association end, values defined at the standard panel are used. If values of tags *add*, *update*, *copy*, *delete*, *search*, and *changeMode* are set to false in the standard panel, the value set at the association end is ignored. This helps adhering to rules that are usually consequences of problem domain constraints independent of the usage context.

Stereotypes *Zoom*, *Next*, and *Activation* model the corresponding type of activation as defined by the HCI standard. Stereotype *Hierarchy* denotes that the destination panel has the role of an element in the parent-child or many-to-many panel. Role of the destination panel is set by the value of the *level* tag. For many-to-many complex panels, *level* = 1 is the panel that represents the header, *level* = 2 is the panel for choosing data, and *level* = 3 is the panel that contains the transferred data (for example, see class *PickAuthors* in Figure 15). For parent-child complex panels, *level* = 1 is the standard panel being the
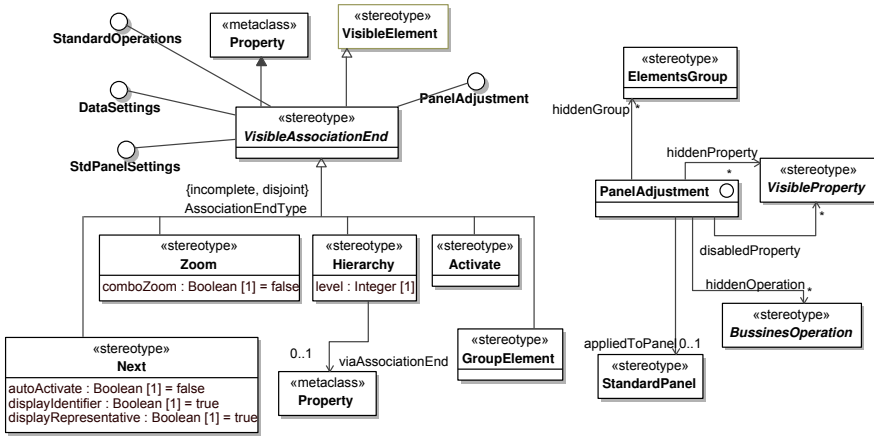
Fig. 13. Visible association ends

root of the tree, $level = 2$ is the child panel, $level = 3$ is the child of the child panel, and so forth: for $n > 2$, $level = n$ is a panel that is the child for panel at $level = n - 1$ (for example, see class *JournalPaperComposite* in Figure 15).

Composing parent-child and many-to-many complex panels requires defining only levels of hierarchy for each contained panel; runtime association of panels is performed by analysing their associations. If two or more associations exist between two panels, or there is a recursive association, association end to be used must be explicitly stated in the *viaAssociationEnd* tag.

Stereotype *GroupElement* denotes that the destination panel is an element of a complex panel, where its role and behaviour are defined in the application code and/or using values of tags inherited from *VisibleAssociationEnd*.

3.8 Example

Figure 14 presents a domain model of a part of a CERIF-compliant research management system presented in [17]. All classes and attributes in this model are persistent, but their stereotypes are not displayed for the sake of brevity.

The problem domain model in Figure 14 is automatically transformed into the UI model presented in Figure 15 that contains an initial version of the user interface. The application developer can manually change this version to meet the users' requirements. The diagram in Figure 15 shows manually added classes *PickAuthors* (a many-to-many form for choosing paper authors) and *Journal-PaperComposite* (a parent-child form for managing journal papers) with corresponding associations. Properties with *Lookup* and *ElementsGroup* stereotypes in all classes are also manually added. The rest of the model is automatically generated: persistent classes from the domain model was mapped to UI classes with *StandardPanel* stereotype, persistent properties to UI properties with *Editable* stereotype, association ends with cardinality 0..* to UI association ends with *Next* stereotype and association ends with cardinality 0..1 or 1 to UI association ends with *Zoom* stereotype.
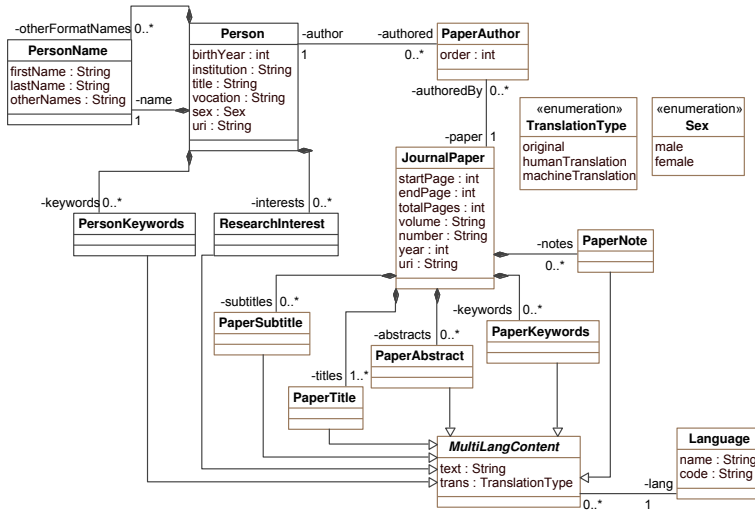
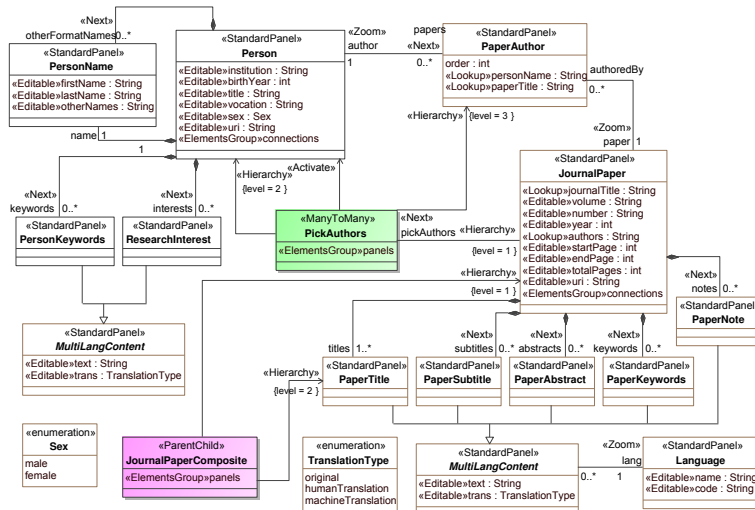**Fig. 14.** A domain model of a part of CERIF-compatible system

Fig. 15. A UI model of a part of CERIF-compatible system

An example of a web-based form generated for *JournalPaperComposite* is presented in Figure 16.

Fig. 16. Managing journal papers data

4   Related Work

In order to compare the EUIS profile with other profiles presented in the literature, this section reviews recent papers ranging in subject from modelling user interfaces of business applications to complete methodologies and tools for information system development, including its presentation aspects. Papers that deal with developing user interfaces in general are not discussed here.

The papers [21, 22], like this paper, propose the use of the problem domain model as a starting point that is transformed into a model of the user interface. The approach to modelling of application views based on the available classes (various complex panels, navigation among them) is also similar. However, [21, 22] introduces more transformation levels in order to achieve portability across different implementation platforms while not restricting to a particular fixed set of components (we deal with portability as well, but with a limited set of components). Besides, [21, 22] require the development of an information retrieval model in order to implement fetching of data used in the user interface, while we opt for implicit mapping of the user interface model to the persistence layer (the cases where implicit mapping is insufficient are defined by OCL constraints or an associated method).

The papers [10, 5] present a way of automatic user interface generation based on the following: a business logic model (UML activity diagrams) and a user interface model (UML class diagrams). An activity diagram is supplied with elements of the UML profile for defining system and user actions with the specified inputs and outputs, while the class diagram that is produced from the

activity diagram is supplied with elements of the UML profile for user interface specification (e.g., *ContainerElement*, *GuiElement*, *ActionElement*). The profile does not support modelling the relationships between forms (navigation is omitted). Furthermore, obtaining classes that provide management of data from the problem domain model is not specified, although their presence is assumed (the *dataProvider* attribute in the *ChoiceElement* stereotype, and the *methodURL* attribute in the *ActionElement* stereotype).

In [23], the authors propose the use of patterns for accelerating the user interface development. Those patterns are at a higher abstraction level and may be used in task models, presentation models, and component layout models. This paper also proposes the use of a number of tools that guide the designer in the choice and the application of patterns during modelling, assist in model synchronisation, and generate the user interface.

Compared to elements of the standard presented in Section 2, the patterns used in [23] represent finer-grained application components. The paper [23] does not specify neither the relationship between the problem domain model and the user interface model, nor if there is a mapping of the generated user interface to the data persistence layer (whether the generated user interface is immediately testable in the real users' environment).

The paper [27] presents a method for developing web-based information systems based on problem domain models, applications and navigations that are directly mapped to existing development frameworks. The mapping is provided by the UML profile named *FrameWeb* whose stereotypes correspond with the categories of the framework used, so that the development team can deliver the implementation in a straightforward way (the implementation is manual, there are no code generators used). The majority of stereotypes in the presented UML profile are aimed at the development of the problem domain model and its mapping to the persistence layer, while support for specifying the user interface is relatively modest (there are only four stereotypes that specify the type of the web page).

The series of papers [2, 8, 7, 18] presents a methodology for developing web-based information systems UWE (*UML-based Web Engineering*) that uses a UML profile for modelling hypermedia and the ArgoUWE tool that provides for definition of different application models, their transformation, and semi-automatic code generation. The UML profile provides for the creation of navigation models, navigation structure models, and presentation models.

The concept of modelling an application in UWE methodology is the closest to the proposition in this paper – in both cases, the starting point is the problem domain model expressed as a class diagram that is automatically mapped to the application model, data model, and other models needed. Thanks to this approach, there is a direct mapping of application elements to the layer that implements business logic, a feature missing in the majority of reviewed solutions. The most notable differences between UWE and EUIS approaches are the following:

- The UWE methodology and profile are focused solely on developing web-based systems, while the methods presented here can be applied to both web and "classical" information systems.
- The UWE method does not rely on an internal HCI standard (there is only one type of forms).
- Our approach proposes a single user interface model that defines coarse-grained application building elements, their structure and layout (using the *ElementsGroup* stereotype), and navigation among them. Sketches of forms need not be made thanks to the mechanism for intelligent component layout that forms a usable user interface according to rules and groups, and which can be further adapted during implementation.

Although not based on a UML profile, the concept of specifying GUI forms and generating the database schema and the functional prototype of the application using the IIS*Case tool [20, 11, 6] is similar to the solution presented here, apart from the order in which artifacts are implemented. Using IIS*Case, the modelling starts with specifying form types, while database schema and the prototype application are generated. Here we start with the model of the problem domain, that is used to generate the user interface model, database schema model, and the middle-tier model (in the case of three-tier architectures). After manual changes applied to these automatically obtained models, the application is generated.

Our previously implemented tools for generating UIs of business applications for various platforms are presented in [12–16, 9]. All tools are based on the internal HCI standard presented in Section 2, but the difference is that UI model was not generated from the domain model, but was kept as metadata in the application repository. Metadata was further customised by the Form Generator tool, which utilised this information to generate source code. Metadata in the application repository, although stored in the database or an XML file and edited by a special-purpose tool, can be considered to be a DSL (domain specific language) for the description of UIs. The UI model enriched with EUIS stereotypes is based on the same metadata, but this UML-based form is more suitable for team work of experts from different fields (developers, UI design specialists, problem domain specialists, users) during application development.

5   Conclusions

Automatic generation of UIs in the general case requires the development of a number of UI models and thus requires much time and effort, often with unsatisfactory results. Synchronisation and integration among different UI models, as well as with domain models, is another big problem.

If we restrict to the problem of developing special-purpose UIs, these problems are largely simplified. This paper presents UML profile that facilitates efficient modeling and generation of fully-functional UIs of enterprise business applications which is based on an internal HCI standard that defines layout and

behaviour of coarse-grained objects for business applications. Relying on this standard has enabled the rapid development of UIs for this particular type of applications. Being a regular UML extension, this language can be used in any general-purpose UML modelling tool and can easily be integrated with other UML-based models of the application.

Our tools developed to support the presented concepts [12–16, 9] are used for the implementation of more than 70 projects of business information systems by several different development teams. The percentage of the generated code in the overall code base (database, middle tier, UI) ranged from 81.8% to 98.2%, depending on the type of application. The code generation tool that relies on the presented EUIS profile is still in development, but initial results show that the percentage of the generated code will increase when all elements are implemented. The current version does not support parsing OCL constraints. Since we have already implemented a dynamic general-purpose parser Arpeggio [1], the support for OCL expressions is soon to be finalised.

References

1. Arpeggio Parser, `http://code.google.com/p/arpeggio/`
2. Baumeister, H., Koch, N., Mandel, L.: Towards a UML Extension for Hypermedia Design, In: Proceedings of The Unified Modelling Language Conference: Beyond the Standard (UML 1999), France R. and Rumpe B., Eds, LNCS vol. 1723, pp. 614–629, Springer Heidelberg (1999)
3. van den Bergh, J., Coninx, K.: Using UML 2.0 and Profiles for Modelling ContextSensitive User Interfaces, In: Model Driven Development of Advanced User Interfaces, Montego Bay, Jamaica (2005)
4. Dejanović, I., Milosavljević, G., Perišić, B., Tumbas, M.: A Domain-Specific Language for Defining Static Structure of Database Applications, Computer Science and Information Systems 7(3), (June 2010), pp. 409 - 440
5. Funk, M., Hoyer, P., Link, S.: Model-driven Instrumentation of Graphical User Interfaces, In: Second International Conference on Advances in Computer-Human Interaction, Cancun, Mexico (2009)
6. Govedarica, M., Luković, I., Mogin, P.: Generating XML Based Specifications of Information Systems, Computer Science And Information Systems 1(1), pp. 117–140 (2004)
7. Knapp, A., Koch, N., Zhang, G.: Modelling the Structure of Web Applications with ArgoUWE, LNCS vol. 3140, Springer Heidelberg (2004)
8. Koch, N., Kraus, A.: The Expressive Power of UML-based Web Engineering, In: Proc. 2nd International Workshop on Web Oriented Software Technology, pp. 105–119 (2002)
9. Komazec, S., Milosavljević, B., Konjović, Z.: XML Schema-Driven GUI Forms Environment, In: 11th IASTED Intl. Conf. Software Engineering and Applications, pp. 342–348, Cambridge, MA (2007)
10. Link, S., Schuster, T., Hoyer, P., Abeck, S.: Focusing Graphical User Interfaces in Model-Driven Software Development, In: First International Conference on Advances in Computer-Human Interaction, Saint Luce, Martinique (2008)
11. Luković, I., Mogin, P., Pavievi, J., Risti, S.: An Approach to Developing Complex Database Schemas Using Form Types, Software: Practice and Experience 37(15), pp. 1621-1656 (2007)

12. Milosavljević, B., Vidaković, M., Milosavljević, G.: Automatic Code Generation for Database-Oriented Web Applications, In: Power, J., Waldron, J. (eds): Recent Advances in Java Technology: Theory, Application, Implementation. pp. 89–97, Trinity College Dublin (2003) ISBN 0954414500
13. Milosavljević, B., Vidaković, M., Komazec, S., Milosavljević, G.: User Interface Code Generation for EJB-Based Data Models Using Intermediate Form Representations, In: Principles and Practice of Programming in Java, pp. 125–132, Kilkenny, Ireland (2003)
14. Milosavljević, B., Vidaković, M., Komazec, S., Milosavljević, G.: User Interface Code Generation for Data-Intensive Applications with EJB-Based Data Models, In: Software Engineering Research and Practice (SERP'03), pp. 23–27, Las Vegas, NV (2003)
15. Milosavljević, G., Perišić, B.: Really Rapid Prototyping of Large-Scale Business Information Systems, In: IEEE Intl. Workshop on Rapid System Prototyping, pp. 100–106, San Diego, CA (2003)
16. Milosavljević, G., Perišić, B.: A Method and a Tool for Rapid Prototyping of Large-Scale Business Information Systems, Computer Science And Information Systems 2(1), pp. 57–82 (2004)
17. Milosavljević, G., Ivanović, D., Surla, D., Milosavljević, B.: Automated Construction of the User Interface for a CERIF-Compliant Research Management System, The Electronic Library (in print)
18. Moreno, N., Melia, S., Koch, N., Vallecillo, A.: Addresing New Concerns in Model-Driven Web Engineering Approaches, In: Proc. Web Information Systems Engineering (WISE), LNCS vol. 5175, pp. 426–442, Springer Heidelberg (2008).
19. Paterno, F.: Towards a UML for Interactive Systems, In: Proc. Engineering for Human-Computer Interaction, pp. 7–18, Toronto, Canada, (2001)
20. Pavićević, J., Luković, I., Mogin, P., Govedarica, M.: Information System Design And Prototyping Using Form Types, In: International Conference on Software and Data Technologies, pp.157–160, Setubal, Portugal (2006)
21. Schattkowsky, T., Lohmann, M.: Towards Employing UML Model Mappings for Platform Independent User Interface Design, In: Model Driven Development of Advanced User Interfaces, Montego Bay, Jamaica (2005)
22. Schattkowsky, T., Lohmann, M., UML Model Mappings for Platform Independent User Interface Design, In: MoDELS 2005 Workshops, LNCS 3844, pp. 201-209, Springer, Heidelberg (2006)
23. Seffah, A., Gaffar, A.: Model-Based User Interface Engineering with Design Patterns, Journal of Systems and Software 80(8), pp. 1408–1422 (2007)
24. Shneiderman, B.: Designing the User Interface: Strategies for Effective Human-Computer Interaction, Addison-Wesley, Third Edition (1998)
25. da Silva, P.P.: User Interface Declarative Models and Development Environments: A Survey, In: Proc. Design, Specification and Verification of Interactive Systems, LNCS vol. 1946, pp. 207–226, Limerick, Ireland (2000)
26. da Silva, P.P., Paton, N.W.: Improving UML Support for User Interface Design: A Metric Assessment of UMLi, In: Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction at International Conference on Software Engineering (ICSE 03), pp. 76–83, Portland, Oregon, USA (2003)
27. Estêvão Silva Souza, V., Almeida Falbo, R., Guizzardi, G.: A UML Profile for Modelling Framework-based Web Information Systems, In: Workshop on Exploring Modelling Methods for Systems Analysis and Design (EMMSAD'07), pp. 149–158, (2007)