e-Informatica Software Engineering Journal, Volume 8, Issue 1, 2014, pages: 39–52, DOI 10.5277/e-Inf140103

Generating Graphical User Interfaces from Precise Domain Specifications

Kamil Rybiński*, Norbert Jarzębowski*, Michał Śmiałek*, Wiktor Nowakowski*, Lucyna Skrzypek*, Piotr Łabęcki*

**Faculty of Electrical Engineering, Institute of Theory of Electrical Engineering, Measurement and Information Systems, Warsaw University of Technology*

rybinskk@iem.pw.edu.pl, jarzebon@iem.pw.edu.pl, smialek@iem.pw.edu.pl, nowakoww@iem.pw.edu.pl, skrzypel@ee.pw.edu.pl, labeckip@ee.pw.edu.pl

Abstract

Turning requirements into working systems is the essence of software engineering. This paper proposes automation of one of the aspects of this vast problem: generating user interfaces directly from requirements models. It presents syntax and semantics of a comprehensible yet precise domain specification language. For this language, the paper presents the process of generating code for the user interface elements. This includes model transformation procedures to generate window initiation code and event handlers associated with these windows. The process is illustrated with an example based on an actual system developed using the presented approach.

1. Introduction

Requirements Engineering (RE) is a very distinct area of Software Engineering, because requirements define the problem space while other software artifacts operate in the solution space. Problems with requirements usually get amplified in later stages of software development leading to project failures [1]. This makes RE research especially important and challenging. When defining research directions for RE [2], we need to bear in mind that RE starts with ill-defined and often conflicting ideas and have to be handled by very varied groups: from domain experts and end-users to downstream developers. Challenges in this broad research field include finding ways to effectively elicit and formulate requirements and then turn them into other SE artifacts (design, code, tests, etc.).

A very promising approach to meet the RE challenges is Model-Driven Requirements Engineering (MDRE) [3]. MDRE is an emerging area of Model Driven Software Development (MDSD) [4, 5]. The basis for constructing an MDRE approach is a model-based language for expressing requirements. Probably the first such language is the Requirements Modeling Language proposed by Greespan et al [6, 7]. More recent languages include the Requirements Specification Language [8] and the Unified Requirements Modeling Language [9].

Building on the success of MDSD for design and implementation, Requirements Engineering can benefit from its techniques when properly balancing flexibility for capturing varied user needs with formal rigidity required for model transformations [10]. MDRE makes it possible that the requirements models define the real scope and all details of the envisioned software system, furthermore that the whole development [11, 12], testing [13] and documentation process will be driven and controlled by these requirements models as well.

ReDSeeDS [14] is a tool representing the MDRE approach by offering an open framework consisting of a scenario-driven development method and domain vocabulary management. It implements the Requirements Specification Language (RSL) [8, 15] meta-model which uses constrained natural language sentences allowing the end-users to understand specifications presented as precise requirements models. Moreover, the precisely written platform-independent specification allows to translate it directly to code using one of the platform-specific transformations. The latest ReDSeeDS transformations generate not only the entire code of the application logic layer, and the method stubs for the model layer, but also a fully functional graphical user interface. This paper concentrates on this last topic. It presents an approach to generate fully functional code of the UI elements from precisely specified domain models, expressed in RSL.

The solution separates essential complexity connected with the domain description such as business rules and application logic, from the accidental (technological) complexity related with platform specific design and implementation [16]. The complexity of software development process using ReDSeeDS is significantly reduced from the user and developer point of view. Most of the accidental complexity is hidden within a special model transformation program, used to convert requirements specifications into code.

2. Related Work

Transition from requirements to design or implementation is considered as a difficult activity during software development. The complexity related to it can cause various errors mainly caused by ambiguity of requirements. To eliminate this ambiguity, some form of constrained language could be used. This would allow for providing semi-automated ways to generate analysis and design models or code artifacts. There exist various approaches to solve this problem.

Some work has focused on requirements in respect to their precision, both by defining new languages for this purpose [17], as well as properly using the existing ones [18]. The disadvantage of these approaches is that they do not propose further code generation. Some approaches can be distinguished by their use of use case scenarios for requirements specification. Giganto et al. [19] propose an algorithm to identify use case sentences from requirements specifications written in controlled natural language and as a result – automatically obtain classes from use cases. Whereas Mustafiz et al. [20] propose transformation rules for creating different types of behavioral diagrams from use case scenarios. Deeptimahanti et al. [21] suggest to analyze requirements specifications presented in natural language, using Natural Language Processing techniques and generate use case diagrams and class models.

Most of the solutions focused on code generation use graphical notations like UML [22] to specify static and dynamic aspects of systems. One example is the open source AndroMDA [23] code generation framework which supports the Model Driven Architecture [24] paradigm. As input, AndroMDA takes UML models from various CASE tools and provides generation of deployable applications and software components. In turn, textual specifications used as input, turn out to be often too formalized and thus difficult to understand by the end-users where the purpose is specifying requirements [25].

There are also number of solution focusing directly on graphical user interface generation. Many of them, however, use notations designed specifically for this purpose e.g. the one proposed by Falb et al. [26]. Some other solution use the existing software development notations. However, these notations operate at significantly lower level of abstraction than requirements specification, as e.g. propopsed by Janssen et al. [27]. There also exist a solution based on requirements scenarios [28], but it only generates graphical user interface mockups.

3. Syntax for Domain Elements in RSL

RSL is based on scenarios consisting of sentences that describe interactions between the actors and the system, written in constrained natural language. Scenarios are also grouped into RSL-specific “Use Cases”, which are similar to

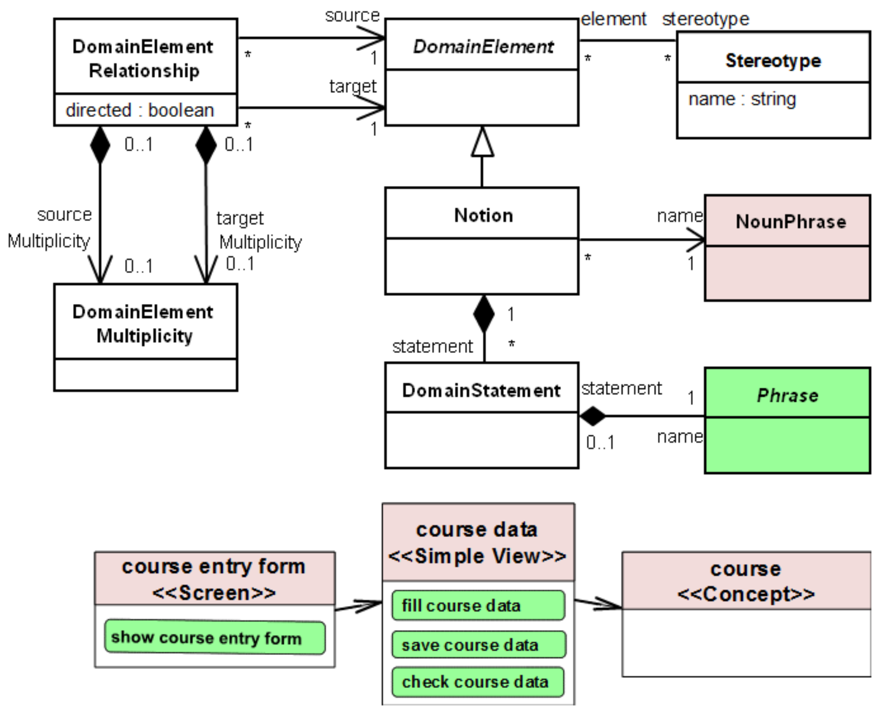


Figure 1. Abstract and concrete syntax for domain elements

the widely-known UML use cases. Additionally, every specification in RSL contains a domain model created from the notions used in use case scenarios. Each noun phrase in a scenario sentence should have a corresponding domain element.

The RSL's domain models are based on the metamodel, where its simplified version is presented in Figure 1 (upper part). The high-level elements in the RSL's metamodel can be compared to those of UML and to represent them we could use simply a profile of UML. However, RSL defines a very detailed notation for requirements representations which are precisely linked to domain elements. This unique feature of RSL allows for capturing precise models of the software system's essence [10].

In concrete syntax, domain elements resemble UML classes with associations, as presented in the lower part of Figure 1. For our considerations we will concentrate on "Domain Elements" of type "Notion". "Notions" represent business entities, buttons, windows and other elements that occur in the problem and system domain. Each "Notion" has a name represented as a "Noun Phrase" and contains "Domain Statements" with "Phrases" coming from use case scenarios. "Domain Elements" can be structured through specifying relationships and generalizations between them. Some "Notions" can be defined as attributes of other "Notions". Different types of notions are distinguished through their "Stereotypes".

An important part of the RSL meta-model is centred around "Phrases", which is presented in Figure 2. Phrases occur in scenario sentences and in domain statements. Phrases are divided into "Noun Phrases" and "Verb Phrases", where the second type can be further divided into "Simple Verb Phrases" and "Complex Verb Phrases". A "Noun Phrase" contains a "Noun" and an optional "Modifier", which can describe the "Noun" more precisely. A "Simple Verb Phrase" can be used as a sentence predicate and consists of a verb and a noun e.g. *adds* **selected student**. "Complex Verb Phrases" extend "Simple Verb Phrases" with a preposition and an additional "Noun Phrase" representing the indirect object. More detailed description of RSL syntax and its role in code generations can be found in [29] and [11].

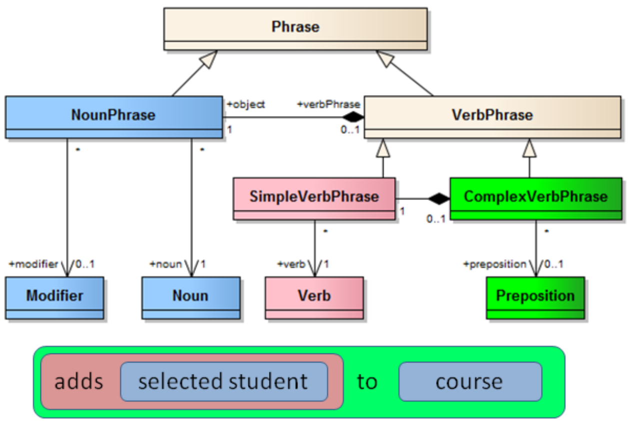


Figure 2. Abstract and concrete syntax for phrases

For the purpose of generating user interface code, the above meta-model of domain elements in RSL needed some additions. The fundamental distinction takes the form of a "notion type" which stems from the "Stereotype" attached to a "Notion". Possible notion types are: Concept, Attribute, Simple View, List View, Screen, Message and Trigger. These types of notions can be used as sentence objects in use case scenarios. For user interface generation it is also important to distinguish verbs associated with these notion types, as part of verb phrases. The verbs like 'show', 'refresh' and 'close' are associated in phrases with Screens and form so-called System-to-Actor sentences (e.g. "**System** *shows* **new student window**"). The verbs like 'select' are associated with Triggers and form so-called Actor-to-System sentences (e.g. "**User** *selects* **save button**").

4. Semantics for Domain Elements in RSL

The notion types described at the end of the previous section have specific meaning, which determines correct generation of the user interface code. A **Concept** is a representation of a business entity stored and processed by the system. It has no direct impact on generating user interface code, however it groups atomic data attributes for the purpose of their further processing. An **Attribute** describes one of the Concept's properties just like class attributes in UML. It should be noted that in RSL, Attributes are separate model entities. Each Attribute should be connected with at least one Concept. Moreover, each Attribute has its specific type which describes specific kind of data it represents like Text, Number, Date etc. A **Screen** is a representation of a window or a web-page depending on the transformation's target technology (for web-based technologies like JavaFX, Echo3, a Screen will be transformed into a web-page and for desktop technologies like Swing – into a window). A variant of Screen is a **Message** that represents a simple modal window used to show some error or confirmation message to the user. It causes generation of a proper pop-up message window.

A **Simple View** represents a set of data made available to the user during some interaction with the system. It can point to the attributes of many different Concepts, but should have defined a main Concept. If the Simple View is connected with a Screen, its attributes will be used as the basis for the window content. For every Attribute connected to a Simple View, an appropriate user interface widget will be created. Its type will depend on the Attribute type; for example it will be a text field for a text or a number Attribute or a check-box for a true/false attribute. Attributes typed as Date should generate not only a labeled text field, but also a button to call a calendar pop-up with the possibility to select the date.

A **List View** is similar to a Simple View, however presents many instances of given data set in an ordered form. It causes generation of a table or a list. Attributes connected to a List View are used in the creation of its fields in the way analogical to that of a Simple View. Both Simple Views and List Views can be called Data Views. Data Views are usually connected to Screens. The direction of this relationship indicates type of access to data. Connection from a Screen to a Data View indicates that data will be entered, and connection from a Data View to a Screen indicates that some existing data will be presented. In case that there is no direction – both access types are assumed (modification of existing data).

A **Trigger** is a representation of a link, button or any other element of user interaction. Just like the Screen - it is a platform-independent term and its final form depends on a platform-specific transformation. Additionally, some Trigger invoking an operation, can be connected to a Data View that determines the data involved in that operation. There is no need need to define relations between Triggers and Screens. The transformation generates them based on scenarios assuring that there won't be any Trigger without functionality described in a scenario and there will be no Trigger described in a scenario which does not have a related Screen.

In addition to these domain elements, user interface elements are generated based on certain use case scenario configurations. This involves two types of sentences. A System-to-Actor sentence refers to a Screen or Message. It denotes an interaction of the system with an actor through displaying a window or message. These kind of sentences result in generating code that contains invocations of methods to display, refresh or close some user interface element. An Actor-to-System sentence refers to a Trigger. It denotes an interaction of an actor with the system through selecting some active element (button, hyperlink) in a window. An Actor-to-System sentence generates an appropriate event handler code. This code is generated in the code of the user interface element that was referred by a previous System-to-Actor sentence.

5. Code Generation Process

Figure 3 presents an overview of the software development process using the ReDSeeDS tool. The first step is to formulate and write requirements in RSL according to the rules described in the previous sections. The tool supports this process by offering a specialized scenario editor, automatic notion creation, notion editor with type assignments and much more.

The next step in the process is to execute a model transformation and generate detailed-design-level UML models with embedded code. The appropriate transformation program for generating the user interface elements was developed in the language MOLA (MOdel transformation LAnguage) [30]. MOLA is a graphical language which uses pattern matching algorithms on meta-model level to transform one model into another. In our case, this will be an RSL model translated into a UML class model with inserted code fragments. MOLA contains both declarative and imperative constructs. The declarative elements include rules which represents queries on the model, connected with indications which elements should be created or deleted. MOLA declarative rules are presented as gray rectangles with rounded corners, containing objects from the meta-model. Query elements have solid black borders, whereas create elements have thick red dashed borders. Imperative elements include control flows between the rules which are denoted by dashed arrows in a notation similar to UML's activity diagrams. Also, loops are possible, which are denoted by thick black boxes with rules that are to be iterated, contained inside them. The first rule inside a loop is the loop's iterator rule with one element being a loop-head and denoted with a thicker border. MOLA is also a procedural language, where procedure calls are denoted with special actions with procedure names and parameters. Procedure definitions declare these parameters as large arrow-shaped boxes. Procedures also declare variables as white rectangular boxes.

To present the idea of the user interface generation program, we provide three of its fragments.

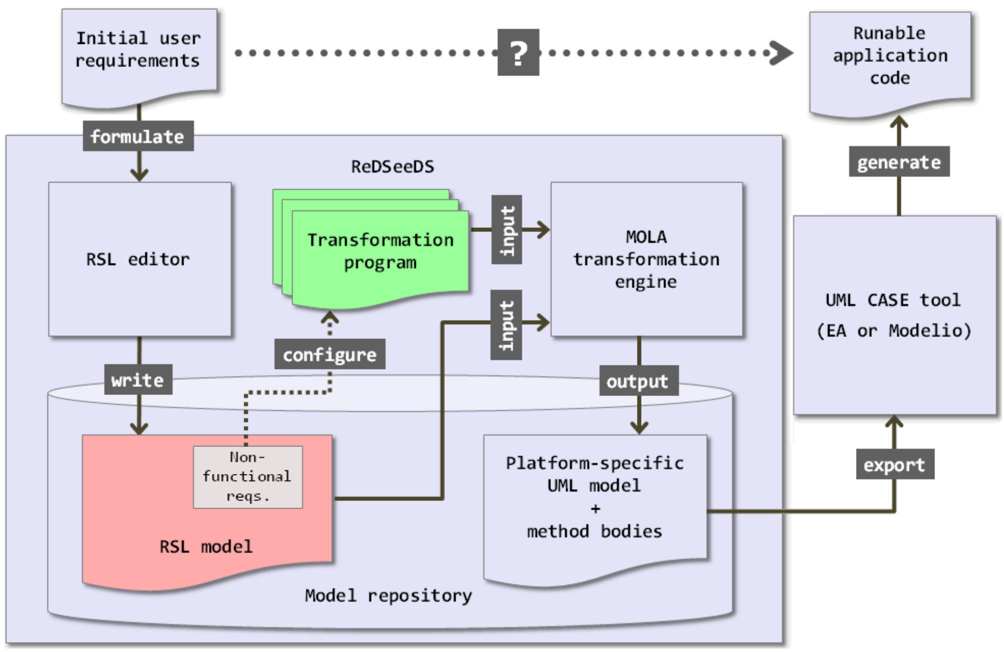


Figure 3. Process overview

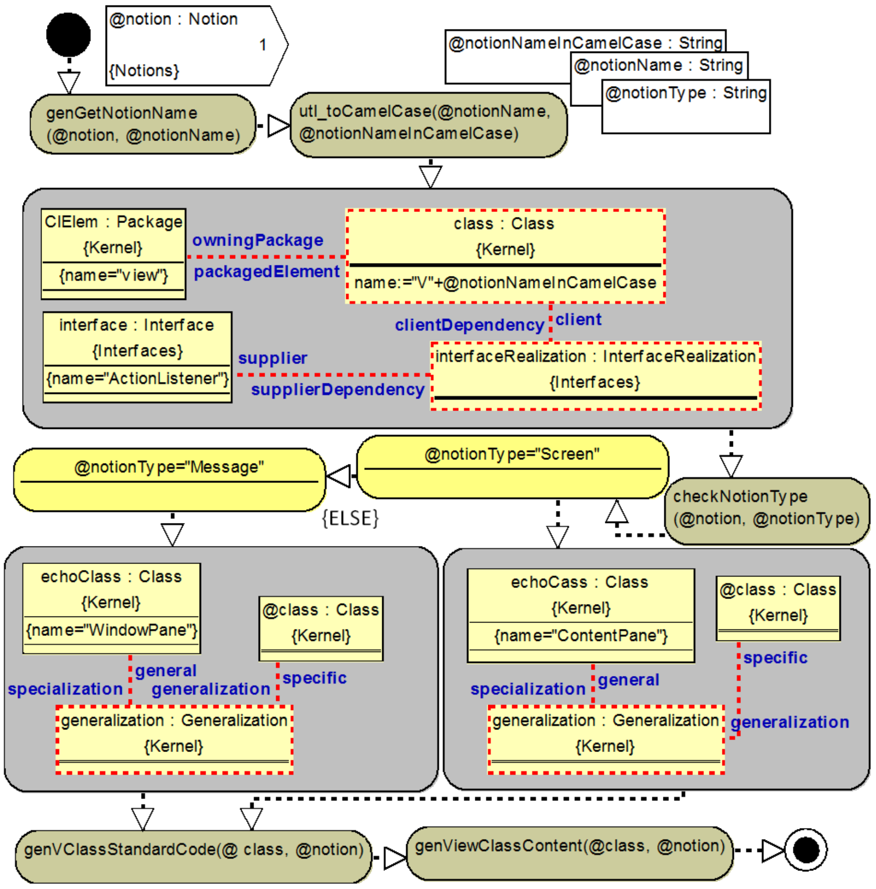

@notion : Notion
1
{Notions}
@notionNameInCamelCase : String
@notionName : String
@notionType : String
genGetNotionName (@notion, @notionName)
utl_toCamelCase(@notionName, @notionNameInCamelCase)
ClElem : Package {Kernel} {name="view"}
owningPackage
packagedElement
class : Class {Kernel}
name:="V"+@notionNameInCamelCase
clientDependency
client
interface : Interface {Interfaces} {name="ActionListener"}
supplier
supplierDependency
interfaceRealization : InterfaceRealization {Interfaces}
@notionType="Message"
@notionType="Screen"
{ELSE}
checkNotionType (@notion, @notionType)
echoClass : Class {Kernel} {name="WindowPane"}
@class : Class {Kernel}
general
specialization
generalization
specific
generalization : Generalization {Kernel}
echoClass : Class {Kernel} {name="ContentPane"}
@class : Class {Kernel}
specific
specialization
general
generalization : Generalization {Kernel}
generalization
genVClassStandardCode(@ class, @notion)
genViewClassContent(@class, @notion)

Figure 4. Procedure ('genViewClass') for generating classes from Screens and Messages

The actual transformation is much more elaborate and thus had to be simplified and abridged. Figure 4 shows the basic procedure ('genViewClass') for the creation of classes that handle widow-related code. These classes stem from the Screens or Messages. The appropriate Notion is given as the parameter to this procedure. After retrieving the Notion's name and converting it to camel case format, the procedure creates a properly named class (prefixed with 'V'). This class realises (see 'InterfaceRealisation') the standard 'ActionListener' interface. Then, the notion type is checked and depending on this, appropriate generalisation is created with either the standard 'WindowPane' or 'ContentPane' class. After this, the transformation calls the procedure to generate common code for all such classes ('genVClassStandardCode') and code individual for each class ('genViewClassContent').

Figure 5 shows this second procedure, which is more interesting. It generates the contents of the previously generated class, based on the features of the appropriate Notion and the associated elements. Firstly, the transformation checks the direction of the relation between the given Notion (Screen or Message) and another Notion which is a Simple View or a List View. Then depending on this determined direction, it assigns the type of access to window elements, to be provided by the generated controls. After that, the transformation generates an operation ('addContent') to fill the window content and fills it with standard code ('generateContentGridCode'). The last part of the procedure contains two loops (for two possible directions between the Notion and its related Data Views). In each iteration, an appropriate Data View and its Attributes are processed and appropriate field initiation code is created and inserted into the 'addContent' method.

Figure 6 shows fragment of the procedure that generates the actual field initiation code. Firstly, the standard initial part of code for the control group is generated through a call to an appropriate procedure. Then, a loop is performed for each Attribute pointed-to from the Data View which is the procedure's parameter. Inside the loop, firstly, the notion name is retrieved and converted to camel case format. After that, a private class property (attribute) is generated to hold the label field for the given Attribute. Then, the Attribute's data type is retrieved and depending on it, a property (attribute) for holding the actual control type is generated. For simplicity, the Figure shows fully only the fragment associated with the generation of Text Fields. The last part of the loop contains a call to the procedure that generates the proper code that initialises the just generated attributes.

As we can see, the output of the presented transformation is a UML model consisting of classes with attributes, operations and code embedded in these operations. The next step is to export this UML model and generate code with a UML tool providing an appropriate code generator (see Fig. 3). The code generator is invoked automatically and thus from the user perspective is seen as part of the overall transformation process. ReDSeeDS currently supports export and code generation using Enterprise Architect [31] and Modelio [32]. The full generated code complies with the Model-View-Presenter pattern [33] and is also based on the Echo framework [34].

6. Illustrative Example

The presented approach has been validated during a case study which was to implement a sports centre management system. This involved about 30 use cases, of which some are presented in Figure 7. In this brief example, we will show mainly the code generated around the domain models for the use case surrounded by the green thick frame ("Add promotion"). This use case has two scenarios, presented in Figure 8. In addition, we will show the user interface generated from the use case surrounded by a dashed blue frame ("Display Promotion management"). This will allow to present support for generating lists.

Figure 9 presents the domain model that complements the scenarios of "Add promotion", together with the actually generated user interface for the "new promotion form" window. We can observe that "new promotion form" is a Screen which points at "promotion data" which

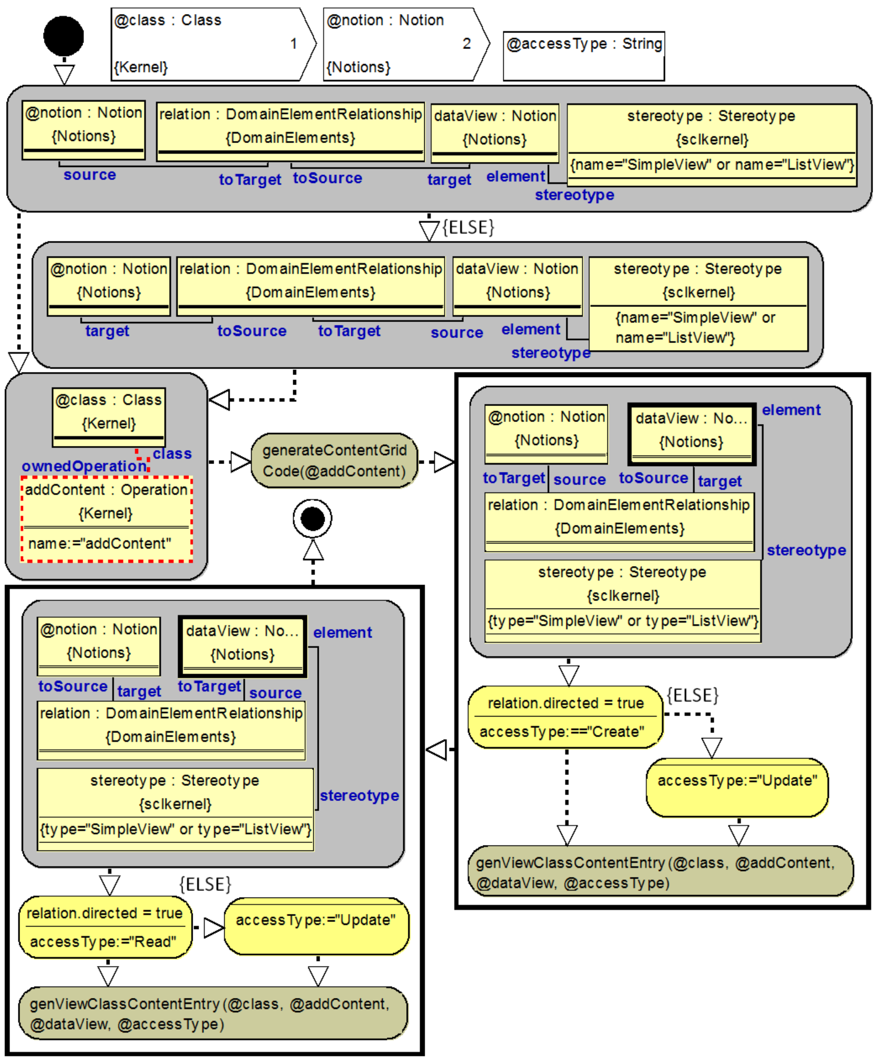


Figure 5. Procedure (‘genViewClassContent’) for generating the ‘addContent’ operation

is a Simple View. This results in generating the “new promotion form” window (1) with the appropriate section corresponding to “promotion data” (2). The connection is directed from the Screen to the Simple View, which means that the window will serve entering data. The main Concept associated with “promotion data” is the “promotion”. The Simple View points to several Attributes contained in the “promotion” and in the associated “promotion type” Concept. This set of relations to Attributes means that the section corresponding to “promotion data” will be filled with controls to input data related to the mentioned Attributes.

The types of these controls depends on the data types of the given attributes. We can see the equivalence in Figure 9. For instance, “Promotion name” (3) typed as Text is created as a Text field, and “Expiration date” (6) typed as Date is generated as a Text field with a button to open the date chooser.

A special case is the “Promotion type” (4) which is part of a Concept that is not the main Concept. It is generated into an separate embedded group of labelled controls. In this particular case, only one Text field (“Promotion type name”) is generated from the appropriate Attribute. We can also notice an additional button (“Select”) which was not covered by the semantic rules in the previous sections and can be used to select one value from a pop-up list. “Promotion type” takes the form of an embedded group

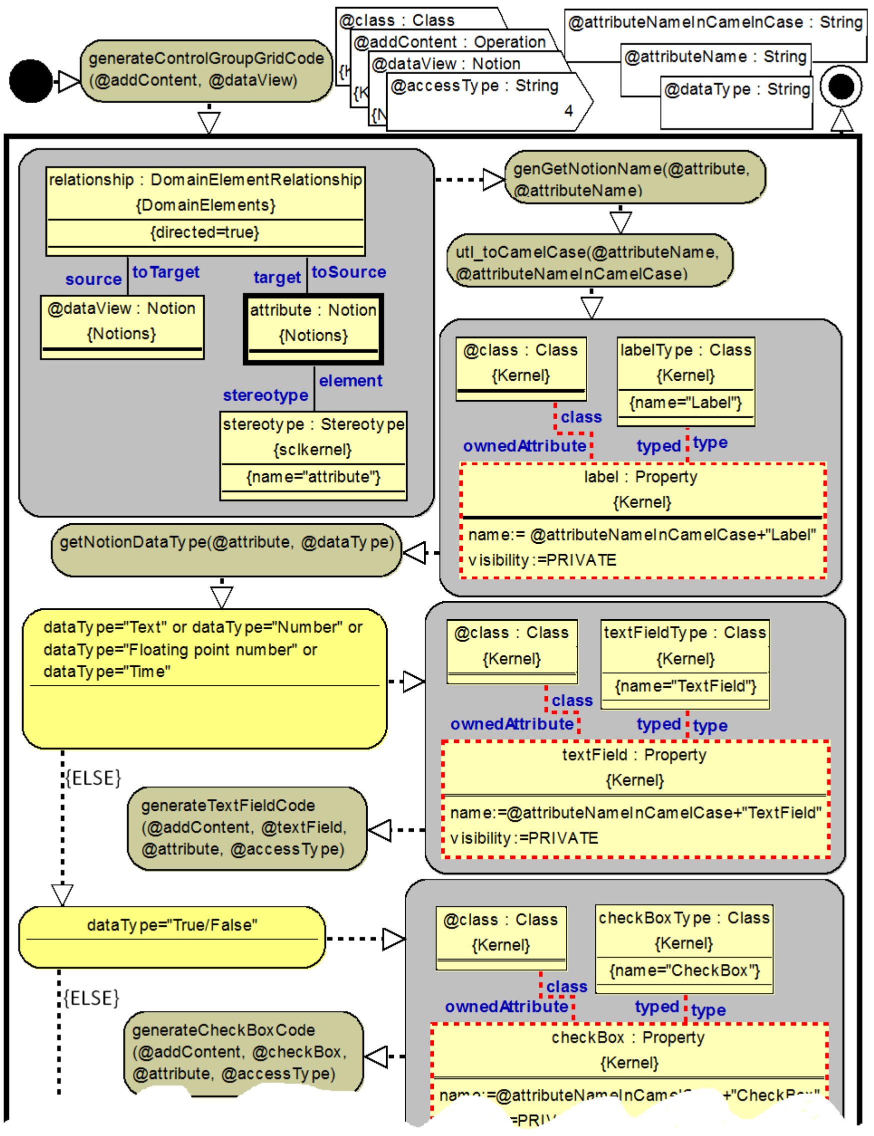

generateControlGroupGridCode (@addContent, @dataView)
@class : Class
@addContent : Operation
@dataView : Notion
@accessType : String
4
@attributeNameInCamelInCase : String
@attributeName : String
@dataType : String
relationship : DomainElementRelationship {DomainElements} {directed=true}
source
toTarget
target
toSource
@dataView : Notion {Notions}
attribute : Notion {Notions}
element
stereotype
stereotype : Stereotype {sclkernel} {name="attribute"}
genGetNotionName(@attribute, @attributeName)
utl_toCamelCase(@attributeName, @attributeNameInCamelCase)
@class : Class {Kernel}
labelType : Class {Kernel} {name="Label"}
class
ownedAttribute
typed
type
label : Property {Kernel}
name:= @attributeNameInCamelCase+"Label"
visibility:=PRIVATE
getNotionDataType(@attribute, @dataType)
dataType="Text" or dataType="Number" or dataType="Floating point number" or dataType="Time"
textFieldType : Class {Kernel} {name="TextField"}
textField : Property {Kernel}
name:=@attributeNameInCamelCase+"TextField"
visibility:=PRIVATE
{ELSE}
generateTextFieldCode (@addContent, @textField, @attribute, @accessType)
dataType="True/False"
checkBoxType : Class {Kernel} {name="CheckBox"}
checkBox : Property {Kernel}
generateCheckBoxCode (@addContent, @checkBox, @attribute, @accessType)

Figure 6. Procedure (‘genViewClassContentEntry’) for generating field initiation code

of controls, not a list, because of the singular multiplicity of the relationship between the promotion (main Concept) and the promotion type (associated Concept).

Code for creating these controls as the content of “new promotion form” is shown in Figure 10. Apart from generating the fields, code contains creation of the “Add promotion” button. This is based on sentence 4, in relation to sentence 2 of the use case scenario shown in Figure 8. The code generator produces also an event handler associated with this button, presented in Figure 11. This is presented to show completeness and coherence of the generated code but more detailed discussion is out of scope of this paper.

In addition to generating simple forms, the code generator can produce lists from List View elements. This is illustrated in Figure 12. The situation is in most part similar to the previous case, but data is represented in a collection form because a List View is used instead of a Simple View. Moreover, only some of the Attributes of the “promotion” are presented on the screen, because not all are connected to the List View.

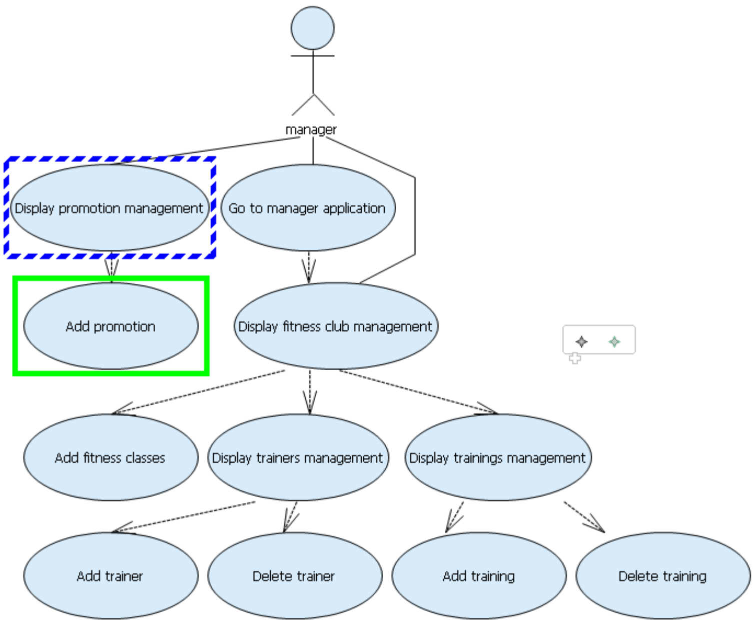
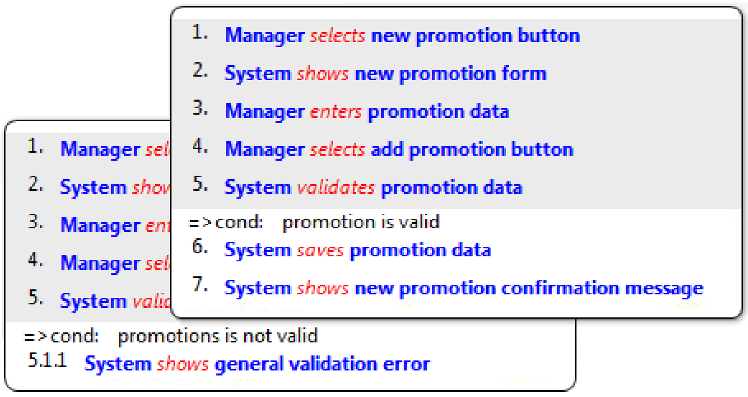

manager
Display promotion management
Go to manager application
Add promotion
Display fitness club management
Add fitness classes
Display trainers management
Display trainings management
Add trainer
Delete trainer
Add training
Delete training

Figure 7. Use case model fragment for the case study



1. Manager *selects* new promotion button
2. System *shows* new promotion form
3. Manager *enters* promotion data
4. Manager *selects* add promotion button
5. System *validates* promotion data

=>cond: promotion is valid

6. System *saves* promotion data
7. System *shows* new promotion confirmation message

1. Manager *sel*
2. System *shov*
3. Manager *ent*
4. Manager *sel*
5. System *valic*

=>cond: promotions is not valid

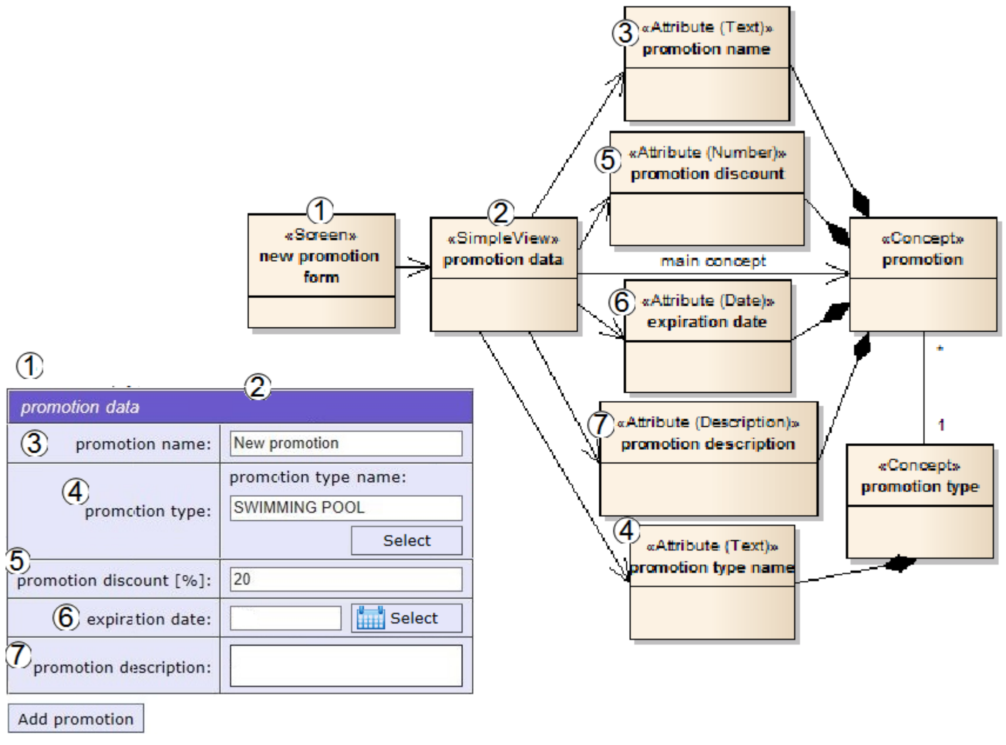
5.1.1 System *shows* general validation error

Figure 8. Scenarios of the "Add promotion" use case

7. Conclusion and Future Work

The presented approach aims to give the requirements model the feature of executability. The functional requirements are represented using the Requirements Specification Language in which emphasis is placed on both readability and precision. Using the presented transformation program in combination with a precise RSL specification, we obtain a typical business application, with simple, but fully functional graphical user interface, ready for deployment. Still, we can find some limitations of the presented transformations due to limitations of the current RSL syntax for domain elements. However, we plan to remove these limitations by extending the RSL notation and refining its semantics.

Currently, this approach can be used with success for fast prototyping. furthermore, through refinement of the graphical interface arrangement and use of appropriate outlook styles, it can also be brought to the condition of the final product. The presented solution is being validated on a much larger case study based on a legacy corporate banking system. Furthermore there are plans to conduct experiments with university students. Their goal will be to compare productivity and quality when the presented solution is used versus traditional approaches.


«Attribute (Text)» promotion name
«Attribute (Number)» promotion discount
«Screen» new promotion form
«SimpleView» promotion data
main concept
«Concept» promotion
«Attribute (Date)» expiration date
«Attribute (Description)» promotion description
«Concept» promotion type
«Attribute (Text)» promotion type name
*
1
1
2
3
4
5
6
7

promotion data

promotion name:	New promotion
promotion type:	promotion type name: SWIMMING POOL [Select]
promotion discount [%]:	20
expiration date:	[Select]
promotion description:	

Add promotion

Figure 9. Domain model for the "new promotion form"

Future development work will include extending the ReDSeeDS tool with an editor to enable management of user interface element arrangement. There are also plans to further develop the overall transformation, taking into account various new technologies and platforms. There is ongoing work on developing new transformations which will provide high-level separation of concerns and thereby high reusability. In the future, the transformations are planned to offer several technology options to build the presentation layer such as Google Web Toolkit, Apache Wicket, JavaFX, Adobe Flex.

Acknowledgment

Part of this research has been carried out in the REMICS project and partially funded by the EU (contract number ICT-257793 under the 7th Framework Programme), see http://www.remics.eu.

References

[1] K. El Emam, "A Replicated Survey of IT Software Project Failures," *IEEE Software*, Vol. 25, No. 5, 2008, pp. 84–90.

[2] B. H. C. Cheng and J. Atlee, "Research Directions in Requirements Engineering," in *Future of Software Engineering, FOSE '07*, 2007, pp. 285–303.

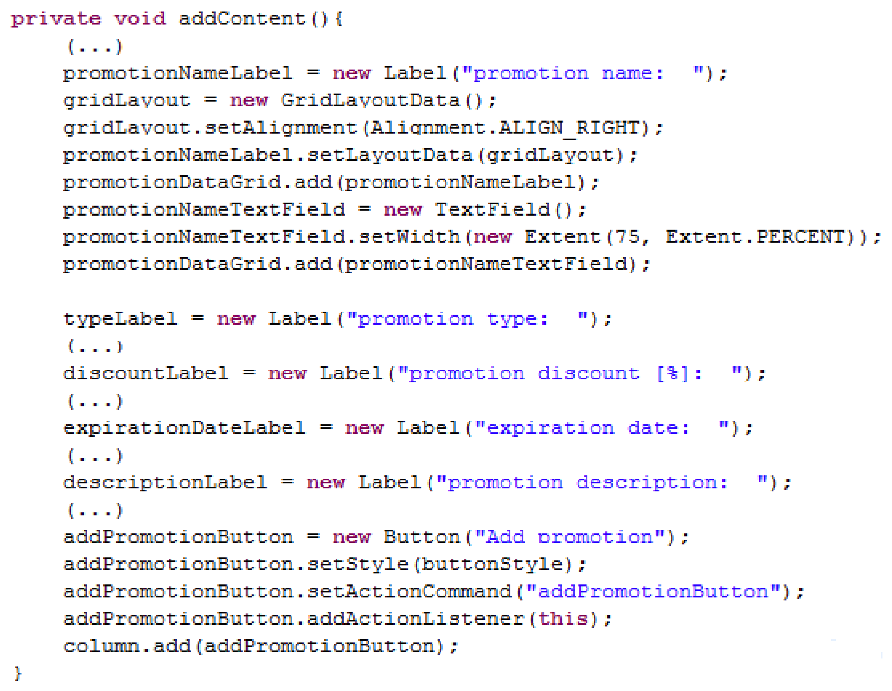
[3] B. Berenbach, "A 25 year retrospective on model-driven requirements engineering," in *Model-Driven Requirements Engineering Workshop (MoDRE), 2012 IEEE*, 2012, pp. 87–91.

[4] M. Brambilla, J. Cabot, and M. Wimmer, *Model-driven Software Engineering in Practice.* Morgan & Claypool, 2012.

[5] D. Schmidt, "Guest Editor's Introduction: Model-Driven Engineering," *Computer*, Vol. 39, No. 2, 2006, pp. 25–31.

[6] S. Greenspan, J. Mylopoulos, and A. a. Borgida, "Capturing More World Knowledge in the Requirements Specification," in *Proc. 6th International Conference on Software Engineering.* IEEE Computer Society Press, 1982, pp. 225–234.

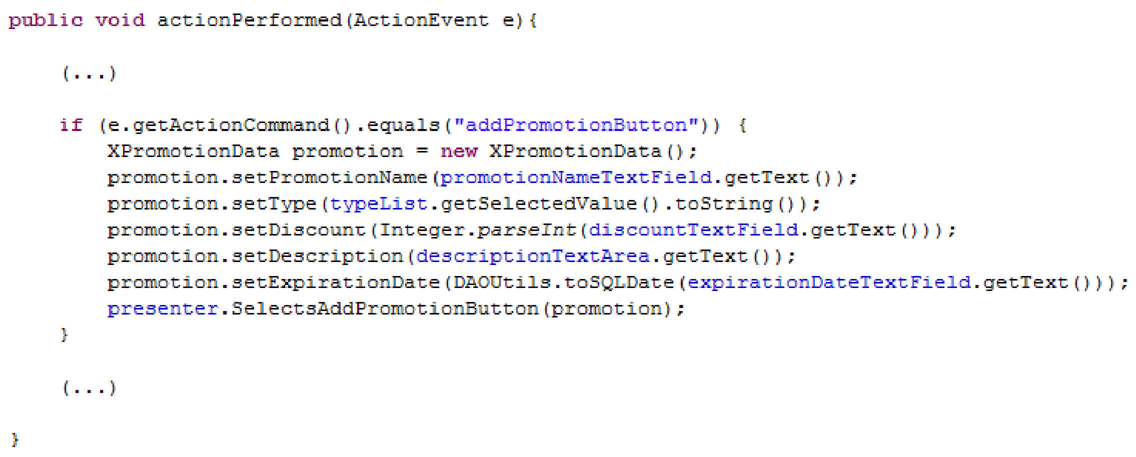
[7] S. Greenspan, J. Mylopoulos, and A. Borgida, "On formal requirements modeling languages: RML revisited," in *ICSE '94: Proc. 16th International Conference on Software Engineering.* Los



```
private void addContent(){
    (...)
    promotionNameLabel = new Label("promotion name:  ");
    gridLayout = new GridLayoutData();
    gridLayout.setAlignment(Alignment.ALIGN_RIGHT);
    promotionNameLabel.setLayoutData(gridLayout);
    promotionDataGrid.add(promotionNameLabel);
    promotionNameTextField = new TextField();
    promotionNameTextField.setWidth(new Extent(75, Extent.PERCENT));
    promotionDataGrid.add(promotionNameTextField);

    typeLabel = new Label("promotion type:  ");
    (...)
    discountLabel = new Label("promotion discount [%]:  ");
    (...)
    expirationDateLabel = new Label("expiration date:  ");
    (...)
    descriptionLabel = new Label("promotion description:  ");
    (...)
    addPromotionButton = new Button("Add promotion");
    addPromotionButton.setStyle(buttonStyle);
    addPromotionButton.setActionCommand("addPromotionButton");
    addPromotionButton.addActionListener(this);
    column.add(addPromotionButton);
}
```

Figure 10. Fragment of code for creating content of “new promotion form”



```
public void actionPerformed(ActionEvent e){

    (...)

    if (e.getActionCommand().equals("addPromotionButton")) {
        XPromotionData promotion = new XPromotionData();
        promotion.setPromotionName(promotionNameTextField.getText());
        promotion.setType(typeList.getSelectedValue().toString());
        promotion.setDiscount(Integer.parseInt(discountTextField.getText()));
        promotion.setDescription(descriptionTextArea.getText());
        promotion.setExpirationDate(DAOUtils.toSQLDate(expirationDateTextField.getText()));
        presenter.SelectsAddPromotionButton(promotion);
    }

    (...)

}
```

Figure 11. Handler code for the “add promotion” button

Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 135–147.

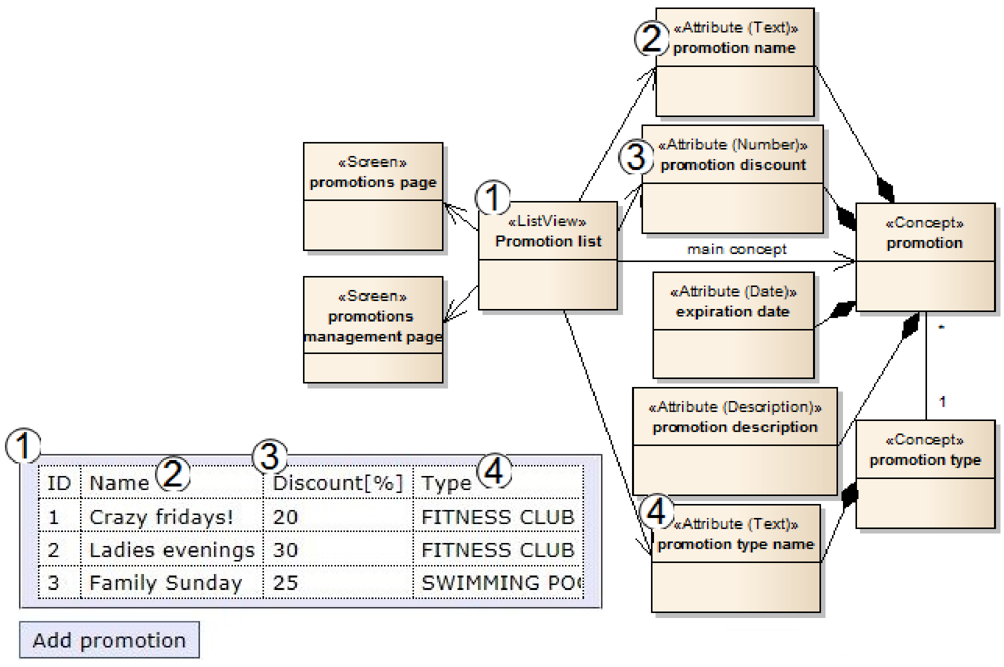
[8] H. Kaindl, M. Śmiałek, , P. Wagner, D. Svetinovic, A. Ambroziewicz, J. Bojarski, W. Nowakowski, T. Straszak, H. Schwarz, D. Bildhauer, J. P. Brogan, K. S. Mukasa, K. Wolter, and T. Krebs, “Requirements Specification Language Definition,” ReDSeeDS Project, Project Deliverable D2.4.2, 2009. [Online]. www.redseeds.eu

[9] J. Helming, M. Koegel, F. Schneider, M. Haeger, C. Kaminski, B. Bruegge, and B. Berenbach, “Towards a unified Requirements Modeling Language,” in *Requirements Engineering Visualization (REV), 2010 Fifth International Workshop on*, Sept 2010, pp. 53–57.

[10] W. Nowakowski, M. Śmiałek, A. Ambroziewicz, and T. Straszak, “Requirements-Level Language and Tools for Capturing Software System Essence,” *Computer Science and Information Systems*, Vol. 10, No. 4, 2013, pp. 1499–1524.

[11] M. Śmiałek, N. Jarzebowski, and W. Nowakowski, “Translation of Use Case Scenarios to Java Code,” *Computer Science*, Vol. 13, No. 4, 2012, pp. 35–52.

[12] M. Śmiałek, W. Nowakowski, N. Jarzebowski,


«Attribute (Text)» promotion name
«Attribute (Number)» promotion discount
«Screen» promotions page
«ListView» Promotion list
main concept
«Concept» promotion
«Screen» promotions management page
«Attribute (Date)» expiration date
*
1
«Attribute (Description)» promotion description
«Concept» promotion type
«Attribute (Text)» promotion type name
1
2
3
4
Add promotion

ID	Name	Discount[%]	Type
1	Crazy fridays!	20	FITNESS CLUB
2	Ladies evenings	30	FITNESS CLUB
3	Family Sunday	25	SWIMMING PO(

Figure 12. Domain model for the "promotion list"

and A. Ambroziewicz, "From Use Cases and Their Relationships to Code," in *Second IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012*. IEEE, 2012, pp. 9–18.

[13] T. Straszak and M. Śmiałek, *Advances in Software Development*. Polish Information Processing Society, 2013, ch. Acceptance test generation based on detailed use case models, pp. 116–126.

[14] "ReDSeeDS project home page," http://redseeds.eu/.

[15] M. Śmiałek, A. Ambroziewicz, J. Bojarski, W. Nowakowski, and T. Straszak, "Introducing a unified Requirements Specification Language," in *Proc. CEE-SET'2007, Software Engineering in Progress*. Nakom, 2007, pp. 172–183.

[16] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, Vol. 20, No. 4, April 1987, pp. 10–19.

[17] P. Shaker, J. Atlee, and S. Wang, "A feature-oriented requirements modelling language," in *Requirements Engineering Conference (RE), 2012 20th IEEE International*, 2012, pp. 151–160.

[18] M. El-Attar and J. Miller, "AGADUC: Towards a More Precise Presentation of Functional Requirement in Use Case Mod," in *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, 2006, pp. 346–353.

[19] R. Giganto and T. Smith, "Derivation of Classes from Use Cases Automatically Generated by a Three-Level Sentence Processing Algorithm," in *Systems, 2008. ICONS 08. Third International Conference on*, 2008, pp. 75–80.

[20] S. Mustafiz, J. Kienzle, and H. Vangheluwe, "Model transformation of dependability-focused requirements models," in *Modeling in Software Engineering, 2009. MISE '09. ICSE Workshop on*, 2009, pp. 50–55.

[21] D. K. Deeptimahanti and R. Sanyal, "Semi-automatic generation of UML models from natural language requirements," in *Proceedings of the 4th India Software Engineering Conference*, ser. ISEC '11, 2011, pp. 165–174. [Online]. http://doi.acm.org/10.1145/1953355.1953378

[22] *Unified Modeling Language: Superstructure, version 2.2, formal/09-02-02*, Object Management Group, 2009.

[23] "AndroMDA project home page," http://andromda.org/.

[24] "MDA website," http://omg.org/mda/.

[25] Y. Wang and M. Wu, "Case studies on translation of RTPA specifications into Java programs," in *Canadian Conference on Electrical and Computer Engineering*, Vol. 2, 2002, pp. 675–680.

[26] J. Falb, S. Kavaldjian, R. Popp, D. Raneburger, E. Arnautovic, and H. Kaindl, "Fully Automatic User Interface Generation from Discourse Models," in *Proceedings of the 14th International Conference on Intelligent User Interfaces*, ser. IUI '09. New York,

NY, USA: ACM, 2009, pp. 475–476. [Online]. http://doi.acm.org/10.1145/1502650.1502722

[27] C. Janssen, A. Weisbecker, and J. Ziegler, "Generating User Interfaces from Data Models and Dialogue Net Specifications," in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, ser. CHI '93. New York, NY, USA: ACM, 1993, pp. 418–423. [Online]. http://doi.acm.org/10.1145/169059.169335

[28] M. ElKoutbi, I. Khriss, and R. Keller, "Generating user interface prototypes from scenarios," in *Requirements Engineering, 1999. Proceedings. IEEE International Symposium on*, 1999, pp. 150–158.

[29] M. Śmiałek, N. Jarzebowski, and W. Nowakowski, "Runtime semantics of use case stories," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, Sept 2012, pp. 159–162.

[30] A. Kalnins, J. Barzdins, and E. Celms, "Model Transformation Language MOLA," *Lecture Notes in Computer Science*, Vol. 3599, 2004, pp. 14–28, MDAFA'04.

[31] "Enterprise Architect Website," http://www.sparxsystems.com/products/ea/.

[32] "Modelio Website," http://www.modelio.org/.

[33] M. Potel, "MVP: Model-View-Presenter, The Taligent Programming Model for C++ and Java," Taligent Inc., Tech. Rep., 1996, http://www.wildcrest.com/Potel/Portfolio/mvp.pdf.

[34] "Echo Framework Home Page," http://echo.nextapp.com/.