Rich design models

Beyond the conventional modeling

Tomas Cerny
Department of Computer Science and Engineering
Czech Technical University
Prague, Czech Republic
cernyto3@fel.cvut.cz

Eunjee Song
Department of Computer Science
Baylor University
Waco, TX, USA
eunjee_song@baylor.edu

***Abstract*—Model Driven Development (MDD) has provided a new way of engineering today's rapidly changing requirements into codes. However, creating view forms has usually been done manually although it is still one of the most important parts in developing and operating data collecting applications. For example, in case of Java 2 Enterprise Edition (Java EE) web applications, developers create view forms, i.e., rich user interfaces, manually by referring to entity beans to determine the contents of forms, but such a manual creation and maintenance is pretty tedious and certainly very much error-prone. One promise in MDD is an automatic generation of concrete artifacts (e.g., code) from its abstractions (e.g., design models). Existing design models in MDD, however, do not provide as detailed class attributes as are required in generating rich user interfaces. In this paper, we propose an approach to specifying rich design models that can capture more detailed information of class attributes and also show how the current MDD approach should be extended to support the automatic form generation from our proposed rich design models.**

Keywords: Rich User Interfaces, Model Driven Development, Rich Models.

I. Introduction

Our goal is to generate a rich user interface[1] directly from our design model. Rich user interface is a term used with connection to standalone and web applications. In the past rich user interface was offered only by standalone applications but now with Web 2.0 [17] we can provide the same for internet applications. These applications are called Rich Internet Applications (RIA), they offer a rich, engaging experience and advanced user interface features that improve user productivity and satisfaction [18,19]. In this paper we evolve the idea of our previous work form builder [5]. The idea proposes to keep additional field information in form of annotations for defining what the field means so we can evaluate correctness and also generate rich user interface (we focus on data input interface called forms). In the past work we have shown rich user interface generation from entity beans and Java code. In this paper we promote this idea in MDA. We propose an annotation alternative for model class attributes which also allows us to define additional constraints for business logic.

1 We focus on rich view forms in this work since they reflect the data model. A form example is shown in figure 2 and 3.

A. *Motivation example*

Assume that we are to generate an application from the simple design model in figure 1.

In the design model are not captured information of the meaning of a particular field. The only information for correctness evaluation and view form generation is the field type. We can generate the view form in the figure 2.

This view form data cannot be validated properly since we do not have any additional information about its fields. For example, a person can be born in future, has invalid email, one letter name, malformed web link and negative salary. This is not a view that we are satisfied with and it is a reason for many developers to develop their front-end manually. The view form that we want to create is shown in figure 3.

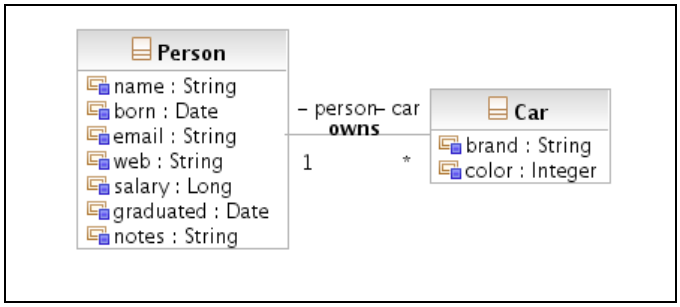

Figure 1. Example of a design model

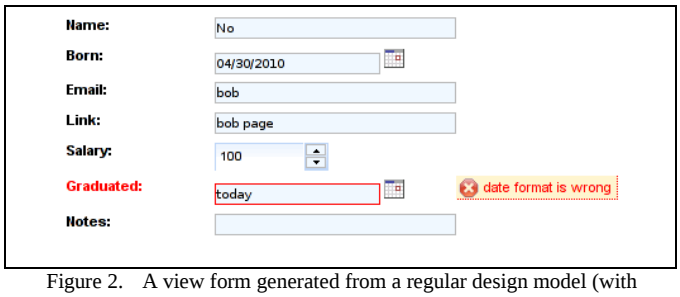

Figure 2. A view form generated from a regular design model (with validation).

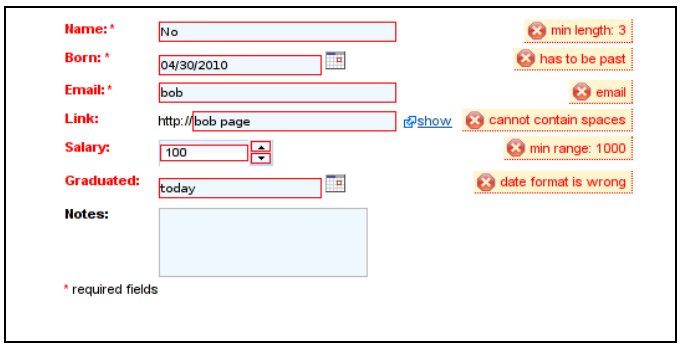

Figure 3. A view form that we want to create (the form shows a submission data validation).

In this paper we show a way how to generate the rich view form in figure 3 directly from the design model. This view form satisfies all our constraints and our static business rules (with static we mean a data independent).

B. *Paper organization*

Our paper is organized as follows: in chapter two we introduce model driven architecture. In the next chapter we provide a motivation for our goals. Chapter 4 deals with class model attribute extension to capture an additional information about an attribute. It also provides our proposed solution using meta-modeling and object constraint language. Next chapter provides an example. The paper then mentions a related work, provides a summary and references.

II. Background

When we develop a software we have a plenty of options which direction to go. When we choose a particular option, it is mostly because of our policy, experiences, expectations or our requirements [6,7] that are easy to realize. Many software projects have a lot in common. It was a motivation for creating design patterns [8] and methods that simplify software development. To simplify software development we need to be unified with terminology and principles. Unification was one of the reasons why an Unified Modeling Language (UML) [1] was born. UML is so popular and widely used on the public that the language is taught in most universities around the world and every software engineer knows at least some subset. Language UML is under international, non-profit computer industry consortium Object Management Group (OMG) [1]. This group provides enterprise integration standards for a many various broadly used technologies. OMG's modeling standards, include not only UML but also new development direction called Model Driven Architecture (MDA)[2,16].

In the past a software development team used a middleware that was matching their project requirements. A problem with such a development directions is with product compatibility with various operating systems. A problem is also with later compatibility with another middlewares or just with customer's later requirements. MDA provides a solution by a middleware independence. The independence is achieved by the strength of UML which provides models that are platform independent and platform specific. MDA uses both types of models. Independent model represents project's business functionality and behavior and is called Platform Independent Model (PIM). Platform specific model provides technical information and is called Platform Specific Model (PSM).

MDA development process starts with modeling and evolving PIM. The PIM is later mapped to a PSM specific to a particular platform which is evolved with platform specific information and constraints. The PSM is then mapped to a programming language that is compiled in a final product (see figure 4.).

Since we have a platform independent model we can have mappings to many various platform specific models and it's code. Since many middlewares are very similar we can also define a bridge between two platform specific models or generated middleware specific codes (see figure 5.), as we did with FromBuilder [5]. Generated code holds the same information as the platform specific model. The difference is that the PSM is modeled by UML and code uses programming language which is a text. Programmer still can evolve generated code.

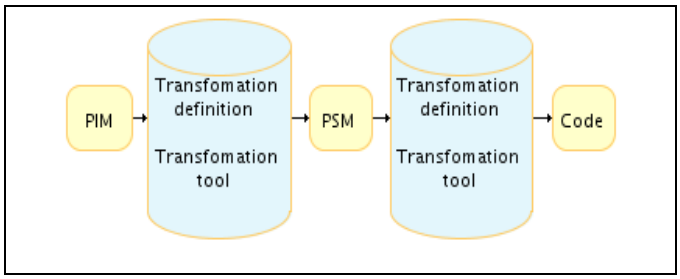

Figure 4. MDA process.

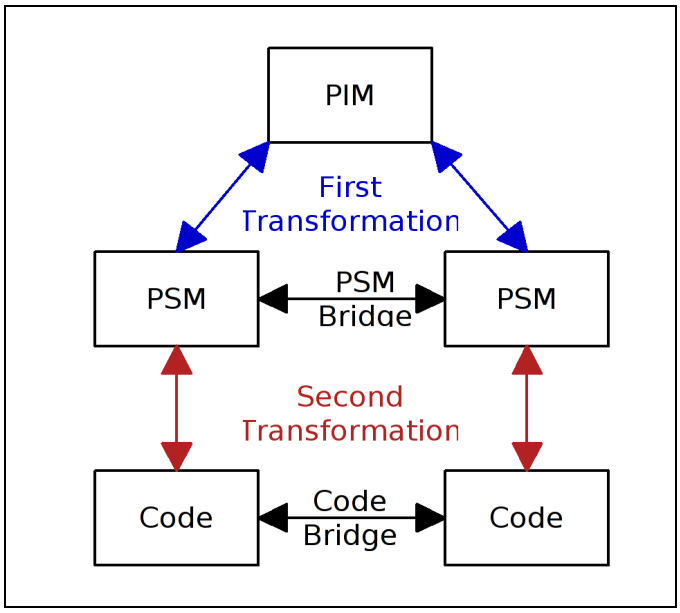

Figure 5. MDA interoperability using bridges.

The main advantage of MDA is that we design our project using UML models and then middleware experts provide mappings from PIM to PSM as well as PSM to code. It means that if our project was originally in an application using Delphi then we could easily convert it to Java 7. MDA is useful for standalone applications and it is becoming widely accepted approach for developing distributed applications [9].

III. FormBuilder motivation

Our previous work focused on data validation and view form generation form entity beans [5]. We explored dependencies between entity bean fields and view form elements. Entity bean field partially determines a type of an input element used in the form. The proper view form input element is chosen by a field type and by it's additional properties that are captured by Java annotations. These annotations are also used for field validation and for our constraints and static business rules. We call field types *horizontal properties* and annotations *vertical properties*. Vertical properties provide the meaning of the field.

Our main ideas can propagate in MDA. Since we have simplified form development by the usage of horizontal and vertical properties in entity beans. We can extend the idea to the design model. Since the design model can be rich as our proposed rich entity beans we can generate view forms directly from the design model. From MDA perspective our design model, which contains a few information about used middleware (SQL, EJB and Java), is a platform specific model (usage of SQL and Java types) [2, 16]. Considering distributed application frameworks like JSF, PHP or ASP that contain very similar data forms, we propagate the idea also into the platform independent model. The classification of our entity bean design model as the PSM could be doubtful. The only argument we can make is that our design model uses Java (SQL) specific data types. For example C++ contains unions, structures and unsigned numbers, and Java has only objects and signed numbers.

IV. Attributes constraints

Since we have proposed our goal, we need to describe how to achieve it. In an EJB entity beans we have used Java annotations for each field. In a design model (UML class diagram) is nothing similar to an annotation [10]. Each field (attribute) in a class diagram has an unique name, a type and a visibility. To be able to extend the attribute capability to hold also view specific information we need to know something about meta-modeling.

A. Meta-modeling

To describe meta-modeling and the MDA process lets summarize standard code compilation for regular programming languages [4] and show the similarities.

Every programming language has a defined syntax and semantics. To compile source code we need to go through two main stages. First we need to analyze the given source code and create an internal form. Then in the second stage we optimize this internal form and generate the final code (machine code). The whole process starts with a lexical analyzer, a processor of regular grammar for recognition of key words and variables, whose output is then passed to a syntactic analyzer which uses the context-free grammar to construct a derivation tree for syntax verification. The process is followed by internal form generation and semantics analysis (that is achieved by a translation from an attribute grammar). In this stage we verify declarations and do type conversions. The final code might be generated directly or by a translation to another language which may then generate the final code.

Meta-modeling mechanism does the same thing for MDA as the first stage of a programming language compiler, the second part is then done by a mapping from a PIM to a PSM and from the PSM to a code. The code can be then compiled by the programming language compiler. We already mentioned that MDA is driven by models. These models are syntactically and semantically described by meta-models, other-words meta-model is an instrument for model language definition. Similar to syntactic and semantic analyzer meta-modeling describes every element of a model language that can be used by the language. A quick example is for UML class diagram, where we use elements like classes, attributes and associations. UML meta-model then defines all the properties and characteristics for every model language element. Since a meta-model is also a model that needs to be described, it has it's meta-model that describes its semantics. Language of meta-model is called meta-language. Meta-language is different from a modeling language because it is used to describe modeling languages.

OMG uses four-layered architecture for MDA framework. Layers are called M0, M1, M2 and M3 where M0 describes behavior of instances (object model), M1 describes the system model (class model), M2 is a model of the M1 model describing classes, attributes and associations, where every element of M1 is an instance of an M2 element and finally M3 is a model of the M2 model where also every element of M2 is an instance of an M3 element. OMG defines Meta-object facility (MOF)[1] that is the standard M3 language. The architecture is shown by figure 6.

B. Solution to attributes constraints

Now we have enough instruments and knowledge to extend the capability of the class model attribute. This information is held by M2 layer of OMG standard architecture. We also need to consider that there are various constraints for different data types. The solution is to define an abstract class *Constraints* which will hold a general information and will be overridden with constraints for a particular type. See figure 7.

Since we already defined annotations necessary for rich user interface generation in FormBuilder [5] lets do the same here in a detail with providing a wider collection. Lets start with the abstract class *Constraints* that will contain attributes *required, unique, ignore, formOrder, tableColumn, inputLength = 30 and compareTo.* (see Figure 8 and table 1.).

The attribute *compareTo* is different from the others. It will validate that an annotated attribute is lower, equal, greater.. than another class attribute. An example might be for a competition system where a registration date must precede a competition date. This constraint has additional requirements

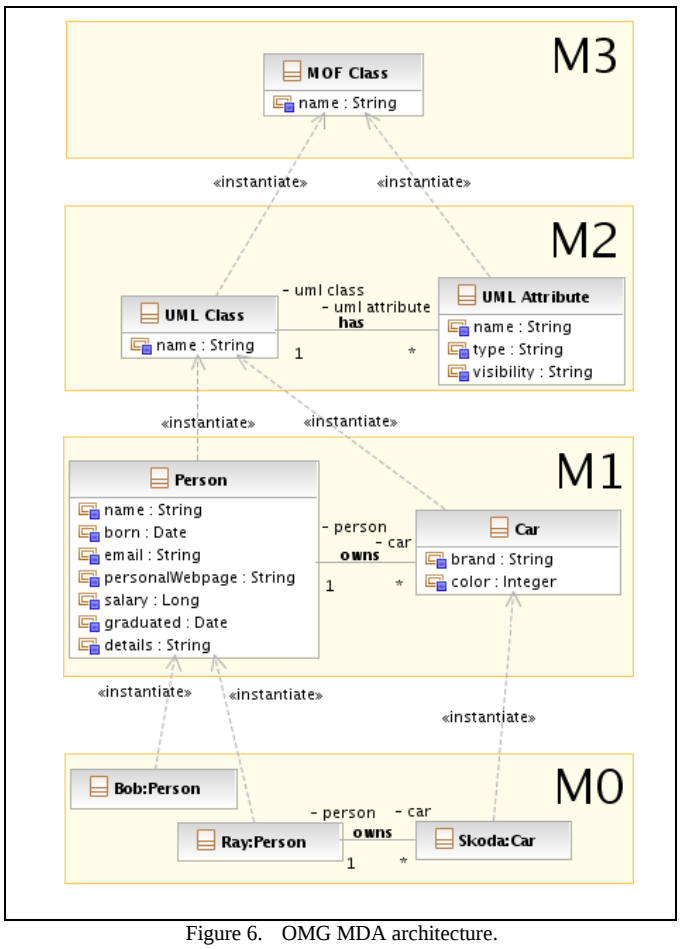

Figure 6. OMG MDA architecture.

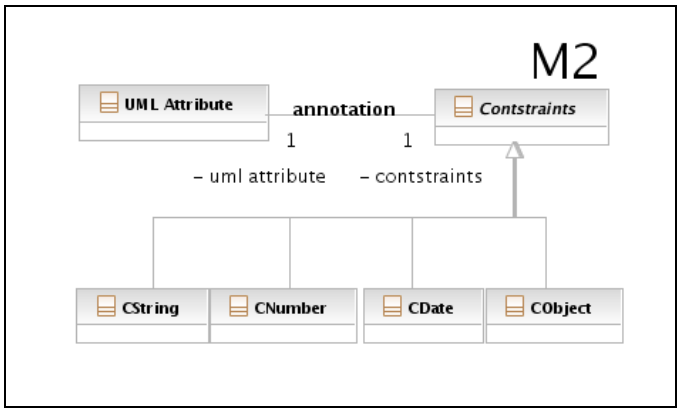

Figure 7. Extension for attribute class

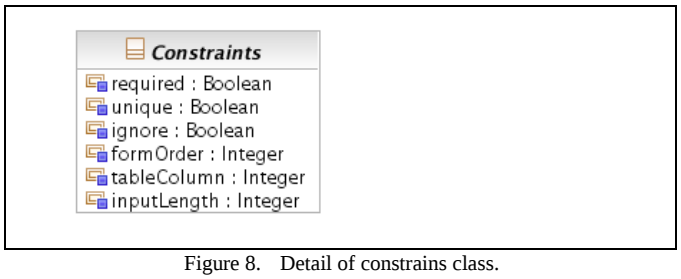

Figure 8. Detail of constrains class.

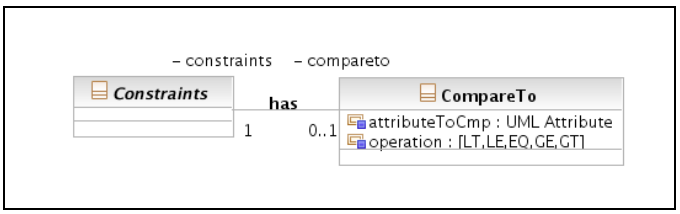

Figure 9. Extension of constraints class.

that both compared fields have to be the same data type and also be comparable. Since the constraint has two attributes we decompose *compareTo* in an additional class as shown in figure 9. All the annotations are described in Table 1. and 2.

The abstract class *Constraints* is inherited by classes defining constraints for basic types.

Class for String type *CString* contains attributes *maxLength = 255, minLength = 0, pattern, JSPattern, link, html, textArea, password, email and creditcard.*(see Figure 10. and table 1.)

Class for Date type *CDate* contains attributes *past, future, pattern, JSPattern, date, time and timeStamp.* (see Figure 10., table 1. and 2.)

Class for Number types *CNumber* contains attributes *max, min, pattern, JSPattern and color.*(see Figure 11. table 1 and 2.)

TABLE I. ANNOTATION DETAILS (1)

Annotations		
Name	Usage	Applicabel
required	the field is required	*
unique	field is unique	*
ignore	ignore field in the view	*
formOrder	position in the view form	*
tableColumn	position in the view table	*
inputLength	preferred length of the view input element (default 30)	*
compareTo	forces its operation to be satisfied within a class	*.comparable
maxLength	maximum length of the data (default 255)	String
minLength	minimum length of the data (default 0)	String
pattern	Java pattern to be satisfied	String,Date,Number
JSPattern	additional pattern to be satisfied (Javascript)	String,Date,Number
link	field is used as a web link	String
html	field is used as a HTML data	String
textArea	field is used for long text	String
password	field is used as a password	String
email	field is used as an email	String
creditcard	field is used as an credit card	String

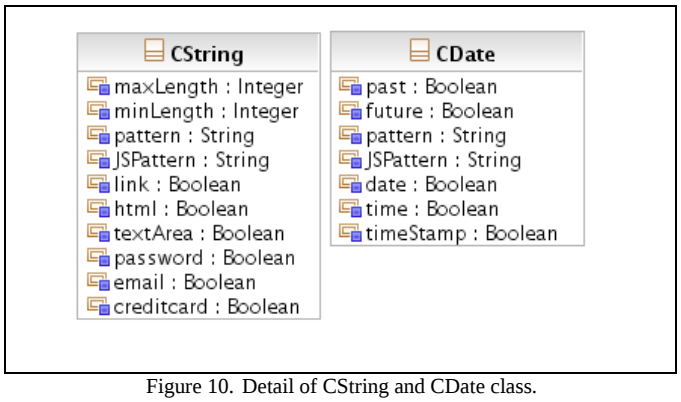

Figure 10. Detail of CString and CDate class.

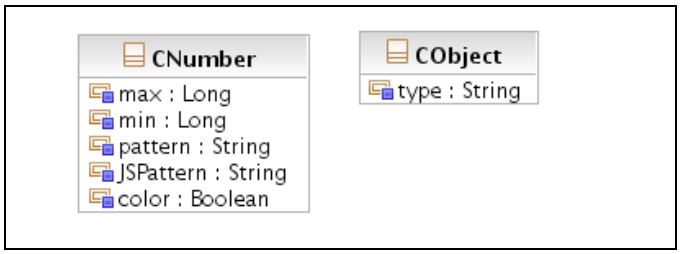

Figure 11. Detail of CNumber and CObject class.

TABLE II. ANNOTATION DETAILS (2)

Annotations		
Name	Usage	Applicable
past	date in the past from now	Date
future	date in the future from now	Date
date	date format "dd.mm.yyyy"	Date
time	date format "HH:mm:ss"	Date
timeStamp	concatenation of data and time	Date
max	maximum value of the field	Number
min	minimum value of the field	Number
color	used as a color picker	Number
type	type of a component for value selection	Object

Class for other Objects *CObject* contains attribute type (Selection, Option, Search ...). This class is also used for types Boolean and Enumeration. (see Figure 11. and table 2.).

Invariants for the extension should be also defined to avoid an integrity violation. Object Constraint Language (OCL) [1] is used to define invariants.

String type constraints should use only one of the following annotations:

```
context CString: inv
self.html->notEmpty() implies
self.link->isEmpty()
    and self.textArea->isEmpty()
    and self.password->isEmpty()
    and self.email->isEmpty()
```

For simplicity we define a function majority with threshold one `M1(arg...)`. It is a function that has `n` boolean inputs and one boolean output. Function has positive output if at most one input is true, otherwise false.

```
M1(true,true,false) = false,
M1(true,false) = true,
M1(false,false) = true


context CString: inv M1(
    self.html->notEmpty(),
    self.link->notEmpty(),
    self.textArea->notEmpty(),
    self.password->notEmpty(),
    self.email->notEmpty())
```

The *minLength* String constraint attribute should be a positive number.

```
context Cstring:
inv self.minLength >= 0
```

compareTo should contain only a valid class attribute with compatible type that is comparable and not a self reference:

```
Context Constraint: inv
self.compareTo->notEmpty() implies
self.Attribute.UMLClass.Attribute
->select(a:Attribute |
        a.type = self.type
    and a.name = self.compateTo.name
    and self.Attribute != a
)->size() = 1
and self.compateTo.operation
->isApplicableTo(self.type)
```

Majority 1 for *date*, *time* and *timeStamp*:

```
Context CDate: inv
M1(self.date->notEmpty(),
    self.time->notEmpty(),
    self.timeStamp->notEmpty())
```

We also need to make sure that Constraint types are applied to a correct data types.

```
context CString inv:
self->notEmpty() implies
self.UMLAttribute.type = "String"
```

```
context CDate inv:
self->notEmpty() implies
self.UMLAttribute.type = "Date"

context CNumber inv:
self->notEmpty() implies
self.UMLAttribute.type
->select(t:String |
      t = "Long"  or t = "Integer"
   or t = "Byte"  or t = "Short"
   or t = "Float" or t = "Double"
)->size() = 1
```

Now we have described all the information we need to hold in the model to provide a rich design model. We could find more information to hold in attributes and also more types, but this is left for a particular user and a particular area.

V. Example

At the beginning of this paper was provided a motivation example showing what view form we want to create. With our annotations we can generate entire view form without a manual intervention. The example is focused on view form generation. The entire process is described in [2]. Our design model from figure 1. is changed to a model using annotations in figure 12.

Considering this as a PIM model we transform it in Java web application using 3 PSM models as denoted in figure 13.

The first PSM is for SQL database. Here is transformed the PIM model in a binary model (similar to E-R scheme) called Relational-PSM. We use PIM attribute constraints for specifying PSM attribute integrity rules like `unique`, `not null` and so on.

The second PSM for EJB called EJB-PSM will extend our PIM attribute constraints. Figure 14. shows the model. In the figure the `Person` class is equivalent to the one from figure 12.

The third platform specific model is for the view. This model looks pretty much the same as the model for EJB.

After all PSM's are transformed we generate a code for each of them.

SQL code generation for a database is trivial and well described in [2]. We normally use SQL standard as a definition of SQL (2006), but in a particular cases we can generate database specific SQL code.

EJB generation is more complicated than the previous one. EJB here consists of two layers: entity layer (domain) and session layer (business). Domain layer contains entities that reflects the design model and contains all attribute constraints in form of annotations. Business layer consists of entity home objects and entity query objects (we expect Seam [20] as a framework). Entity home objects provide CRUD for given instance of an entity and entity queries provide selection for tables. Without going in a deep details we can choose to generate stateful or stateless (RESTful) applications. Our extension affects only entity beans and not session beans. The following code is an example of the `Person` entity showing its field `name`:

```
@Entity
@Table(name = "Person", catalog = "FormBuilder")
public class Person implements
                    java.io.Serializable {
...
private String name;
...
    @Column(name = "name",
          nullable = false, length = 100)
    @NotNull
    @Length(max = 100)
    @FormOrder(1)
public String getName() {
        return this.name;

}
public void setName(String firstName) {
        this.name = firstName;
}
...
```

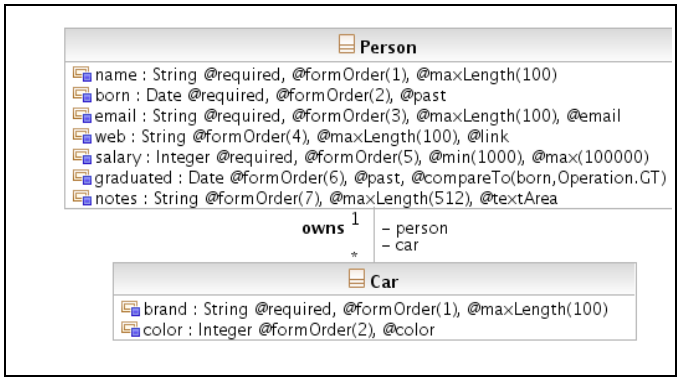

Figure 12. Example of rich design model

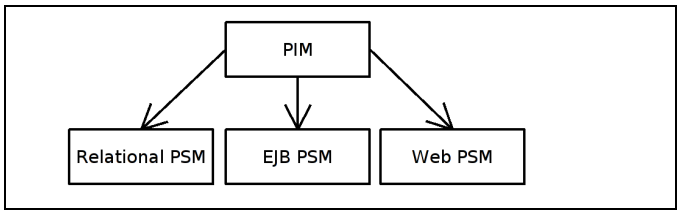

Figure 13. PIM to 3 PSM mapping

.

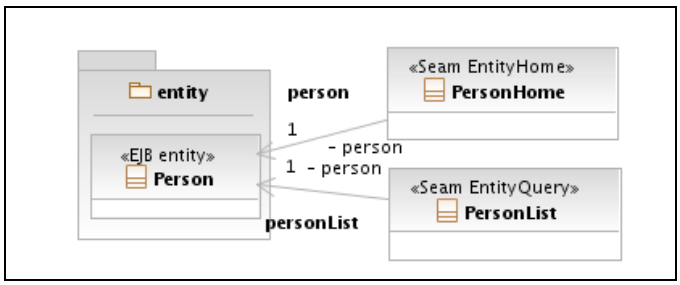

Figure 14. EJB-PSM

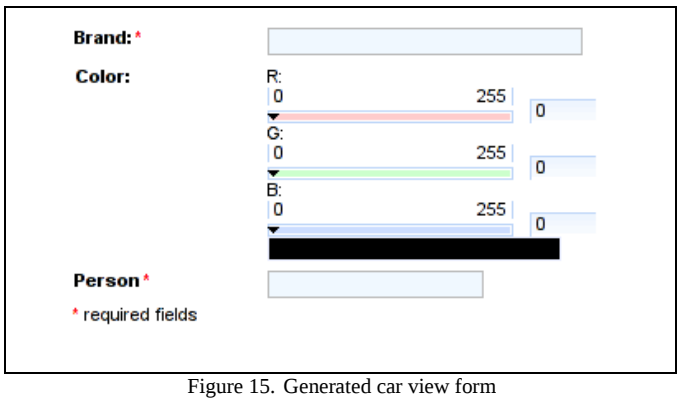

Figure 15. Generated car view form

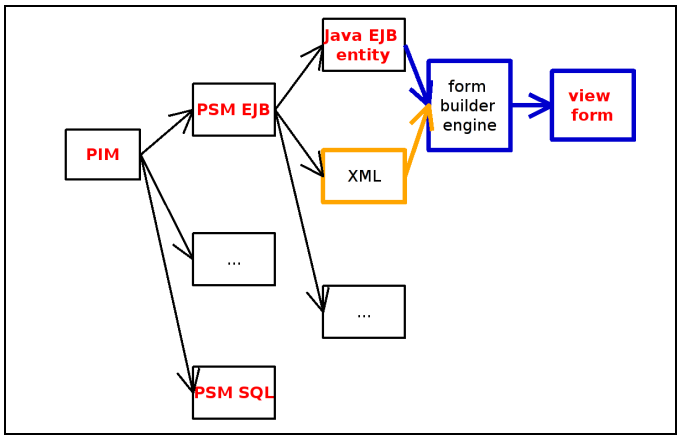

Figure 16. FormBuilder as an engine for view form generation

As the last step we generate the view [2]. We generate forms, tables, navigation and security. Our extension is for forms. They can be generated directly from Web-PSM, EJB-PSM or from EJB entity beans using a bridge [5,16]. The attribute constraints (annotations) are applied for all form elements. A view code for `Person` is shown on the following code:

```
<h:form>
 <util:inputText label="Name:"
                  edit="#{edit}"
                 value="#{bean.name}"
              required="true"
                  size="30"
             minlength="0"
             maxlength="100"
              rendered="#{empty
      nameRender ? 'true' : nameRender}"
                    id="name"/>
</h:form>
```

From the design model in figure 12. can be generated view form from figure 3. For the completeness is shown a generated view form for *Car* class in figure 15 (Please note the color picker which is type Integer).

VI. Related work

A work focusing on a visualization of design patterns in UML class diagrams [11] discusses and provides another solution to a class, attribute and operation annotations. Authors are extending UML by stereotypes.

An another publication [10] explores possibilities how to represent annotations in UML. Authors explores stereotypes as possible solution as well as definition as a normal modeling elements. The paper also denotes that stereotypes can model only a subset of modeling possibilities covered by the other solution. From our perspective the concrete implementation is not as important as the idea of rich design models.

Our previous paper [5] provides information from lower level perspective. Where we generate rich user interface from entity beans. Entity beans hold all the information for a view form generation and we call them rich entity beans. Providing developers with accessible templates for definition of their view input elements increases flexibility of the final view generating tool.

Many tools were introduced for a form generation. Some tools are graphical user interface wizards helping with form creation. Others reflect its XML configuration for a form building.

IBM XML Forms Generator [12] is a tool, in the form of eclipse plug-in, that can generate XForms [13] from given XML data instance. It can generate form elements that satisfy type and length constraints and control types according given XML scheme. Unfortunately XForm technology is not the only technology used for web forms. Our idea with rich design model goes well with this work. Our rich model can provide data in form of XML for XForm generation. The similar could be achieved with the tool from our previous work. The only module that is currently missing is a XML reader that would provide the same information as our class reflection method module. Figure 16. shows the missing module for our form builder (orange), it also shows how the form builder can be used for view form generation from MDA perspective.

Another idea can be found in IBM reflexive User Interface Builder (RIB) [14]. This tool is focused on standalone applications using Java Swing, AWT or SWT. RIB uses XML for describing Java GUI. Authors provide the a solution that reflects the year when their work was done. Similar approach was done by Java Server Faces. The thing is that at that time the latest version of Java was 1.4 . There were no annotations and generics. The new trend is trying to avid XML usage for external definitions because of it's weak type safety. Annotations are used instead. Out tool currently cannot generate the entire XML that RIB uses, but with usage of an appropriate templates we could generate parts of the XML script. The main difference with our tool is that we generate the forms from design model (or EJB entity beans) and not generate the forms from an external XML that is not a part of the design model. Although our tool could generate the input for RIB.

An example of form generation for scripting languages is GROK [15], which is a web application framework for python. GROK has the capability to generate forms from it's design model.

We have shown a lot of areas from distributed web applications, standalone applications to scripting languages where the from generation can be used. This fact is a good reason to consider rich form generation in MDA process.

VII. Conclusion

In this paper was proposed an extension for design models. We believe this extension allows to MDA languages to enable constraints definition and rich user interface generation. The proposal is influenced by distributed web applications but the idea is general and is useful also for standalone applications also.

Common practice in application development is to define an entity model and from there are manually defined view form. This is tedious, error-prone an mostly not necessary if few more information are captured in lower layer. The problem with manual development mostly comes with application maintenance when entity level is in scope of a backend developer and view level in scope of a frontend developer. Our approach actually eliminates the need of the frontend developer in the maintenance cycle for this task. Having the constraints set up also eliminates the need to manually enforce these in the business layer, this task can be simply automated by used platform. We use this approach successfully for more than one year in a large enterprise application and it is one motivation why we want to share this idea, which we believe strongly simplifies the application development and mostly maintenance.

This paper extends the idea from our previous paper FormBuilder[5] that proposed and introduced a tool for view form generation directly from EJB entity beans. Here we promoted the idea to the platform independent model of MDA. With our extension are design models capable to hold additional information for complete data validation and rich view generation.

We provide a tool that defines new constrains for EJB entity beans. Using the existing model information and new constraints the tool provides a configurable translation to view forms. The tool captures proposed idea and works as a code to code transformation bridge.

As a future work we plan to implement an XML module that will parse additional model information from XML files. These XML files would reflect rich design model. The advantage of this module is that we can easily export design models from modeling tools in an appropriate XML formats.

Acknowledgment

We would like thank to Michael Jeff Donahoo for good advises and comments. We would also like to thank to Bozena Mannova for good motivation.

References

[1] OMG, Unifiend Modeling Language (UML), Meta Object Facility (MOF), Model Driven Architecture MDA, Object Constraint Language (OCL), http://www.omg.org

[2] Anneke Kleppe, Jos Warmer and Wim Bast, MDA Explained, Addison-Wesley, Boston, February 2007

[3] Jos Warmer and Anneke Kleppe, The Object Constraint Language Second Edition, Addison-Wesley, Boston, August 2003

[4] Borivoj Melichar, Jazyky a preklady, CVUT, Praha, 2003

[5] Tomas Cerny, Michael J. Donahoo, Eunjee Song, “FormBuilder” CLI ICPC 2008, Banff

[6] Bashar Nuseibeh & Steve Easterbrook, “Requirements Engineering: A Roadmap”, ACM 2000, Future of Software Engineering Limerick Ireland

[7] Pamela Zave, Michael Jackson, “Four Dark Corners of Requirements Engineering” ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 1, January 1997, Pages 1–30.

[8] Kent Beck, Ron Crocker, Gerard Meszaros, James O. Coplien, Lutz Dominick, Frances Paulisch, John Vlissides, “ Industrial Experiences with Design Patterns”, Proceedings of ICSE-18, IEEE, 1996

[9] Nathalie Moreno, Jose Raul Romero, Antonio Vallecillo, Web Engineering: Modelling and Implementing Web Applications, chapter 12, An Overview Of Model-Driven Web Engineering and the Mda, Springer, London, 2008

[10] Vasian Cepa, Sven Kloppenburg, ”Representing Explicit Attributes in UML”. 7th International Workshop on Aspect-Oriented Modeling, Jamaica, 2005

[11] Jing Dong, Sheng Yang and Kang Zhang, “Visualizing Desing Patterns in Their Applications and Compositions”, IEEE Transactions on software Engineering, July 2007, Vol. 33, No. 7.

[12] Kevin E. Kelly, Jan Joseph Kratky, Steve Speicher, Keith Wells, Gee Chia, IBM XML Forms Generator, http://www.alphaworks.ibm.com/tech/xfg

[13] Xforms, standard W3C, http://www.w3.org/MarkUp/Forms/

[14] Barry Feigenbaum, Michael Squillace, IBM reflexive User Interface Builder, http://www-128.ibm.com/developerworks/java/library/j-rib/

[15] Dirceu Pereira Tiegs, GROK, http://grok.zope.org/documentation/how-to/automatic-form-generation

[16] Dariusz Gall, Michal Molenda, EDOC to EJB transformations within MDA, Blekigge Institute of Technology, Sweden

[17] Sam Thompson, Web 2.0 user interface technologies, IBM, 30 Jan 2007, www.ibm.com/developerworks/library/wa-web2ui.html

[18] Roadmap for Accessible Rich Internet Applications (WAI-ARIA Roadmap), 4 February 2008, www.w3.org/TR/wai-aria-roadmap/

[19] Ron Rogowski, The Business Case For Rich Internet Applications, Forrester Research, Inc, March 12, 2007 (Adobe)

[20] JBoss Seam, www.jboss.com/products/seam