A Profile Approach to Using UML Models for Rich Form Generation

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*—The Model Driven Development (MDD) has provided a new way of engineering today's rapidly changing requirements into the implementation. However, the development of user interface (UI) part of an application has not benefit much from MDD although today's UIs are complex software components and they play an essential role in the usability of an application. As one of the most common UI examples, consider view forms that are used for collecting data from the user. View forms are usually generated with a lot of manual efforts after the implementation. For example, in case of Java 2 Enterprise Edition (Java EE) web applications, developers create all view forms manually by referring to entity beans to determine the content of forms, but such manual creation is pretty tedious and certainly very much error-prone and makes the system maintenance difficult. One promise in MDD is that we can generate code from UML models. Existing design models in MDD, however, cannot provide all class attributes that are required to generate the practical code of UI fragments. In this paper, we propose a UML profile for view form generation as an extension of the object relational mapping (ORM) profile. A profile form of hibernate validator is also introduced to implement the practical view form generation that includes an user input validation.**

***Keywords*—Keywords: UML User Interface Modeling, Model Driven Development, Profile, Code Generation**

I. Introduction

The main idea behind the model-driven development (MDD) is that models, rather than code, should be the primary artifact where most of software development and maintenance activities are centered. Despite all its advantages, model-based user interface tools (e.g., [1], [3]) have not been widely used in practice [2]. In many cases, the user interfaces (UIs) are manually created by developers who are usually very much familiar with the current implementation. The manual development like this could work initially for small scale applications, but is not desirable when the system is under maintenance or should evolve for changes. The problem with manual UI creation becomes worse when the code fragments have been generated using a model-driven tool because manually added code makes the maintenance within MDA, which aims that any future changes in design also propagate to all generated fragments, difficult. Therefore, there is a need to integrate the required UI code fragment information within design models. Many UML tools that are currently available, however, do not support one to include properties necessary for the UI code fragment generation in the underlying design models.

For example, consider a UML design model shown in Fig. 2. We want to build a view form for a Java EE web application from it and each input should be validated when it is collected through the form. An example view form like one given in Fig. 2 can be generated for *Person* class using the information currently given in Fig. 1. However, such a form has the limited capability in detecting the meaning of fields and in validating user inputs because the given design model does not provide any form generation rules with validation details other than field types. Other problems in the current form generation can be summarized as follows. We might want to have some fields in a different order, e.g., the current date should be given before a person's birth date. An input to the email address field can be invalid, *http* link can be malformed, or there can be a negative value given to the salary filed. In addition, the current form includes the field like *id* that should be assigned by the system not by the user. Consequently this is not a view form that was expected. Therefore, we often generate an initial code for the form first and then add additional constraints and validation rules to each field in the form and reorder the fields. Fig. 3 shows a form that can be generated

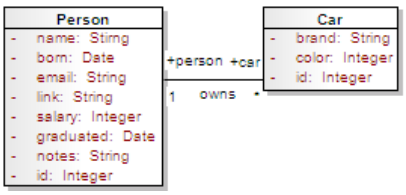

Fig. 1. A UML class model example

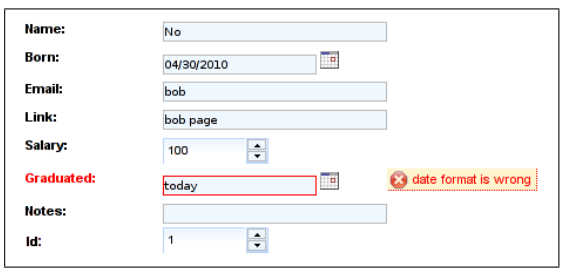

Fig. 2. A view form generated from a conventional design model (with validation)

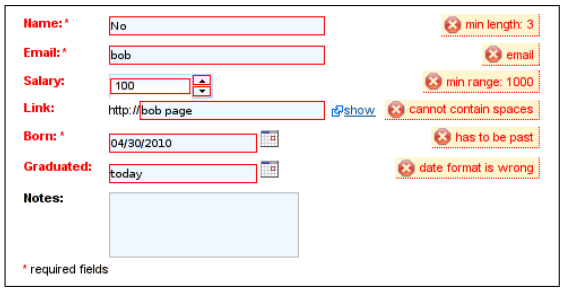

Fig. 3. A view form that we want to create (with validation)

by this additional modification. One of main problems in this kind of form generation process is that changes on the design model cannot be automatically applied to the form that was originally generated. Therefore, one needs to modify all related view forms and their underlying entities including database. To avoid this error-prone and tedious work, we propose a way to add the form generation details with validation rules to the design models so that they can be used to auto-generate the view forms with proper user input validation (UIV).

UIV is critical to application security and is required in every application interacting with users. Validation should happen at the server-side but for usability purposes is often needed at the client-side as well. Various validation libraries are available such as Hibernate Validator (HBV) [22]. This validator expects application to use Object Relational Mapping (ORM) and extends its capabilities. The use of ORM is a common practice in software development nowadays. Many ORM libraries like Open JPA [25], TopLink [26] or Hibernate [27] are available for its popularity. These ORM frameworks follow the Java Persistence API (JPA) standard [21]. ORM entities, although in lower layers, also determine the content of view forms where also UIV comes in place. Unfortunately there exists a gap between forms and entities where the data and validation rules are defined twice. One promising approach to reduce the gap brings form auto-generation [9] that requires to apply addition extension at the JPA entities .

In this paper we show an approach to bridging gaps between the ORM, UIV and form generation. We promote the code development trends to the model-driven development by defining UML profiles for ORM [20], UIV and form generation. These profiles implement model extensions and can be applied to standard UML design tools. Our goal is to generate user interface fragments[1] and automated UIV directly from design models using UML profiles and common development libraries[2]. A tool capable of rich form[3] generation is provided with a widget library for Java Server Faces (JSF). We believe to motivate code developers to use model-driven development receiving all its benefits of code generation [3].

Our paper is organized as follows: in Section II we look into the background and introduce model driven architecture. In Section III we provide a motivation of our approach introducing a tool we built for form generation. Section IV introduces UML profiles for class model extension to capture additional information and puts together the form generation tool with defined UML profiles. It is followed by a case study Section V that should enlighten the profile usage. The paper provides a broad discussion of related work in Section VI and conclusion.

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

[2]Such as JPA, HBV or FormBuilder (FB) [9]

[3]A form that integrates UIV

II. Background

Software development is complicated process influenced by internal policy, experiences, expectations or requirements on the software [10], [11]. Many software projects have a lot in common which was a motivation for creation of design Simplification with unification brought the Unified Modeling Language (UML) [5] which is popular and widely used language, that is taught in most universities around the world and every software engineer knows at least its subset. Language UML is under international, non-profit computer industry consortium Object Management Group (OMG) [5]. 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) [6].

In the past, a software development team used a middleware that was matching their project requirements. A problem with such a development direction is with product compatibility with various operating systems, with later compatibility with another middleware or just with customer's later requirements. MDA provides a solution by middleware independence that is achieved by the strength of UML which provides models that are platform independent and platform specific.

Models in MDA 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. An example for UML class diagram is that 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.

We mentioned that UML models are not capable to capture all information we need. There are constructions to use to make the models suitable for capturing all details we need, either we can use Constraints package in UML infrastructure or define UML profiles [5], [13]. Profile package contains mechanisms that allow extending meta-classes from existing meta-models. Profiles were defined for extending UML standard. UML profile is a package that has stereotypes as a specific meta-classes and tagged values meta-attributes. Stereotypes in UML reflect code annotations, and tagged values reflect annotation attributes. A majority of programming languages

has some form of annotations for additional information or constraints that apply in given contexts. Annotations were defined as a way from a large amount of XML configurations. Annotations in UML can be represented also as a normal modeling elements [14] that provide more modeling possibilities than stereotypes.

III. Motivation

Modern trends of application development go the direction from data model over the behavioral model to the user interface. First the database schema is defined, then a class model is built where we deal with ORM issues. The JPA is a standard that helps us with this mapping. It is a work of community process involving experts from various parts of industry dealing with ORM frameworks for Java platform. JPA annotations are relevant to the ORM and to persistence. Annotations are used for entities, their attributes, associations, inheritance, etc. Promotion of JPA to the model-driven approach was the goal of [20]. Once the mapping is finished the behavior comes in place. Many frameworks define controllers, managers or beans that deal with behavior. The presentation is the last step in the development process. It also means that presentation is dependent on the underlying code. If we focus on user interaction with a software application, we see that user invokes actions, and submits data. Data submission goes through forms and the input data must go through a validation. Validation is a standard process that must happen before the data persistence. One of the most popular frameworks for ORM - Hibernate provides a validation library HBV that takes care of data validation by annotating the data field. The similarity of annotations in JPA and HBV raises a question if we could integrate these somehow for view form generation and make view forms derive the information from entity attributes and its annotation constrains. We could, and it can make our forms more capable, because attributes will know what data inputs to expect. View forms, however, have more restrictions than what is already defined by JPA and HBV. For example we use a string field once as short input text, another time as text area, somewhere as password, html, or http link. We also need to derive proper order for the form elements.

Our previous work [9] focused on validating data and generating view forms from JPA entity using the HBV. We explored dependencies between entity and view form fields. An entity field partially determines the type of an input element used in a form. A proper view form input element is chosen based on the field type and its additional properties captured by annotations. These annotations are also used for field validation, for constraints and static business rules. A tool FormBuilder (FB) we provide is capable of generation of rich view forms that integrate all constrains defined in JPA entity. Currently the tool is used by a large enterprise application in production where every change to an entity is immediately reflected in the view form without programmer interaction.

Based on our experience with the tool we extend the idea to the design model. A design model using MD-JPA profile with MD-HBV profile and MD-FB profile can generate entity beans with validation and view annotations that can be applied to view form generation (and regeneration). Similar idea is also implemented in Naked Objects [19]. Considering distributed application frameworks like JSF, PHP, ASP etc. that contain very similar data forms, we believe this can propagate on other platforms. All the cross platform developers can benefit from transparency of view rich forms.

IV. Profiles

To flex class models and make them capable of gathering ORM, validation and view form specification information we use UML profiles. We build them incrementally so a developer can apply a subset. A popular MDA-based modeling tool Sparx System's Enterprise Architect [23] is used to define and apply these UML profiles.

A. ORM Profile

Implementation of ORM that uses JPA involves mostly annotations of classes, attributes and associations that are implemented as attributes. In the UML we use stereotypes and tagged values that extend UML meta-classes. Stereotypes can represent all the JPA annotations. Fig. 4[4] shows a subset of UML profile for JPA relevant to our goal. For full reference see [20]. Table I provides connection between annotations, stereotypes and their meanings.

Having these stereotypes in the model we know more about the meaning of each attribute. Also *entity* stereotype is important, because it reflects the entity name and enables the JPA persistent services. *Column* stereotype provides restrictions that specify how can be a given attribute used. This will significantly influence the upper layers of an application. Most of the attributes of a given entity are persistent but some of them may be *transient*. Sometimes we want to logically separate class in more peaces and for that reason we use *embedded* which physically embeds another class in the context class. Stereotypes for internal entity management

[4]Note that a full triangle arrow head should be used for stereotype metaclass extension according to the UML specification, but we use a stick arrow head instead because it is not supported by the Enterprise Architect.

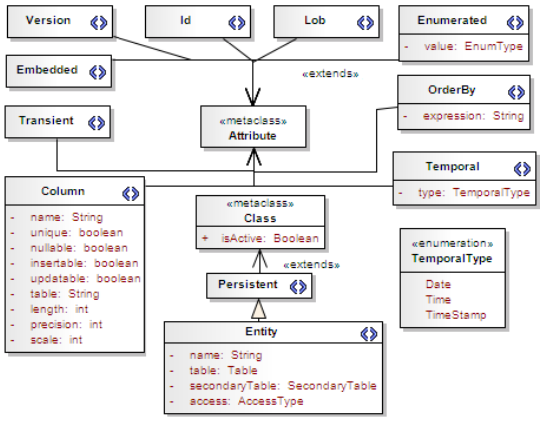

Fig. 4. UML profile for JPA - MD-JPA

TABLE I
MD-JPA STEREOTYPES DETAILS

Annotation	Stereotype	Description	Applicability
@Version	Version	Optimistic locking	Integer, Date
@Embedded	Embedded	Intrinsic part of owning entity	T extends Class
@Transient	Transient	Not persistent	*
@Id	Id	Primary key	*
@Lob	Lob	Binary data	Byte
@Enumerated	Enumerated	Persist ordinal or Strings	Enum
@OrderBy	OrderBy	Associated collection sort	Collection
@Temporal	Temporal	Database type Date, Time, TimeStamp	Date
@Column	Column	Matching table column	*
@Entity	Entity	Enable JPA for POJO	Class

are *id* that reflects the primary key and *version* that is used for optimistic locking. These internal management attributes are hidden to the application user. Associations are in JPA actual attributes that have specified mapping (one/many to one/many) when we need to order a collection of related classes we do so by annotating the collection by *orderby*. *Enumerations* are special type of data and we can map them by their ordinal or string value. To denote we use a binary data we use *lob* annotation. Specification of a date format is made by a tagged value of *temporal* stereotype.

If we look at these stereotypes from another perspective then besides ORM, they also provide meaning for each attribute. Some specify that attributes are internal (*version, id*), some indicate how to use them (*column, transient*) and some denote the specialization of an attribute (*lob, temporal, enumerated*). Could we use this information also for determination of how to design the human-computer communication? This could tell us that some attributes are not relevant to a user (*id, version*), some of them can be only displayed, but not modified (*transient, insertable, updatable*) and some are actually required in some special form (*nullable, unique, length*). This ORM information allows us to validate what user sends to the system before we try to persist. Considering usability practices this also influences the UI that prevents the user from actions that system denies. Our thinking must be correct because enterprise ORM framework Hibernate goes a similar direction with HBV that extends the JPA annotations for input validations.

B. Validation Profile

HBV is based on defined metadata model for JavaBean validation (JSR 303). Its usage is not much different from the previous JPA. Annotations are used for entity attributes to specify additional constraints. We implement the MD-HBV profile that allows us to specify these constrains in design model. Fig. 5 provides the pallet of available stereotypes. All of these apply on attributes. Table II provides connection between annotations, stereotypes, meanings and applicability.

Validation stereotypes interfere a little bit with the JPA stereotypes and tagged values (*NotNull* vs. *nullable* tagged value in JPA, *Length* vs. *length* tagged value in JPA), this is a result of new specification that is delayed in JPA 2. Validation annotations give us very precise constrains on applied attributes. These determine a lot of meaning of each attribute. We use *length* to specify string length. For numbers we can set *range* they are expected to be in when we can set just a *min* or *max* value. For collections and arrays we may set expected *size*. For string properties we may expect *email*, *credit card number* or 13-digit *EAN* codes (or *UPC-A*), we may also need to set a specific *pattern*. Some money attributes will need a validation for *digit* amount. Dates may need to be set for *future* or *past* (for DOB). All fields might be checked for being *not null* or *not empty*.

Having specified JPA entity using JPA and HBV annotations we look at a system user. He sends data through view forms. If the forms are plain and validation fails, user must re-submit his data. From the usability perspective we rather design view

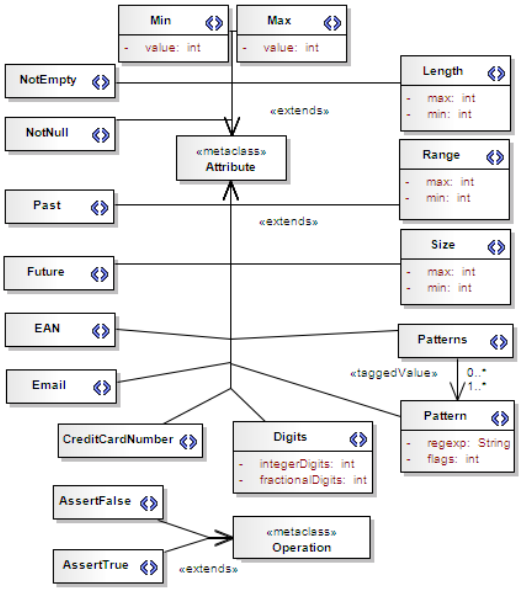

Fig. 5. UML profile for Hibernate Validator - MD-HBV

TABLE II
MD-HBV STEREOTYPES DETAILS

Annotation	Stereotype	Description	Applicability
@Min	Min	Value higher or equal to min	Num/String
@Max	Max	Value less or equal to max	Num/String
@Length	Length	Value length in the range	String
@Range	Range	Value between min and max	Num/String
@Size	Size	Value size is between min and max	Collection, Map, Array
@Pattern	Pattern	Value matches the reg-exp	String
@Patterns	Patterns	Value matches the reg-exps	String
@Digits	Digits	Number with up to specified integer, fractional digits	Numeric, String
@CreditCard-Number	CreditCard-Number	Match credit card number	String
@Email	Email	Match email	String
@EAN	EAN	EAN (13) or UPC-A code	String
@Future	Future	Future date	Date
@Past	Past	Past date	Date
@NotNull	NotNull	Not null attribute value	*
@NotEmpty	NotEmpty	Not empty attribute value	*
@AssertFalse	AssertFalse	Operation must return true	Boolean op.
@AssertTrue	AssertTrue	Operation must return false	Boolean op.

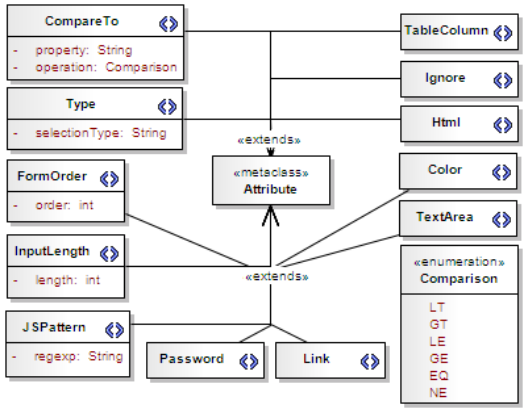

Fig. 6. UML profile for FormBuilder - MD-FB

TABLE III
MD-FB STEREOTYPES DETAILS

Annotation	Stereotype	Description	Applicability
@FormOrder	FormOrder	Order in view form	*
@InputLength	InputLength	Input widget length	*
@JSPattern	JSPattern	Java Script regular expr	String
@Password	Password	Password expected	String
@Link	Link	Link expected	String
@TextArea	TextArea	Long text expected	String
@Color	Color	Color expected	Integer
@Html	Html	Html expected	String
@Ignore	Ignore	Do not generate field	*
@TableColumn	TableColumn	Will be used for table	*
@CompareTo	CompareTo	Must satisfy comparison	*
@Type	Type	Type of widget to use	T extend Class

forms based on entity attribute constraints. Stereotypes that hold the constraints determine what data can be persisted and which are wrong. To direct our user, and prevent him from wrong data submissions, we must reflect all the attribute stereotypes in the forms. Doing so manually is tedious replication, so automation is the correct direction. But our plans halt when we want to generate forms with defined order (reflective API does not warrant attribute order), or when we want use a string attribute to set a password, use it as http link or render it as HTML. What if some integer value has a meaning of a color, etc.? We could generate a form and modify it manually, but if a change to the JPA entity comes we must to start over. What is necessary to auto-generate forms that will have all we need? We explore these in the following subsection.

C. Form Builder Profile

We explored and discussed common practices for ORM and input validation that are not sufficient for rich form generation. In this section we identify the missing elements for view form auto-generation and future re-generation. The MD-FB profile provides a way to capture all necessary information in design model. Fig. 6 provides the variety of MD-FB stereotypes. Table III provides connection between annotations, stereotypes and its meaning with applicability.

The first thing that is missing in the previous profiles is *form order*, i.e., Java Reflective API does not warrant class field order. This way we can set the form order we expect. Input fields are capable of restriction on data size but there is also the visual size of the element (*input length*). Some attributes in the entity might be *ignored* for the generation because they are responsible for internal logic. String attributes may be distinguished for *passwords*, *text areas*, *html*, *http links* or default. Sometimes we interfere Java regular expressions with JavaScript ones but sometimes not (*JSPattern*). Some complex attribute types will need a component that allows selection, just imagine select menu for *country*, option box for *gender* or suggestion box for *person*, for all of these we set *type* to a string key that will be mapped to a particular view widget. Rarely we want to set a *color*. We also extend the validation to attribute *comparison* satisfaction. This will be useful for flight booking where we set two dates one for departure and one for return, the departure date must be before the return one. Some attributes might be used for table generation (*table column*).

D. Constraints for Model Verification

The defined profiles bring the advantage to verify validity of the design model. In order to verify models we specify a set of constraints using Object Constraint Language (OCL) [5]. OCL language defines invariants for stereotypes, which must be true for all elements. These invariants can be verified in integrated development environment that supports OCL interpreter. Many OCL constraints for MD-JPA have been defined in [20]. For structural description, we also consider [7], [8]. Most of stereotypes are applicable to specific attribute types. We verify these by applying constrains inside each stereotype. For instance for the MD-FB *Password* stereotype we specify OCL in Listing 1. Similarly we apply these for many other stereotypes and particular types. For MD-HBV we also specify constraints for *Length*, maximal and minimal *Range*, *Size*, etc. Specifically for *Asserts* we need to check the return values. For evaluation that *CompareTo* property exists we can use the metamodel. For code implementations it is possible to have not synchronized JPA and HBV annotations like *Length* and *column* attribute *length* or *NotNull* and *column nullable*. In here we can force that they are in sync.

```
context Password: inv self.type.name = 'String'
context Length: inv self.max >= self.min
  and self.min >= 0 and self.type.name = 'String'
context Size: inv self.max >= self.min and self.min >= 0
  and self.type.oclIsKindOf(CollectionType)
context AssertTrue: inv
  self.getReturnParameter().type.name = 'Boolean'
context CompareTo: inv self.type = self.property.type
  and self.Class.attributes.exists(self.property)
```

Listing 1. OCL contraints

E. Proposed and Implemented UML Profiles

We have proposed UML profiles for conventions libraries for ORM, validation and view form generation. These profiles are implemented and tested by Enterprise Architect tool where we can import these profiles. At this point the profiles are incrementally dependent, although we can apply only MD-JPA, or add MD-HBV and on the top we could add MD-FB.

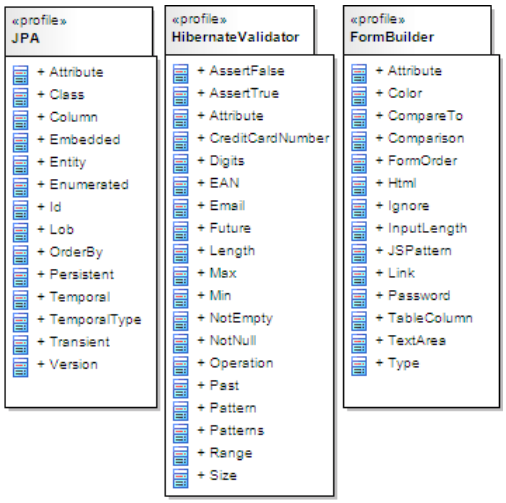

Fig. 7. UML profiles ORM, Validation and FormBuilder summary

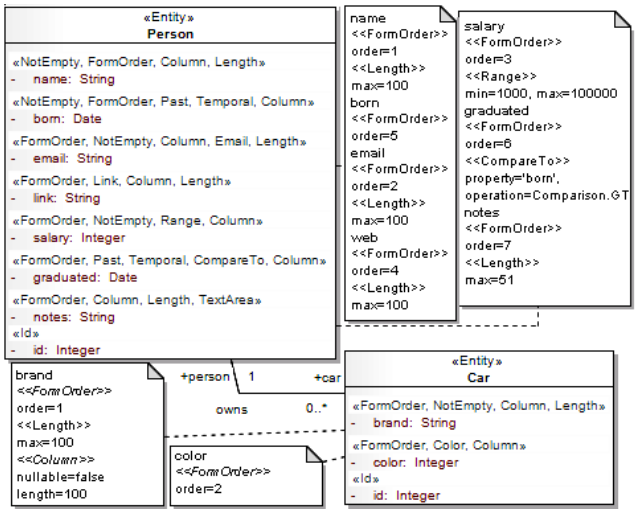

Fig. 8. Example of rich design model

We believe that these profiles will fill the gap between code programmers and model-driven development. Programmer can build models that fully reflect what he could write manually, the advantage is the level of abstraction and its independence. For instance Hibernate exists also on .NET so this community can gain from our profiles as well. The principle is platform independent and not limited to web applications. Summary of the implemented profiles is shown in Fig. 7.

F. Profiles and FormBuilder

We could use only ORM profile for its purposes, but most probably we will need to do validation and build view forms. We already mentioned the disadvantages of manual approach. Here we look at automation with no complicated learning curve. Using the defined profiles we completely describe system's static structure and from the class model generate JPA entities with additional annotations. The tool we provide takes the generated entities as an input. It also allows developer to set component fragments that are decorated with the actual entity attributes and their constraints. We provide this tool with a complete view widget library for JSF applications using components from RichFaces, ICEfaces or JSF. Developer can decide which library to use, modify the widgets or define his own. The tool has a mapping of entity attributes to each widget so the developer can modify the mapping. By default he adds the tool with default configuration to his project and calls the generation on all entities. This generates him rich view forms reflecting JPA entity beans and their constraints. Every-time a change to an entity comes, the forms are re-generated. The widget library also implements client-side validation which provides a fast validation to user data.

V. Case Study

At the beginning of this paper was provided a motivation example showing what rich form we want to generate. With the UML profiles, provided for design models, we can generate the code fragments without any manual intervention. The example is focused on form generation. Our design model from Fig. 1 is changed to a model using annotations in Fig. 8. We start to build on the same model and stereotype the model using MD-JPA for ORM, this enable us to generate code that will use a JPA implementation framework (Hibernate, OpenJPA or TopLink). On the top we add MD-HBV for validation, this allows to generate code that will use HBV library. As next we add profile for MD-FB and add stereotypes for the view form generation. The generated code captures all necessary details. Once we are done with entity code generation, the FormBuilder comes in place. We add it view widgets library and point FormBuilder at chosen entities.

Fig. 8 shows the design model with applied profiles. The *Person* class is equivalent to the one from Fig. 1 The *Person* entity code snippet is in Listing 2.

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

@Column(name = "born", nullable = false)
@Temporal(TemporalType.DATE)
@NotEmpty
@Past
@FormOrder(5)
public String getBorn() {return this.born;}
...
```

Listing 2. Person entity fragment

Using JPA we generate SQL scheme, we can also use business layer generators like Seam-gen [18]. The next step is to generate view like forms, tables, navigation and security. View generation process is well described from the behavioral perspective for the J2EE platform by [4], [17] or could also

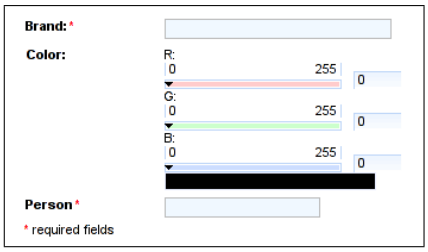

Fig. 9. Generated car view form

be made by [18]. The improvement comes for view forms. FormBuilder refers to the entity and for each field is selected a view widget based on field type and annotations that are also used as parameters for the widget. FormBuilder adds full control over widgets their modification. A view code snippet for *Person* is in Listing 3. All the underlined texts are supplied by FormBuilder based on the field information. The design model in Fig. 8 captures all information to generate the form in Fig. 3. The *Car* form in Fig. 9 is shown for completeness.

```
<h:form id="formPerson">
 <util:inputText label="Name"
                  edit="#{edit}"
                 value="#{bean.name}"
              required="true"
                  size="30"
             minlength="0"
             maxlength="100"
                 title="#{text[t.person.name]}"
              rendered="#{empty nameRender
                          ? 'true' : nameRender}"
                    id="#{prefix}name"/>
 ..   <!-- other elements 2, 3, 4 -->
 <util:inputDate label="Born"
                  edit="#{edit}"
                 value="#{bean.born}"
              required="true"
                 title="#{text[t.person.born]}"
              rendered="#{empty bornRender
                          ? 'true' : bornRender}"
                    id="#{prefix}born"/>
 ..   <!-- other elements 5, 6 -->
</h:form>
```

Listing 3. Person view form code

The advantages of the auto-generation were mentioned many times. Human designer will appreciate it the most once a change request for the entity structure or field signatures comes. The only thing to do will be to modify the model and auto-generate the rest of code. No type errors can happen so the code will work perfect on the first run.

Using the UML profiles on design model for Humam Compter Interaction (HCI) we have two options. When user has an intention to modify some data, we can auto-generate a rich form specific to his request in run-time. So the form generation happens on demand. This will with no doubt simplify the whole process, but on the other hand decrease its performance. We can pre-generate rich forms for all application users (as done in the example) and fetch them statically instead. Static fetch improves the performance.

Another advantage comes when the targeted application communicates over HTTP, WML or with standalone clients. The idea of having a rich model that renders a view in run-time based on the client capabilities also adds benefits. We can also extend the previous using role-based access control. Some user roles can see more information and some less, generating the view can also consider visibility constraints when forms are requested. This idea could be elaborated in similar manner, but it is over the scope of this paper. In this study we dealt with aspects for ORM, validation and view form generation. A modeling tool that has a profile view for each particular aspect would be very useful. Similar feature for dealing with crosscutting concerns would help in integrated development environment (IDE) for code development.

VI. Related work

Many publications related to user interface generation exist [2], [24]. The idea for user interface generation using an MDA approach [1], [3], [4] is not new, neither.

We first look at survey of Model Driven Engineering MDE tools for HCI [1]. MDE developer will benefit from our profiles by capturing more information for each class attribute. This clarifies the meaning of a particular attribute and enbables to generate a better application. Such an improvement helps to avoid unsuccessful form submissions when user provides wrong data. Based on the fine-grained information we can provide user with more usable solutions and better guidance what data we expect. Authors believe in the need for MDE tools for user interface design. We propose and implement profiles that can be applied to existing UML tools so that one can capture more view form specific information.

Authors of [2] mention that despite a lot of research, model-based UI tools have not become common, in part because building models is an abstract process and better results are often achievable by a human designer in less time. We have illustrated significant improvements to this. Using our profiles we can generate view forms with UIV. We provide a tool for J2EE that is successfully used in a large enterprise application for almost two years to simplify its maintenance. This open-source tool experienced more than 700 downloads.

Investigation of the use of MDA for developing HCIs is the goal of [3]. Authors discuss the gap between HCI and System Engineering and mention that one may argue that relating design models to the user interface can be considered as a mix of presentation and persistence logic. Authors, however, point out that HCI concerns cannot easily be described independently from other concerns for a system. Our paper builds on this.

The importance of UI auto-generation from PIM to UI code in MDA is mentioned in [4]. Behavioral diagrams are used for transformations in this paper. Our idea extends the richness of the generated application, from the data input perspective. We have explored some ideas from the paper in more details.

Using MD-JPA[20] for ORM mapping is a correct way to fill the gap between model-driven approach and manual development. Our research goes beyond the UML profile use for ORM, provides profiles for validation and form generation and brings the connection of these aspects.

Most of the related work focuses on complete application generation rather than code fragments that can be optimized for a practical use. For a system developer, such a *full* MDA tool generation could be very complex and the result he gets might be different from what he expects. In this paper we focused rather at improvements of fragments from the whole process which we believe can help practically in projects. We provide UML profiles that extend existing UML diagrams rather than inventing new diagrams as used in [24]

Our previous work [9] provides the insight from the lower level perspective. We looked in details on entities, view widget library, mapping and form generaration introducing our tool.

Many tools were introduced for the form generation. Some tools reflect its XML configuration for a form building. IBM XML Forms Generator [15] is a tool, in the form of eclipse plug-in, that can generate XForms [16] 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 MD-FB profile and FormBuilder goes well with this work. FormBuilder allow to build own library of widgets, these can be in form of XML for XForms.

VII. Conclusion

In this paper we proposed and implemented UML extension for design models in the form of UML profile which is a set of stereotypes and tag values. We discussed the connection between ORM, validation and form generation that all are driven by design model. We believe that this extension brings a way to fill a gap between MDA and manual code development. We have also shown that using few more stereotypes over ORM allows us to generate view forms that fully reflect constrains placed on the persistent entities. Our work is influenced by distributed web applications but the idea is general and is useful for standalone applications as well.

A common practice in application development is to define an entity model and from there manually code the view forms. This is tedious, error-prone an mostly not necessary if few more information are captured in the 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 using the proposed platform. We use this approach successfully for almost two years in a large enterprise application and it is one motivation that made us want to share this idea, which we believe simplifies the application development and maintenance.

This paper extends the idea from our previous work FormBuilder [9] that proposed and introduced a tool for view form generation directly from JPA entity beans. Here we have promoted the concept as a UML profile. With our extension, UML models can be capable of holding the additional information for complete data validation and rich view form generation. We provide a tool that defines new constrains for JPA entity beans. Using the existing model information and a few new constraints, the tool provides a configurable translation from entities to view forms.

References

[1] A Survey of Model Driven Engineering Tools for User Interface Design, Jorge-Luis Perez-Medina, Sophie Dupuy-Chessa, and Agnes Front , Laboratory of Informatics of Grenoble , TAMODIA 2007, LNCS 4849, pp. 84-97, 2007. Springer-Verlag Berlin Heidelberg 2007

[2] Automatic Interface Generation and Future User Interface Tools, Jeffrey Nichols, Andrew Faulring, Human-Computer Interaction Institute, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213 USA

[3] Investigating User Interface Engineering in the Model Driven Architecture, Jacob W Jespersen, Jesper Linvald, IT-University of Copenhagen, Glentevej 67, 2400 NV, Denmark

[4] An Extended MDA Method for User Interface Modeling and Transformation, Wu J-H, Shin S-S, Chien J-L, Chao WS, Hsieh M-C, Fifteenth European Conference on Information Systems, 2007

[5] OMG, Unifiend Modeling Language (UML) Version 2.1.2, Meta Object Facility (MOF), Model Driven Architecture (MDA), Object Constraint Language (OCL) Version 2.0, http://www.omg.org

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

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

[8] On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages, Dimitrios S. Kolovos, Richard F. Paige, Fiona A. C. Polack, Department of Computer Science, University of York, UK

[9] FormBuilder, Tomas Cerny, Michael J. Donahoo, Eunjee Song, CLI ICPC 2008, Banff, http://sourceforge.net/projects/form-builder/

[10] Requirements Engineering: A Roadmap Bashar Nuseibeh, Steve Easterbrook, ACM 2000, Future of Software Engineering Limerick Ireland

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

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

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

[14] Representing Explicit Attributes in UML, Vasian Cepa, Sven Kloppenburg 7th Int'l Workshop on Aspect-Oriented Modeling, Jamaica, 2005

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

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

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

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

[19] Naked-objects, development platform www.nakedobjects.org

[20] Towards a UML profile for model-driven object-relational mapping, Alexandre Torres, Renata Galante and Marcelo S. Pimenta, Instituto de Informatica Universidade Federal do Rio Grande do Sul, Brazil, 2009 XXIII Brazilian Symposium on Software Engineering

[21] JSR 220: Enterprise JavaBeansTM. Version 3.0. Java Persistence API, Linda DeMichiel and Michael Keith, 2006. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

[22] Hibernate Validator, open source validation library for Hibernate framwework https://www.hibernate.org/412.html

[23] Enterprise Architect, UML design tool, www.sparxsystems.com.au

[24] An extension of UML for the modeling of WIMP user interfaces, Jesus M. Almendros-Jimenez, Luis Iribarne, Information Systems Group, University of Almeria Spain, Journal of Visual Languages and Computing 19 (2008) 695-720, 13 December 2007

[25] Apache OpenJPA, openjpa.apache.org

[26] Oracle TopLink, www.oracle.com/technology/products/ias/toplink/jpa/

[27] JBoss Hibernate, www.hibernate.org