**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 the user interface (UI) part of an application has not benefited much from MDD although today’s UIs are complex software components and play an essential role in the usability of an application. It is a common practice that developers create view forms manually by referring to entity beans to determine their content. However, such kind of manual creation is very error-prone and thus makes the system maintenance difficult. One promise in MDD is that we can generate code from UML models, but existing design models in MDD does not capture enough information that are required to generate desired UI fragments. This paper presents our approach addressing these issues. The approach makes it possible to generate complex UIs, rich view forms, that fully satisfy both designers and end-users and to enforce system access control.

**Categories and Subject Descriptors**

D.2.2 [Software Engineering]: Design Tools and Techniques

**General Terms**

Design

**Keywords**

UML User Interface Modeling; Model-Driven Software Development; User Input Validation; Field Access Control

1. **INTRODUCTION**

Recent advances in software development and information processing technologies have led to attempts to build more complex software systems. The complexity of such systems has highlighted the inadequacies of the abstractions provided by modern high-level programming languages [10]. The Model Driven Architecture (MDA) (www.omg.org/mda) is an initiative that was proposed by the Object Management Group (OMG) to deal with such kind of issues. The MDA initiative leads to the Model-driven development (MDD) that is a style of software development where the primary software artifacts are models from which code and other artifacts are generated according to best practices. Despite many advantages in the MDD, however, model-based user interface development tools (e.g., [12], [22]) have not been widely used in practice [20]. In many cases, user interfaces (UIs) are manually created by developers who are usually very familiar with the current implementation. This way of UI development can work for small scale applications initially, but is not desirable even for a small system development when it should evolve. The problem with manual UI creation becomes worse when the code fragments have been generated using a model-driven tool. For example, manually added code makes the maintenance difficult because MDD aims to have all changes in generated fragments propagated from future changes in design. 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 the properties necessary for the UI code fragment generation in the underlying models.

For example, consider a UML design model shown in Fig. 1 from which we want to build a view form for a Java EE web application. In addition, each field should be validated when it is collected through the form. An example view form like one given in Fig. 2 can be generated for the `Person` class using the information currently given in Fig. 1. However, such a form has limited capability in detecting the meaning of fields and in user input validation (UIV), because the given design model does not provide any form generation rules with validation details other than field types. Another problems in the form can be summarized as follows. We might want to have fields in a different order, restrict that the provided birth date should be in the past, an input to
the email address field could be invalid, http link can be malformed, or there can be a negative value given to the salary field. In addition, the current form includes the field 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 by this additional modification and integrates UIV.

Often we need our application to work for multiple users who may have different access rights or different user roles in the system [9]. This may also impact the forms that are provided by the system. An user with full access rights may see the form in Fig. 3 for a person selection, but a user with restricted rights would be allowed to see just a subset of this form, such as the one in Fig. 4. In order to provide both forms in the system may require either a duplication or to embed a presentation logic for each field in the form.

One of main problems in this kind of form generation process is that later 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 the database. If we decide to change restrictions for user rights, we must again manually modify the original forms, which is an error-prone and tedious work. In this paper we show an approach that makes it possible to generate complex UI fragments and provide a tool capable of such generation. Our focus is to fragments where user data play the main role. In addition to form and table generation we consider UIV and the impact of access control for form generation is elaborated in Section 5 with evaluation of multiple approaches. Later, in Section 6, we provide our production experience. Related work is discussed in Section 7 and conclusion is provided in Section 8.

2. BACKGROUND

Complicated process of software development is influenced by internal policies, experiences, expectations or requirements on the product. Different software projects, in fact, often deal with similar challenges, these similarities can be found in development phases, approaches, technologies, architectures, code fragments, etc. Having similar problems when dealing with software development led developers to design frameworks and design patterns. The market today provides large amount of technologies that can be used. Model Driven Architecture (MDA) [18] may bridge multiple technologies and application frameworks. The rational behind MDA suggests that rather than to develop applications in the conventional way, the models should be used as a starting point and the middleware source code generated from those models. This has an advantage in middleware independence. MDA is not the only way models can be used for the MDD. Generally MDA is the way OMG recommends MDD, but multiple other solutions for domain specific languages can be applied.

In this paper we build on the top of MDA. This has the benefit that all extensions we propose are compatible with CASE tools that support UML and MDA. In the Section 1 we mentioned the need to extend UML class model in order to capture additional constraints. UML models are syntactically and semantically described by meta-models. UML meta-model then defines all the properties and characteristics for every model language element. In order to extend UML diagram functionality an extension mechanism called UML profile is provided [19]. An UML profile is a package that has stereotypes for specific meta-classes and tagged values for meta-attributes.

Using UML models in MDA development allows code generation of SQL database schema, platform class model, business logic, navigation, behavior and basic user interface [18]. To address MDD class model UI visualization with desired output, we must look at the system design first. Most of contemporary systems use object oriented design. In this design an entity (domain object) represents a persistent state [11]. This state of an entity is represented either through persistent fields or persistent properties. Many development platforms, such as J2EE, apply object-relational mapping (ORM) on its entities to connect their persistent state and relational databases. In the case of the J2EE the standard JPA [8] provides annotations that are relevant to the ORM.

---

1 http://www.omg.org/cgi-bin/doc?formal/09-02-04
Annotatations are used for classes, their attributes, associations, inheritance, etc. An approach to persistence modeling [24] promotes JPA by an UML profile to UML models. This then allows us to use the model for MDD.

Our attention in this paper is the presentation layer which is influenced by the data model. Data model is visualized in UI mostly by forms and tables. Through forms we submit and modify data, but those must go across validation first before updating the entity object and the database. UIV integration in forms is critical for application security and is required in every application interacting with users. Common approach in application development [1] [8] is to apply ORM rules in the entity, extend these with rules for validation and field access control [3]. Entity seems to be a good location to apply multiple aspects. In the common approach we see that entity captures aspects of ORM, validation and security. An entity, although it does in a lower layer, partially determines the content of its view form. In [6] [16] [21] authors suggest that to fully determine the form content, an entity may apply an addition aspect of form generation. In order to promote ORM aspect to models, [24] provides the MD-ORM UML profile. Similar to the ORM aspect also validation and form generation aspects can be captured by UML profiles [7] and used for model-driven approaches.

In this paper we elaborate the model-driven approach in more detail and show its application for table generation as well. Furthermore, our extension includes an approach to complying with access control for form fields.

### 3. UML PROFILES

To make UML class models capable of ORM, UIV and Form generation, three UML profiles are defined [7]. From the ORM meta-model defined by JPA standard [8] all rules can be transformed to UML stereotypes and tagged values to define the MD-ORM profile [24]. An existing meta-data model for JavaBean validation (JSR 303) [8] is applied to define the MD-Validation profile. MD-Form profile then addresses the missing elements to fully determine the form content and follows the meta-model defined in [6]. Table 1 provides a subset of stereotypes from those profiles, provides their description and applicability to domain types. Fig. 5 shows an overview of profile implementations. These profiles describe the system structure fully from all three aspects and allow us to fully determine view form content.

In the Section 1, we have shown that a form may be used in various mutations for users with different access privileges. To support such an access control, we apply the Role-based Access Control (RBAC) security model [9]. Fig. 6 shows RBAC model where a user is assigned permissions to call operations and access objects, although he is not assigned these permissions directly but through access roles that are shared among user groups. A role is directly associated with multiple permissions, where a permission can be shared among roles. The model allows us to define role hierarchy and associate roles with user sessions. Roles simplify permission assignment to users as instead of associating a user with a permission the user is assigned roles. System administrator can pre-prepare roles that will be used throughout the system, a new user is assigned to have certain roles and restrictions in the system, in addition user can change roles during the life time. Regards the form generation, permissions apply on form fields. A field is rendered when the user has a role granting access to the given field. In our approach
we allow a user to have multiple roles in a given session. The
user is provided with a form containing a particular field, if
he has at least one role that allows access to the field.

To deal with different fields rendered in the form for differ-
ent users, we may apply multiple approaches. One possible
approach is to design forms where each field can be selec-
tively rendered based on supplied conditions that involve an
additional presentation logic. To avoid this addition in the
presentation layer, we suggest that the design model cap-
tures these field access restrictions. We define a stereotype
FormRoles with a tagged value specifying roles that has ac-
cess to the field. Until now it was possible to generate one
form that suited all the requirements, but for the field access
control with logic kept in the model we must either generate
multiple forms for different contexts of their use or generate
them in the run-time as the user rights are evaluated. These
strategies are discussed and evaluated in Section 5.

3.1 Model Verification
In order to verify models, UML profiles specify a set of con-
straints using Object Constraint Language (OCL) [24]. This
language defines invariants that can be applied to stereo-
types. These invariants can be verified in an integrated de-
velopment environment that supports an OCL interpreter. Listing 1 shows selected OCL constraints.

```java
@Entity @Table(name = "Person")
public class Person implements Serializable {
    private String name;
    private Date born;
    ...
    @Column(name="name", nullable=false, length=100)
    @NotEmpty
    @Length(max = 100)
    @FormOrder(1)
    @FormTableColumn
    public String getName() { return this.name; }
    ...
    /* get/set */
    @Column(name = "born", nullable = false)
    @Temporal(TemporalType.DATE)
    @NotEmpty
    @Past
    @FormOrder(5)
    public Date getBorn() { return this.born; }
    ...
    /* get/set */
```

Listing 2: Person entity fragment

Our tool generates UI fragments for provided entities. It
inspects entity attributes and their properties and based
on that selects an appropriate UI widget. Furthermore, it
propagates all constrains and validation settings to the widget
so that UIV can be applied. The designer can define mapping
between the widget and entity field properties and also
design his own library of widgets to allow full customization.
A generated form example for Person entity integrating UIV
and constraint restrictions is shown in Listing 3.

```xml
<h:form id="formPerson">
    <util:inputText label="Name"
        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"
        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

4. MODEL-DRIVEN FRAGMENT GENER-
ATION: AN EXAMPLE

With the UML profiles described earlier, complex UI frag-
ments can be generated. The class model from our exam-
ple in Fig. 1 can be extended using the profiles and used
for ORM, validation, form generation and access control. Fig. 7 shows the design model with applied profiles, the
tagged values are shown in a note. The Person entity gen-
erated from the model contains all information as annotations
which matches manual development approach. The Person
entity, in Listing 2, is equivalent to the one from Fig. 1 with
the addition of the profile stereotypes.

Figure 7: Example of rich design model
All the underlined texts are supplied by the tool based on the attribute inspection. The design model in Fig. 7 captures all the information required to generate the targeted form in Fig. 3. The Car form in Fig. 8 is generated in the same way providing an example of a more complex attribute Person.

Collection UI fragments such as tables can also be generated. Tables are in fact not much different from forms in terms of their source code even though they capture multiple elements. Often we want to provide a table with fewer entity fields than what a form provides. In order to mark entity fields used for table generation, MD-Form profile uses the TableColumn stereotype. The generated person table code in Listing 4 is rendered as the table in Fig. 9.

5. FORM FIELD ACCESS CONTROL

As one entity can be used by multiple users and its view form is influenced by user access rights, there may exist multiple forms for the same entity. The following strategies to deal with access control are considered:

One static form per entity with selective field disabling. This applies additional logic in the presentation layer that decides whether the user can see a given field.

Multiple static forms per entity. Forms are generated for each form-role combination. An appropriate form is selected based on the user roles supplied in the run-time.

Run-time form generation. A form is generated upon a request and user roles are taken into the consideration.

In the following subsections, we discuss advantages and disadvantages of each case and provide our performance evaluation results.

5.1 One Static Form Per Entity

The first strategy is to use only one form that contains all the fields, and to selectively disable fields restricted for a given role(s) and context. In this case all the logic determining which fields to use is added to the presentation layer. On the other hand, the usage is simple and can be utilized by systems with smaller variety in restricted fields per user roles. This strategy can handle role hierarchy and the use of multiple roles per user. The disadvantage of this strategy is that all decisions are repeated per form request which may impact the performance. Second of all, the maintenance of the security is more difficult than in the alternative options. An example code snippet is available in Listing 5.

Listing 5: Presentation layer logic in a form (Seam)

```xml
<ui:decorate
      template="/WEB-INF/form/person.xhtml">
  <ui:param name="bean" value="#{person}" />
  <ui:param name="salaryRender" value="#{fb:hasRole("manager")}" />
  <ui:param name="notesRender" value="#{fb:hasRole("manager")}" />
</ui:decorate>
```

5.2 Multiple Static Forms Per Entity

The second strategy is related to the UML profile discussed in Section 3, which allows us to push the security decisions down to the class model. MD-Security profile allows to specify FormRoles which are the roles with access to the field. Multiple forms are generated per entity for specified roles. This results in a multiple forms in the system for the same entity with different fields available. These forms are then available in a folder specific for each role(s). The proper entity form is selected from a location influenced by active user roles at the request time. For a system where a user has multiple roles and only a single role can be activated in given context, the physical form fragment location is determined by this role (see Listing 7). In this case the system contains roles × entity forms. The difficulty comes when we consider a system with multiple roles activated at the same time. In this case we apply role union defined earlier. For multiple role activation scheme up to 2^roles variations of forms per an entity exists.

This second strategy at first, pushes the security decisions to the domain model (see Listing 6). Second, it generates all the forms before they are used. Third, it provides a form for the combination of an entity and system roles. This, on one hand, speeds up the form load as no field related access decisions are made in run-time. On the other hand, this may result in too many forms per an entity. An example code snippet for the entity is available in Listing 6, with the use in view in Listing 7.

Figure 9: Generated person table

<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Burns Tom</td>
<td><a href="mailto:tom@burnes.com">tom@burnes.com</a></td>
</tr>
<tr>
<td>Doneck Bill</td>
<td><a href="mailto:bill@doneck.com">bill@doneck.com</a></td>
</tr>
<tr>
<td>Nowak Bob</td>
<td><a href="mailto:bob@nowak.com">bob@nowak.com</a></td>
</tr>
<tr>
<td>Smith John</td>
<td><a href="mailto:john@smith.com">john@smith.com</a></td>
</tr>
</tbody>
</table>

Listing 4: Person table code

```xml
<h:form id="formPerson">
  <rich:dataTable id="TablePerson" var="element" value="#{collection}'">
    <rich:column id="name" label="Name" sortBy="#{element.name}'">
      #{element.name}'
    </rich:column>
    <rich:column id="email" label="Email" sortBy="#{element.email}'">
      #{element.email}'
    </rich:column>
  </rich:dataTable>
</h:form>
```
A similar approach is applied to Java classes by Seam Application Framework [1] where an annotation @Restrict controls the access to a class, class method or an attribute. In this case the restriction is more complex, we may apply restriction based on user role, or a context such as equality of an active class fields, context variables etc.

5.3 Run-time Form Generation
The third strategy is not to pre-generate forms, but create them on demand in run-time. No physical location for such a form exists. Instead, the entity class is inspected and the form is generated every time a it is requested. The advantage over the previous two cases is that we do not need any disk space available for multiple role-forms and that the forms do not need to be located and fetched. On the other hand, inspecting an entity and generating a form require us to open multiple files from a UI widget library in order to generate the form. File accesses could be reduced in this case if all widgets are held in memory. In this strategy we see that the performance bottle neck is the per-demand form generation and a shared cache application is appropriate. In fact, if we use a cache then this strategy does not differ from the previous static forms, but provides all above mentioned advantages. The advantage is that only form-role combinations used by the system are taking space in the memory.

An example code is in Listing 6, with the use in view in Listing 8.

```
@Entity
@Table(name = "Person")
public class Person implements Serializable {
    ...
    @FormOrder(7)
    @FormRoles("manager")
    public String getNotes(){ return this.notes; }
}
```

Listing 6: Person entity field access control

```
<ui:decorate template="/WEB-INF/form/
    #{fb:getRoles()}/person.xhtml">
   <ui:param name="bean" value="#{person}" />
</ui:decorate>
```

Listing 7: Person static form selection (Seam)

```
<ui:decorate template="#{fb:genForm('Person.class')}">
   <ui:param name="bean" value="#{person}" />
</ui:decorate>
```

Listing 8: Person run-time form generation (Seam)

```
@FormOrder(7)
@FormRoles("manager")
public String getNotes(){ return this.notes; }
```

5.4 Performance Evaluation
In order to evaluate performance of form field access control strategies, we build a small application in Seam Application framework (J2EE) [1]. We design an entity Test with 10 String fields and apply 3 user roles:

- **Admin** is allowed to see all 10 fields
- **User** is allowed to see 5 fields
- **Guest** is allowed to see 1 field

The Test form is embedded in a web page that is accessible under a selected role. We evaluate the page load times. In our first experiment we use a plain web page with the form that uses presentation logic (a) to restrict field access. Next we apply forms statically generated (b) for all system roles and their combinations. Afterwards we employ forms generated at the run-time (c). Lastly we use optimization for the run-time generation with caching (d). For the measurement we use FireFox 3.5.9 web browser with disabled cache using a web debugging proxy. Each measurement consists of 10 samples. Measurement results are presented in Table 2. In Table 2, the columns (a) - (d) represent strategies discussed earlier. The result shows that (a) is sufficient for applications where the full forms are applied and a small variability for field restrictions applies. As the restricted field count grows, other approaches seem to outperform (a). One surprising result can be that (c) is not much slower compared to (b) although it provides the poorest performance for the full form. Optimization to (c) which we denote as (d) should provide asymptotically similar results to (b).

The above evaluation tests a plain web page with a form, this might provide results that do not reflect a real scenario where a web page contains multiple components and resources which impact the page size and overall load time. In order to provide broader evaluation, we apply our approaches to an enterprise level production application (Section 6), where we use a web page that contains additional components and measure how our form field restriction approach influences the load time. In this measurement, we again apply the Test entity. Table 3 provides the measurement, where the page size notably increased requesting 17 resources (file) from the server. The second measurement amortizes the overall load time because of the additional components of the page. The comparison results in this measurement show that (a) needs the longest time to load the page, where (b) provides the best timing, (c) and (d) provide performance very close to (b).

From both measurements we received impressive results for the dynamic form generation with (c) and (d). The performance for the form generation could be improved by keeping all the form widget templates in memory.

### Table 2: RBAC Performance evaluation – plain page

<table>
<thead>
<tr>
<th>Role</th>
<th>Approach</th>
<th>Page size</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a) One static form</td>
<td>[ms]</td>
<td>[B]</td>
</tr>
<tr>
<td>Admin</td>
<td>296</td>
<td>313</td>
<td>1157</td>
</tr>
<tr>
<td>User</td>
<td>245</td>
<td>196</td>
<td>1037</td>
</tr>
<tr>
<td>Guest</td>
<td>244</td>
<td>100</td>
<td>947</td>
</tr>
</tbody>
</table>

### Table 3: RBAC Performance evaluation – Enterprise level production application

<table>
<thead>
<tr>
<th>Role</th>
<th>Approach</th>
<th>Page size</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a) One static form</td>
<td>[ms]</td>
<td>[B]</td>
</tr>
<tr>
<td>Admin</td>
<td>314</td>
<td>337</td>
<td>1117</td>
</tr>
<tr>
<td>User</td>
<td>283</td>
<td>277</td>
<td>1037</td>
</tr>
<tr>
<td>Guest</td>
<td>947</td>
<td>95</td>
<td>947</td>
</tr>
</tbody>
</table>
teams, staff, sponsors, arrivals, hotels, the contest scores, etc. This system is built on J2EE technology [1] with three tier architecture. Significant amount of data is collected every year. The data gathering process goes through the web interface that involves many forms and actions. CM was built from the domain model through the business logic to the view. This system used form generation from its first iteration. Form generation has shown to be a significant simplification in the development process. Furthermore, multiple domain model changes did not cause any incompatibility with the forms. One of goals we had for CM was to design a form that is suitable for both read-only and editable modes. In fact both views are necessary as we may need to print out a page and the editable form may crop some texts. Each CM form is embedded into a widget with a mode parameter. In addition each field can be selectively rendered, which we apply to restrict access for specific system user roles with the addition of a presentation layer logic. In the CM application overall 41 forms are generated. Form field labels and tooltips using internationalization are supported as well as resource definition enforcement by the generation. The way the forms are generated unifies the naming conventions of the application and simplifies testing. UI tests can predict the generated forms and auto-prepare semantically valid testing data as the field meaning is available. We apply these tests using Selenium Framework [2].

## 7. RELATED WORK

Multiple earlier approaches on user interface generation exists [6] [7] [22] [12] [14] [21]. The survey of Model-Driven Engineering (MDE) tools for human computer interaction (HCI) [22] discuss advantages that can be brought by MDE tools from the usability perspective. Authors believe in the need for MDE tools for UI design. Our proposed profiles can be applied to existing UML tools allowing us to capture more presentation specific information. By extension of the models we can capture more information for classes and their attributes. This provides the ability to generate more user friendly UI, better guidance, expected data description and UIV. Investigation of the use of MDA for development of HCIs is also goal of [12]. Authors bring a discussion of a gap between HCI and System Engineering. They mention that one may argue that relating design models to the user interface can be considered as a mix of presentation and persistence logic; however, HCI concerns cannot easily be described independently from other concerns for a system. In our approach we believe that the separation of concerns is important, but looking at aspect oriented programming (AOP) [17] we see a need of joint points which can be seen as stereotypes we describe. In addition AOP is commonly applied to security where the access control is weaved into the clean application implementation. Moreover AOP allows us to define multiple fragments focused on specific aspects and define joint points where these fragments are weaved together. This may produce significantly more code than what the input fragments provide. This approach supports maintainability as simple fragments are easy to read and the coupling is reduced. Our approach with form generation, mostly then with security consideration with multiple roles, is not very different, when we think of the entities as the original code and the stereotypes as joint points determining which view widgets to assemble in order to receive the desired view fragment.

Using MD-JPA [24] for ORM mapping is a promising way to fill the gap between model-driven approach and manual development. Further research [7] goes beyond the UML profile use for ORM, provides profiles for UIV and form generation with access control and brings the connection of these aspects. The insight from the lower level perspective is described by [6]. Details on entities, view widget library, mapping and form generation dependencies are explained and demonstrated by a tool called FormBuilder. A similar idea is also implemented in Naked Objects [21] where developers focus only on the domain object implementation, and the platform generates the whole UI. These were successfully used for Irish Government. A tool called MetaWidget [14] applies, similarly to FormBuilder, software mining [16] and an inspection [15], it also allows to set various layouts. In [15] authors argue that research has shown that some 48% of application code and 50% of application time is devoted to implementing UIs and automation is beneficial. They discuss advantages and disadvantages of three approaches used these days for UI generation. Interactive and model based tools require to restate information already existing in the system and lack of support for maintenance. Language based tools on the other hand need addition information introduced in the code. Our approach is more abstract as we do not bind to a specific domain or a programming language. The main differences between our tool and MetaWidget is that MetaWidget is given a collection of widgets which developer cannot modify. We provide access to the widget collection, which makes it possible to adapt to any Java-based framework, android, DSL language or Web services. This also allows application of any security framework, view template framework or form field access control discussed in this study.

Many tools were introduced for the form generation requiring addition model often defined in XML. IBM XML Forms Generator an eclipse plug-in [13] can generate XForms [5] from given XML data instances. It can generate form elements that satisfy type and length constraints and control types according to a given XML scheme. Unfortunately XForm technology is not much adopted. Our tool allows us to customize widgets, which can be for example produce XForms in result. Eclipse E4/XWT (XML Window Toolkit) [4] uses a XML based design of the UI, where the XML rules are translated to the Java code. Our approach is much different, from a UML class model we generate the XML fragments and avoid manual XML coding with weak type-safety.
8. CONCLUSION
When many rich view forms are needed for an application, one of common practices is to define a data model with its design entity class model first, and then manually design view fragments from them. However, this way of manual form development is tedious, very error-prone and the problem becomes even worse when the entities are modified later but those changes are not propagated to view fragments. Therefore, ensuring all changes that are manually made during the maintenance time to be applied to all related view forms has been one of key challenges in many form generation approaches. MDD approaches have been recommended as a new way of developing high quality applications from precisely specified modeling artifacts. Unfortunately, MDD has not been very successful in addressing the generation of complex UI fragments. In this paper, we have shown the use of UML profiles to capture additional information that is necessary for the form generation.

This paper extends our previous work [7] and our extension includes how we can generate forms for certain user roles that provide only information related to that user. We have provide our production experience summary as well, which addresses previous work from the paper that targeted access control and on demand generation. We provide a measurement and evaluation of on-demand form generation that provides similar performance as an existing form-pre-generation. In addition, we have shown that our approach can benefit to the load time performance of a form that displays only a small subset of elements of the complete form to the end user. We have also shown the use of our form builder for the table generation.

Our current approach can also be used with other traditional (programming) development tools such as ruby on rails [23] or seam-gen [1] as well as other MDD-based development tools. As one of future extensions of our work, we plan to measure the coupling between entities and their UI fragments.

9. REFERENCES