Towards Effective Adaptive User Interfaces Design

Tomas Cerny
Czech Technical University,
Charles square 13
121 35 Prague 2, Czech Rep.
tomas.cerny@fel.cvut.cz

Michael J. Donahoo
Baylor University,
One Bear Place #97356
Waco, TX, 76798-7356, USA
jeff_donahoo@baylor.edu

Eunjee Song
Baylor University,
One Bear Place #97356
Waco, TX, 76798-7356, USA
eunjee_song@baylor.edu

ABSTRACT

The increasing use of Web-based applications continues to broaden the user groups of enterprise applications at large. The importance of providing easy-to-use user interfaces (UIs) that conform to each user's specific preferences, such as different skill levels, capabilities and physical locations has, therefore, been significantly increasing. Unfortunately, designing a single UI satisfying all end users remains challenging. To address this issue, researchers and developer are looking to Adaptive User Interfaces (AUIs) that aim to provide end users with more personalized user interaction experiences. However, very few production system provide such malleable interfaces due to the excessive cost for the development and maintenance.

In this paper, we propose a technique that provides AUIs for production enterprise systems while reducing development and maintenance efforts to a level comparable with a single UI development, called Rich Entity Aspect/Audit Design (READ). READ complies with application development standards used in industry to support an easy transition from design to production systems. We conclude by evaluating our approach along with a case study that demonstrates reduction in development and maintenance efforts while preserving performance.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; I.2.2 [**Artificial intelligence**]: Automatic Programming—*Program synthesis*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software maintenance*

General Terms

Design

Keywords

Aspect-driven design, Inspection-based approach, Adaptive user interfaces, Reduced maintenance/development efforts

RACS'13 October 1-4, 2013, Montreal, QC, Canada.

1. INTRODUCTION

Although an application should serve various users from different geography locations, with different capabilities and skills, it is a common practice to design a single UI for everyone [18] instead of providing user-specific UIs. The primary reason for this one-size-fits-all approach to UI design relates to the costs of development and maintenance for multiple UI versions. For example, [10] states that around 48% of application code and 50% of development time are devoted to implementing UIs. Thus, providing multiple versions of UIs for individual users is typically considered to be unrealistic.

In case of many existing programming techniques, it is difficult to support UI features, such as the adaptivity to users, mainly because they capture field-specific information twice; once in the data-model and again as a reference in the presentation that is often specified in XML with no type safety (form, table, etc.). In addition, current practices realize multiple UI concerns mixed together in a single component, which makes such a component less cohesive and hard to reuse. As shown later, this results from the inability of conventional approaches to capture different concerns separately. The development of less cohesive components results in multiple, highly-similar components that only differ in details. Having a multi-location field definition and multiple similar components for a slightly different presentation brings further difficulties throughout the development. For example, changing the underlying data definition requires all of its presentation components to be updated, which is a non-trivial task as no type safety may exist. Considering that such a component update process is manual and certainly error-prone, it is most likely to introduce more errors or omit required component updates, which eventually results in the inconsistency in presentation.

Our proposed technique utilizes the information from an application's data-model and its existing structures obtained from the automated code-inspection. Such information is then extended and transformed into the UI in multiple steps through aspect-oriented programming (AOP) [12]. Concerns that are commonly tangled together are now separated into easy-to-maintain, reusable units, called *aspects*. The transformation process weaves all separated concerns together at runtime and thus allows us to consider user-context conditions individually. The resulting UIs can dynamically adapt to all considered concerns to satisfy users. To evaluate our technique, we implement a library and demonstrate its use in a case study with enterprise JEE6 application.

The main contribution of our approach is the reduction of information restatement in UI development and the sep-

aration of concerns that are directly responsible for tangled UI code. Multiple information restatement steps required in existing approaches collapse into a single focal point of information in our approach, which makes the enforcement of its UI compliance easier. Since it is executed at runtime, it can dynamically adapt the UI to a user-specific context. The approach reduces both development and maintenance efforts through the component reuse. Despite the addition of these benefits, our approach has a minimal impact on application performance.

The remainder of this paper is organized as follows: Section 2 describes the background of adaptive user interface development. Section 3 provides an overview of existing approaches. Our approach is presented in detail in Section 4, and its evaluation is discussed in Section 5. The final section presents our conclusion and future work.

2. BACKGROUND

One approach often taken to deal with system complexity is to break the system down into units of behavior or function such as subsystems, modules or objects, called functional decomposition in Object-Oriented Programming [12]. Such a decomposition concept is necessary because it helps one to put logically-related concerns together, improves the readability and reusability, and eventually supports the ease of maintenance [13]. In addition to functional decompositions, the Aspect-Oriented Programming (AOP) community proposes another way of thinking about a program structure. The key unit of modularity in AOP is the aspect that enables the modularization of concerns, such as transaction management, that cut across multiple types and objects [13].

UI development also employs such decompositions, but its data presentation makes the decomposition process more challenging, especially when a markup language is used to describe the UI, which is common for web systems. For example, consider designing the Person form given in Figure 1. The arrows highlight various concerns considered in the design. `Arrow 1` shows that form fields are bound to a particular data class, an entity, and its fields. This binding means that, for example, when the field called `name` in `Person` splits into `first name` and `last name`, its corresponding form field must split as well. Unfortunately, there is no enforcement mechanism to guarantee that the corresponding entity and its UI comply with each other unless a language with type-safety is used. An entity field UI presentation is denoted by `Arrow 2`; an appropriate UI widget and its properties are chosen based on the type of a particular field and its constraints. Anytime a field constraint changes, an underlying widget or its properties should reflect the change as well. However, there is no automated mechanism to do so, thus a manual update is necessary for each field change. `Arrow 3` demonstrates that the form may allow one to select a particular presentation layout. A layout is responsible for rearranging form fields in a given order, grouping them together or presenting them within a given screen size. Designing a non-trivial form layout often results in a layout code entangled together with form fields. We provide an example of tangling such concerns later in Listing 1. When an application adjusts a form layout at runtime based on a given condition, it is possible that multiple variants of forms physically exist to represent the same data. Next, `Arrow 4` indicates that form fields should consider additional UI conditions such as security or visibility. For example, some fields

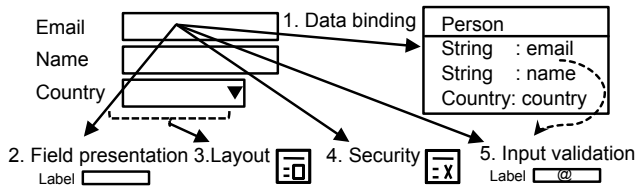

Figure 1: UI form decomposition

should be rendered as read-only or left unrendered based on the given user authorization. In order to apply the conditionals, we further extend the form fragment, leading to more complex readability and perhaps duplication among fragments applying various layouts. Finally, `Arrow 5` shows that certain constraints from the bound entity fields should be applied for its input validation. For instance, web applications with client-side validation must restate constraints in a scripting language, such as JavaScript. Listing 1 shows a very simplified implementation of Figure 1; this JavaServer Faces (JSF) code shows data binding to the form through a data instance i (*value* attribute in widgets), field representations through UI components (*x:input*), table layout tangled through the fields, security condition (*render* attribute) and validation (*validate* attribute) determined by methods in a controller accessible in the context as *bean*. The maintenance of such fragments becomes difficult because all five concerns are captured together, and it is non-obvious which code refers to a specific constraint such as security, presentation, layout. The reuse of tangled UI fragments is limited since it only allows the slight variation of concerns. AUI design only compounds the problem since it typically increases the number of concerns.

The [13] explains that an n-dimensional concern space is expressed in the implementation space using a one-dimensional language. Unfortunately, orthogonality of concerns in the concern space gets lost when it is mapped to the one-dimensional implementation space. For our case we have a 5-dimensional concern space as shown in Figure 2 (a). This concern space is mapped into one-dimensional implementation space in Figure 2 (b). This corresponds to what we see in Figure 1 and the one-dimensional implementation in Listing 1.

Consider design of an adaptive (rather than merely single) UI, where the above mentioned concerns extend with user-specific presentation and field restrictions influenced by the user's location, his age, temporal information or layout adjusted to the user's screen size, etc. In this case, the number of concerns in Figure 2 (a) grows and the complexity represented by Figure 2 (b) becomes even greater, because all considered concerns tangled together are directly responsible for increased development efforts, hard maintenance, diminishing readability, limiting reuse, higher possibility creating errors, etc.

3. RELATED WORK

UI development approaches can be divided into two groups *restate-to-extend* and *inspection-based* [11]. *Restate-to-extend* requires that the same information in a system be captured twice at different locations, while preserving its integrity. Such information duplicity is then applied to a particular concern such as UI presentation. Development using this approach typically involves interactive graphical tools, model-based generation tools [14, 17] or external models for UI

Listing 1: Sample code snippet for form in Figure 1

```
<table class="classLayout"> <tr><td>Email:</td>
 <td><x:input id="email" value="#{i.email}"
     render="#{bean.render('email')}"
   validate="#{bean.validate('email')}"/></td>
</tr><tr><td>Name:</td>
 <td><x:input id="name" value="#{i.name}"/></td>
</tr><tr><td>Country:</td>
 <td><x:smenu id="country" value="#{i.country}"/></td>
</tr></table>
```

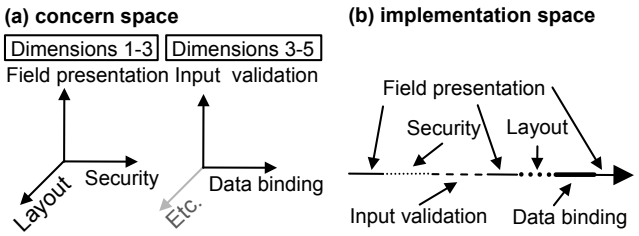

Figure 2: (a) Concern / (b) Implementation space

representation [15]. The main drawback of these approaches stems from the duplication of source information and additional maintenance efforts when source information changes. Model-Driven-Development (MDD) [7] argues that all information should be captured in the model and the code itself is solely generated from the models. Unfortunately, the MDD approach suffers during adaptation and evolution management, as noted in [17]. The specification of possible states and configurations of complex systems can grow exponentially. Once deployed, such systems experience changes in variations, which often take place in code rather than in the model itself so code regeneration from the higher abstraction model can be impractical and the manually added information can get lost [7].

Inspection-based approaches use existing information accessible by code-inspection. The effort in this case is placed on the information source that must capture sufficient information to derive a specific concern. Development using this approach typically involves language-based tools. The disadvantage of this approach is that source information does not necessarily capture all needed concerns. Multiple research proposals such as [6, 10, 11] utilize automated UI generation by applying code-inspection. These approaches inspect previously captured information, build a meta-model adhoc and transform it to the UI. This approach simplifies development and maintenance since it reduces restated information. The difficulty is that such an approach cannot generate the UI unless provided additional information, typically supplied by additional markup within the source information [7]. In this approach, it is important to consider that data-models already capture persistence and validation constraints (e.g., Java technology standards [2, 8, 9]). Such information should be considered in the inspection to avoid duplication. In [7] the authors provide multiple profiles describing data-model extensions for persistence, validation, security and presentation.

These approaches provide individual benefits; however, neither of them addresses cross-cutting concerns. An AOP approach [12, 13, 19] provides methods that allow us to capture different concerns separately in independent code fragments. In our work, we aim to produce a combination of various concerns. In the conventional, object-oriented approach, these concerns are tangled together, creating code that is hard to read and maintain. An aspect approach takes the base code that is simple to read and adds additional aspects to it through a compiler called an aspect weaver. The product of the aspect weaver can have the same execution properties as tangled code, but all the concerns can be defined separately to support readability and maintenance. [12] shows that AOP can reduce the total of lines of code (LOC). In their example, they reduce an application code from 30,000 LOC to 1,000 LOC by applying AOP. AOP concepts are nowadays used in development frameworks [13], modeling [17], security, performance optimization, etc.

The idea of AUI is studied in multiple domains. For example, we can see its application in a hospital navigation case [15] and in a house control unit example [3]. Multiple AUI design methods require the target environment and possible variations of the user interface at design time [14, 16]. In [4] the authors argue that future adaptive systems need to consider runtime information to adapt, thus design time approaches should not be considered. [17] and [3] apply aspect-oriented techniques to a model-based approach to deal with multiple degrees of variability that depends on user needs and context. Most of the related work on AUI focus solely on the problems related to features of AUI and typically apply model-based approaches that would need to restate information from application backend. The UI is then a result of transformations of the models to the UI.

Note that none of the mentioned approaches address adaptive UI. At the same time, runtime adaptivity, reduction of restated information, separation of cross-cutting concerns enhances production environments, while reducing the the source code. In our approach, we try to address all mentioned features. Naturally, we avoid the *restate-to-extend* approach.

4. READ : RICH ENTITY ASPECT/AUDIT DESIGN FRAMEWORK

As shown in the related work, in order to design an AUI with low development and maintenance efforts, we should avoid manual definition of an additional model that restates information captured elsewhere in the application. Instead we should consider a *code-inspection* approach and AOP design. *Code-inspection* fetches already-captured information from the data-model plus application structure and prevents duplication. The meta-model assembled by the inspection should capture information needed for UI composition and adaptation. In order to support flexible adaptation of various concerns, we consider the runtime aspect model. Properties of an aspect approach allows us to capture different concerns separately rather than tangled together. These properties can utilize the meta-model and based on the runtime context, apply appropriate UI concerns that are all together transformed to the UI presentation. The nature of such an approach brings the benefit of coherence between the UI and the backend part of the application. Thus changes in data-model or application structure are immediately reflected in the UI. Other approaches may produce inconsistency as both the data-model and UI part are maintained separately. In the following subsections, we describe our approach on a new UI development framework, READ.

4.1 Introduction to READ conceptual model

In order to describe AOP framework, [19] suggests describing its conceptual model with three main components:

Join Point Model: defines available join points

Pointcut Language: defines the query language to select a subset of join points

Adaptation Mechanism: allows adding or modifying functionality at selected join points

These components well describe existing frameworks such as AspectJ or Hyper/J in which we often modify or add functionality upon method call or code execution. In our case, the adaptation mechanism does not constraint any method or code execution but deals with transformation and composition.

In READ the *join point model* consists of both static and dynamic join points [19]. *Static join points* are object data-model class names, field names and data types, and also field annotations with their parameters. All these join points are known at the compile time. During the code-inspection, all these information are passed to the READ context. *Dynamic join points* do not correspond to elements in code and can be influenced by runtime application context, which can be any information passed to the READ context at runtime from the application such as user access rights, geo-location, local context for presentation, device screen size, etc.

The *pointcut language* defines the query language to select a subset of join points. READ uses an expression language know as Unified Expression Language[1] (EL). EL consists of constructs for conditionals and arithmetical operations, understands basic types, and can evaluate any expression referring to its context. In READ the EL context has access to the READ context, thus all information in READ context can be queried by the pointcut language . Furthermore, it is possible to add to the context custom objects or functions. The language uses both the *state-based* and *specification-based* constructs [19].

The *adaptation* takes place after the code-inspection populates the READ context, which keeps information about inspected data instance in a form of a composite structure. Instance information is at the root, inspected fields are its child nodes and child leaves reflect field information and constrains. The leaf level captures field static join points. In the transformation process, each field is transformed through *presentation rules* that use the pointcut language to query field join points to select appropriate advice in the form of a *presentation template*. Both rules and templates are introduced later. After the appropriate template is selected, its content is interpreted. The template can be seen as a composition mechanism integrating different aspects. It consists of a DSL language that looks like the target domain language describing presentation, but also it uses additional markup language that is being interpreted by the READ weaver. The markup uses the pointcut language to integrate various aspects to the presentation code. Thus it queries the field join points, and based on the result, it integrates given aspects. After all data fields selects a template that provides its UI presentation and integrates addition aspects, then READ considers layout integration. The layout composition uses *layout template*, similar to field template, that uses a DSL language from the target domain language with additional markup. This markup provides the developer mechanisms to describe specific or an anonymous field position in the template or to iteration over multiple anonymous fields.

Listing 2: Example entity with additional markup

```
@Entity @Table(name = "personInfo")
public class PersonInfo {
   ...
  @UiUserRoles({"Admin","Owner"})
  @UiOrder(1) @Enumerated(EnumType.STRING)
  public Title getTitle() { return title; }

  @UiOrder(2) @NotEmpty @Pattern(regex="^[^\\s].*")
  @Length(max=100) @Column(nullable=false, length=100)
  public String getFirstName() { return firstName; }

  @UiOrder(8) @UiProfiles({"US"})
  @NotEmpty @Column(nullable = false)
  public String getHomeState() { return state; }
}
```

Listing 3: Example presentation rules

```
<mapping>
 <type>String</type>
 <default tag="textTag.xhtml" size="20"
   javaPattern="" minLength="0" maxLength="255" />
 <var name="Person.username" tag="emailTag.xhtml"/>
 <cond expr="${email == true}" tag="emailTag.xhtml"/>
 <cond expr="${link == true}" tag="linkTag.xhtml"/>
 <cond expr="${maxLength>255}"tag="textAreaTg.xhtml"/>
</mapping>
```

Consider the data entity implemented in Java in Listing 2. This entity follows persistence and validation standards [8] [2]. You may also notice that it is possible to extend the entity with additional markup; we call such an extended entity a rich entity. This rich entity is a subject of inspection/audit that populates the READ context with the entity related information and makes it available for pointcuts. The READ context now consists of entity-related information. The first-level adaptation mechanism applies *presentation rules* defined in a configuration file. Each rule has a pointcut for the entity field context. A matching pointcut gives advice on a presentation template to use for the field. Consider the configuration file snippet defining *presentation rules* for *String* types in Listing 3. Note that for a given field type, a single advice applies based on the matching a pointcut defined by the *expression* attribute. When no match exists, then the default one is used. The pointcut uses EL and queries the READ context related to the evaluated date field, thus it has access to all the field properties - static join points (name, type, parent, annotations, annotation parameters, etc) or to other parameters - dynamic join points. Second-level adaptation mechanism uses the *field template* for composition. The field template defines field presentation and integrates other aspects to it. It uses the pointcut strategies shown in Listing 4 (a,b,c). The example shows three types of pointcut strategies where all do the same thing. They query the considered field for existence of a join point *minlength*. If it is present, then it embeds it to the code fragment. The pointcut uses EL within the dollar marks and may use any combination of join points with logical/arithmetical operations or constants. The pointcuts in Listing 4 (a,b,c) show *(a)-full/ (b)-brief/ (c)-shorten* strategy for the pointcut/-composition rule. The composition rule integrates the join point for the given data instance to the field presentation. The *full* version description approach separates the pointcut and aspect composition; when the pointcut evaluates to true, then the body applies. *Brief* version provides the same

Listing 4: Pointcut strategies for templates

```
(a)   $not empty minlength ; x = minlength $
      minlength="$x$"
      $$
//-------
(b)   $not empty minlength
       ? "minlength=\"".concat(minlength).concat("\"")
       : "" $
//-------
(c)   minlength="$minlength$"
```

Listing 5: Example template for inputText widget

```
<x:inputText id="#{prefix}$field$"
     label="#{text[‘$entity$.$field$‘]}"
      edit="#{empty edit$field.firstToUpper()$
         ? edit : edit$field.firstToUpper()$}"
     value="#{instance.$field$}"   size="$size$"
  required="$required$"          pattern="$pattern$"
 minlength="$minLength$"       maxlength="$maxLength$"
     title="#{text[‘title.$entity$.$field$‘]}"
  rendered="#{empty render$field.firstToUpper()$
         ? ’true’ : render$field.firstToUpper()$}"/>
```

result but needs less code. The *shorten* version fits to common cases and needs the least code. To see the composition in context, consider the example presentation template for JSF code shown in Listing 5. The third-level adaptation uses a similar approach with the difference that pointcut context is on the class level rather than on the field level. Also for the purpose of complexity reduction, a field iteration mechanism is defined. Consider Listing 6 that presents an HTML table that can weave into it field fragments that result from the second-level adaptation. Note the expressiveness of the markup in the example. The pointcut is specified within dollar marks and can integrate, for example, a specific field to the layout by associating its name within the template, this will embed the field presentation from the second-level adaptation (*af:notes*). Alternatively, a more generic approach can be used to iterate over anonymous fields within a fragment of code (*iteration-part* and *af:next*).

4.2 READ Framework

The UI page can be seen as a composite of components represented by a component tree. For example the root element can be a panel under which attaches other components, such as panels, inputs, labels, etc. An alternative example can be an XML page represented by a document object model or a JSF page consisting of a view root and UI components of various types in a tree structure under the view. When a UI page is being rendered, a page renderer traverses the tree of components and transforms each component to the UI presentation. Each component has commonly associated a handler called by the renderer. To attach our approach to this process, we simply implement a custom READ component handled by a custom READ handler. Thus, when the renderer interprets a READ component, it calls the READ handler and starts the process within the READ framework. Our proposed framework is illustrated in Figure 3, capturing various consequent stages with sequence given by alphabetical order of stage labels and denoting stage transitions by arrows. The first stage a is the renderer that processes the component tree. Once a READ component is processed in b, then the READ handler is called, in c. It should be noted that it is possible that a component has other components attached under it, and thus the handler result can be a component sub-tree, that

Listing 6: Example layout template

```
<table class="classLayout">
<af:iteration-part maxOccurs="100">
 <tr><td>$af:next$</td><td>$af:next$</td></tr>
</af:iteration-part>
 <tr><td colspan="2" class="foot">$af:notes$</td></tr>
</table>
```

Listing 7: Example use of READ UI component

```
<h:outputText value="Person Info Form" />
<af:aui instance="#{bean.instance.personInfo}"
     layout="personInfo-wide-layout"
     edit="true" ignore="password,notes" />
<h:commandButton action="#{bean.save}" value="save"/>
```

can be further interpreted. An example page code fragment for JSF is shown in Listing 7. This fragment contains a two basic components for text and a button, and then a READ component with the prefix af. Note that the component references an instance that is used for inspection and may suggest a layout and addition directives or constraints for presentation.

READ handler (c) can be seen as a controller of the entire process. First, it receives an instance reference from the component property and directives that are made available in the READ context for pointcuts. The READ handler is accessible and thus developer may integrate other third-party frameworks or runtime information, such as screen-size, access roles, geo-location, etc., to be accessible by the READ context. Second, the handler calls inspection/audit on the entity instance for which it generates the presentation (d). The aim of the inspection is to populate entity join points and to make them available to the READ context. Our inspector implementation uses a Reflective API to produce a meta-model that is a three-level hierarchical structure described in the previous section. It is aware of entity settings, its fields, and their properties including third party annotations with their parameters. Since such information is class related, the inspection mechanism caches the meta-model for given class. The meta-model ($d.1$) can be further restricted by a given context. Some of its elements are filtered based on the Annotation Driver Participant Pattern (ADPP), commonly used for custom presentation settings, security, etc. The outcome returns to the handler as a context-aware entity meta-model. The handler then makes this outcome available as join points in READ context. The next stage is the transformation e. The transformation weaver uses presentation rules; each rule consists of a pointcut and advise that suggests a particular composition template which defines integration of composition rules. First, the READ weaver takes each meta-model field with its join-points and finds a matching pointcut in the presentation rules $e.1.1$. The pointcut then advises the appropriate composition template $e.1.2$. The composition template uses the target DSL to provide a basic field presentation and also integrates composition rules shown in Listings 4. For example consider a template shown in Listing 5; it is expected that developer may adjust presentation rules and composition templates according to their system. The READ weaver then interprets the composition template based on the composition rules and uses the READ context to interpret each rule pointcuts and the rule body. The result of the template weaving process is a field code fragment in the target DSL. Such a mechanism scales well towards the concern space

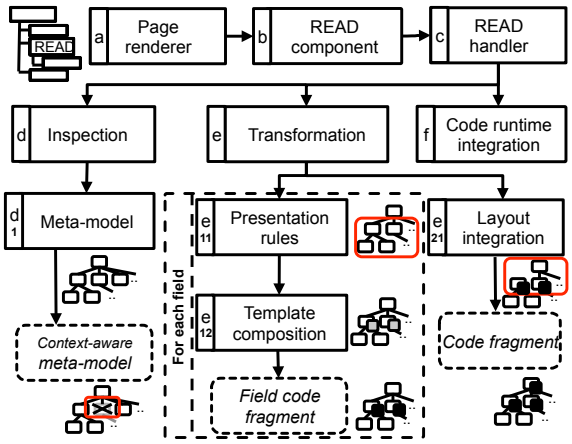

Figure 3: READ framework

as new aspects can be integrated to the template as well as to the presentation rules. After all fields are processed a code representation exists for each field, then the layout $e.2.1$ decorates the field-code. Layout integration can be seen as product of XML transformation to XSLT. In READ the target language field-code is in DSL and the layout described in layout template uses references to named fields, anonymous fields or an anonymous field iterator as shown in Listing 6. This stage results in a DSL code fragment capturing all considered aspects. The product of the transformation is then passed back to the handler. In the final stage, the DSL code fragment compiles at runtime, and the compiled output embeds to the processed component tree for further processing by the page renderer.

4.3 Design with READ

Next, we discuss software design with the use of READ. Assuming that we build on the top of an enterprise architecture using 3-layers, the system has a persistence layer that captures its data-model by classes and applies object-relational mapping (ORM). For example Java EE defines standards [8, 9] for the ORM, which extends the class model with additional markup. Similarly a validation [2] can be added. Generalization of such extensions and further enhancements are suggested by [7]. READ inspection uses all of this information for the meta-model composition and for join points. Besides the data model, READ can also integrate business rules defined in the above layer. Preliminary work in [5] shows that business rules can be inspected and their definitions reused. This can be integrated to the READ context. Considering common development approaches, so far we only expect data-model entity extension. We refer to such extended entities as rich entities. In the presentation layer, common components can be used together with READ components. READ components take as attributes an entity instance and addition presentation directives and builds the presentation for given instance. Such a component can produce a form, table or a report. With READ, the developer does not design a form or a table directly per each page use. Instead, the developer specifies presentation rules that generalize mapping among entity fields and presentation widgets. Presentation rules are generic and can be reused among projects. The developer then designs field templates that are used by the READ weaver. These templates are also generic and can be reused. At the beginning, it might be seen as a lot of effort, but we must consider that all these templates are reused by the entire application, thus the initial work amortizes over the size of the software application. Furthermore, developers can design specialized or generic layout templates.

Where can we see the the main benefits? First of all, the system presentation reflects the actual state of the software system. All actual data definitions, all runtime contexts, and states are considered in the weaving process, thus the data presentation reflects or adapts to it at runtime. Second, with READ, the size of concern space does not increase the complexity of the system, and described concerns can be reused. Change of an individual concern is easy to locate and modify. Third, READ reduces errors because the entity becomes to be a single focal point of information, thus we do not need to restate information multiple times in the UI. Fourth, READ reduces development and maintenance effort since a new entity presentation does not require any coding. In an edge case a new presentation rule or a new template can be designed. Fifth, READ naturally supports adaptive UI design because it evaluates conditions at runtime. Sixth, READ is open for integration with third party frameworks such as EJB, Spring, Security frameworks, etc. In the case of presentation, READ templates can integrate a novel components or mechanisms. A more concrete example to this is when we use Java EE and JSF for presentation, it is possible to make templates for various component providers (such as PrimeFaces, RichFaces, Tomahawk, ICEFaces, etc.). Seventh, READ does not bind the developer to a single use approach; other approaches can be applied at the same time.

READ can integrate any new concerns to its context and can evaluate them at runtime. Our current approach is evaluated on component-based UIs, although it is not limited to it. The limiting factor can be the need to compile the output and apply it to the UI. In some frameworks, this could be complicated as it requires access to low-level UI compiler libraries. The expressiveness of the UI is not limited by READ since designer can adjust the presentation in composition templates. READ also applies to partially rendered pages and AJAX rendered views.

5. EVALUATION

In this section, we consider a subsystem of an existing ACM-ICPC system used for the registration of users and user account management. The application follows mainstream development with 3-layer Java EE. The lowest layer consists of an object data-model with 7 entities with persistence and validation constraints markup [2, 8, 9]. The business layer contains controllers with business logic, CRUD and search functionality. The presentation layer contains UI implementation using JSF technology (no type-safety).

First, we consider this application without AUI. The UI part of the application contains search with result listing plus a detail and modification page. The presentation covers the entire data-model. Form submission of data is validated through enforced business constraints upon the submission. The application provides a single data presentation in one layout. In total there are 7 data classes and 46 fields presented in the UI. Excluding configuration and external libraries, the application consist of 1342 physical lines of code (LOC) of Java, including persistence and business logic, 2221 LOC of XML presentation, and 373 LOC of XML

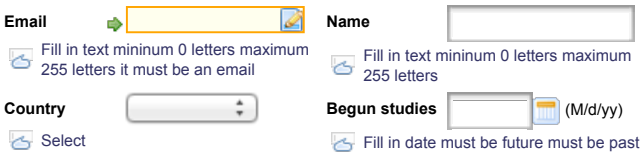

Figure 4: Sample form for confused student

Figure 5: Sample form for child

of application configuration. The type-unsafe XML presentation exhibits 564 occurrences of restated information from the data-model and its constraints [2, 8, 9]. Next, we implement the same application using READ. The data instance source code is extended with additional presentation marks [7] extending the field constraints (see example in Listing 2). The main difference is that all components presenting data in the UI are READ generated. They combine information from data instance inspection, presentation rules and presentation/layout templates. None of the stages involve a direct reference to a particular data field, which leads to 0 occurrences of restated information in the type-unsafe XML. Furthermore, presentation templates are reused. This results in 1530 LOC of Java, including the additional data-model marks and UI handler and 1715 LOC of XML including templates and transformation rules. This shows reasonable code reduction for the presentation part, but at the same time we must see the maintenance impact. In the manual approach we are directly responsible for restating information captured in data model, where the READ handles this for us. With READ we avoid inconsistency and errors, while reducing development time. The greater code reduction effect can be achieved on larger projects. Next, we should consider that presentation templates and transformation rules can be reused among projects, in such case the READ application results in 1439 Java LOC and 1534 XML LOC and equal configuration. The summary can be found in the first part of Table 1. The aspect weaver itself is not included in the evaluation because it is a generic, reusable and external library (see the reasoning in [12]).

One serious drawback of this application example is that it considers a superset of all possible end users. Thus users with large screen are provided narrow layout, elderly might need to zoom the page, internationals might wonder why they need to fill in a *state*, and non-student registrants need to provide student-specific information.

Next, we consider a more user friendly presentation supporting adaptability. It provides end-users with a presentation related to their origin using IP geo-location, adjusting to their browsing device screen size, conforming user rights, and fitting user age and capabilities. In total, there are 3 main layouts to conform the screen-size, although multiple data elements follow a custom field order among different layouts. Furthermore, we provide 4 different presentations for children, elderly, adult and experienced users, all possibly combining a given layout (see UI examples in Figs. 4-6).

The application following the mainstream development applies field restrictions, such as user rights or locations

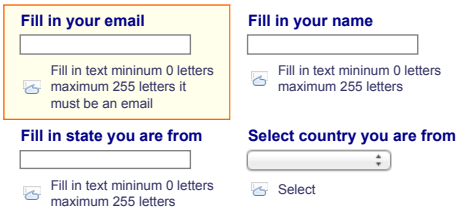

Figure 6: Sample form for elderly

Table 1: Efforts comparison

User interface	Simple features			Adaptive features		
Approach	Man.	READ	reuse	Man.	READ	reuse
Java LOC	1342	1530	1439	1658	1907	1754
UI XML LOC	2221	1715	1534	13072	5036	4508
Conf. XML LOC	373	442	373	373	649	373
Restated inf.	564	0	0	6768	0	0
UI Conditionals	0	0	0	240	20	20

awareness, throughout conditionals added to the presentation components. The problem with this approach is that markup languages have limitations in separating layout from the presentation. But also presentation cannot be separated from field binding and property settings. The mainstream approach results with 1658 LOC of Java and 13072 LOC of XML presentation while the type-unsafe presentation code consists of 240 conditionals and 6768 restated information from the data-model and its component constraints [2, 8, 9]. Consider that with this approach, developers follow the implementation in Figure 2 (b).

The READ approach allows designers separation of presentation, layout and also of security and location-awareness through various stages within the framework. One of main differences in our approach is that each concern is implemented separately as demonstrated in Figure 2 (a). The READ library combines these together. In our study, the application backend Java code includes 1907 LOC, the presentation XML reduces to 5036 LOC, including the presentation and layout templates, and 649 LOC of configuration XML. Conditionals for location and user-rights are captured in data-model, which reduces them to 20. Furthermore there are 0 occurrences of restated information in the type-unsafe XML. The overall summary of the evaluation is provided in Table 1. Consider that in this second example, individual concerns multiply and their combinations apply. Standard approaches fail to effectively design reusable UI components. The reason is behind the common approaches that fail to capture individual concerns separately, which worsen the code readability, reuse and maintenance. Untangling individual concerns through the AOP approach addresses all the code readability, reuse and maintenance more effectively as can be seen from our results.

Next, we evaluate basic maintenance scenarios. With manual development, the UI is fragile because of its coupling to the data-model in the type-unsafe environment. Changes to a data field, its name or constraints causes inconsistency in all its UI fragments. Such a simple change may lead to 12 locations that need to reflect the change. In type-safe code, this can be easily refactored, but in XML it must be be addressed by text search. With our READ approach, the UI does not refer to the data-model directly; it only refers to the instance that is presented (see Listing 7); thus it does not require any UI correction. When we want to globally change the presentation of a particular widget, in the manual approach all widget occurrences must change;

however, with READ such change takes place solely in a template. Changes to user rights manually requires reapplication of new conditionals in UI, since multiple different presentations exist for a single field. This can impact a significant amount of UI code. In READ, such change takes place in the data-model, a single location. The addition of a new form layout may require a new copy of the form with tangled layout. In READ, the layout is a separate fragment, thus only new layout template is designed.

For the performance evaluation, we consider 5 forms with total of 21 fields in it. We evaluate the time needed for the page to render using both the manual and READ approach. The load times for a page containing the forms, averaged over 250 samples were 545ms (std. dev 47) for manual approach and 539ms (std. dev 41) for READ. The measurement shows that the page load time is similar for both approaches.

6. CONCLUSION

Despite many benefits of AUIs, few production systems support employing them. The reasons behind this include the excessive cost of AUI development and maintenance as shown in our case study. We provide an approach that considers existing standards for application frameworks, aspect-oriented programing and employs code-inspection to face the complexity and efforts related to AUI design. Our READ technique considerably reduces the cost involved in the development of AUIs. We have developed a production-level library, called AspectFaces, for Java systems that implements the READ framework. This library is currently used in enterprise-level production systems.

In the future, we plan to provide an evaluation of READ over a large production system. Next, we wish to extend our approach to inspect and reuse application business rules. Our preliminary results show that such approach will provide more options and variety of adaptivity and further code-reduction for business rules-aware UI.

7. ACKNOWLEDGMENTS

Research supported by CTU grant SGS12/147/OHK3/2T/13

8. REFERENCES

[1] Java Unified Expression Language, Aug. 2013. http://juel.sourceforge.net.

[2] E. Bernard. JSR 303: Bean validation, Nov. 2009.

[3] A. Blouin, B. Morin, O. Beaudoux, G. Nain, P. Albers, and J.-M. Jézéquel. Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '11, pages 85–94, New York, NY, USA, 2011. ACM.

[4] M. Blumendorf, G. Lehmann, and S. Albayrak. Bridging models and systems at runtime to build adaptive user interfaces. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '10, pages 9–18, New York, NY, USA, 2010. ACM.

[5] K. Cemus and T. Cerny. Towards effective business logic design. In *Proceedings of the 17th International Scientific Student Conferenece POSTER 2013*, Prague, 16,May 2013. Czech Technical University in Prague.

[6] T. Cerny, V. Chalupa, and M. Donahoo. Towards smart user interface design. In *Information Science and Applications (ICISA), 2012 International Conference on*, pages 1 –6, may 2012.

[7] T. Cerny and E. Song. Model-driven Rich Form Generation. *Information: An International Interdisciplinary Journal*, 15(7, SI):2695–2714, JUL 2012.

[8] L. DeMichiel. JSR 317: JavaTM persistence API, version 2.0, November 2009.

[9] L. DeMichiel and M. Keith. JSR 220: Enterprise javabeans version 3.0. java persistence API, May 2006.

[10] R. Kennard and J. Leaney. Towards a general purpose architecture for ui generation. *Journal of Systems and Software*, 83(10):1896 – 1906, 2010.

[11] R. Kennard and S. Robert. Application of software mining to automatic user interface generation. In *SoMeT'08*, pages 244–254, 2008.

[12] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *In ECOOP'97-Object-Oriented Programming, 11th European Conference*, volume 1241, pages 220–242. Springer, June 1997.

[13] R. Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.

[14] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero. USIXML: A Language Supporting Multi-path Development of User Interfaces Engineering Human Computer Interaction and Interactive Systems. volume 3425 of *Lecture Notes in Computer Science*, chapter 12, pages 134–135. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2005.

[15] M. Macik, M. Klima, and P. Slavik. Ui generation for data visualisation in heterogenous environment. In *Proceedings of the 7th international conference on Advances in visual computing - Volume Part II*, ISVC'11, pages 647–658, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] G. Mori, F. Paterno, and C. Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Trans. Softw. Eng.*, 30(8):507–520, Aug. 2004.

[17] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, Oct. 2009.

[18] J.-M. Oh, Y. S. Lee, and N. Moon. Towards cultural user interface generator principles. In *Proceedings of the 2011 Fifth FTRA International Conference on Multimedia and Ubiquitous Engineering*, MUE '11, pages 143–148, Washington, DC, USA, 2011. IEEE Computer Society.

[19] M. Stoerzer and S. Hanenberg. A classification of pointcut language constructs. In *Workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT) held in conjunction with AOSD*, 2005.