Aspect-driven, Data-reflective and Context-aware User Interfaces Design

Tomas Cerny, Karel Cemus
Czech Technical University,
Charles square 13
12135 Prague 2, Czech Rep.
{tomas.cerny,cemuskar}@fel.cvut.cz

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

ABSTRACT

The increasing use of Web-based applications continues to broaden the user groups of enterprise applications at large. Since ordinary users often equate the quality of user interface (UI) with the quality of the entire application, the importance of providing easy-to-use UIs has been significantly increasing. Unfortunately, designing a single UI satisfying all end users remains challenging. To address this issue, researchers and developers are looking to Context-aware/Adaptive UIs (CUIs) that aim to provide end users with more personalized user interaction experiences. Although multiple proposals have been made, very few production systems provide such malleable interfaces due to the excessive cost of development and maintenance.

In this paper, we propose a technique that aims to reduce development and maintenance efforts of CUI to a level comparable with a single UI. Unlike most of the existing CUI approaches, our technique does not involve an external UI model. Instead, it aims to reflect runtime-information and structures already captured in the application, while extending them to provide an appropriate CUI. With this technique, developers do not design forms or tables directly for each page or panel. Instead they design generic and reusable transformation rules capable of presenting application data instances in the UI while considering the runtime context. To demonstrate our technique and its impact on CUI development and maintenance, we provide a case study. Moreover, we present our experience from its application to an existing production-level enterprise application, with high demands on performance.[1]

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; I.2.2 [**Artificial intelligence**]: Automatic Programming—*Program synthesis*

General Terms

Design, Reliability

Keywords

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

[1]Copyright is held by the authors. This work is based on an earlier work: RACS'13 Proceedings of the 2013 ACM Research in Adaptive and Convergent Systems, Copyright 2013 ACM 978-1-4503-2348-2/13/10. http://doi.acm.org/10.1145/2513228.2513278

1. INTRODUCTION

Despite broad research in the area of Context-aware/Adaptive User Interfaces (CUIs), it is common practice in production applications to design a single UI that serves all types of users and contexts [26]. 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, Kennard et al. [14] 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.

With most existing programming techniques, it is difficult to support adaptive UI features because such approaches capture field-specific information twice, once in the data-model and again as a reference in the presentation that is often specified through a domain specific language (DSL)[23] with weak type safety. In addition, current practices realize multiple UI concerns [11] 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 [16]. 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. Considering that such a component update process is manual, it is most likely to introduce more errors (particularly with no type safety) or omit required component updates, which eventually results in presentation inconsistencies.

Our proposed technique avoids information restatement, as well as supports separation of concerns [11]. The first part is achieved by the utilization of information from an application's data-model and its existing structures that are obtained from the automated code-inspection and by reflecting its meta-model. Such information is then extended and transformed into the UI. To support the second part, this transformation takes multiple steps and bases itself on model-driven development (MDD) [17], generative programming (GP) [9] and aspect-oriented programming (AOP) [16]. Concerns that are in conventional approaches 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. In addition, the transformation process uses a set of generic mapping rules [20] that allows designer to adjust the output as well as to integrate any third-party runtime conditions. From the end-user perspective, the resulting UI dynamically adapts to context and considers all concerns to satisfy expectations. To evaluate our technique, we developed an open-source library called AspectFaces and demonstrate its use in a case study with an enterprise JEE6 application. Furthermore, we provide our experience from a production-level application accessed by users from all around the world with high performance demands.

The main contribution of our approach is the *reduction of information restatement* in UI development and the *separation of UI 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 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 [16] or more generally in General-Purpose Languages (GPL). 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 [18]. In addition to functional decompositions, GP [9] and AOP [16] proposes another way of thinking about program structure. GP proposes the use of a GPL language together with problem descriptions in the form of DSL [23]. The resulting application code is generated at compile time from the DSL specifications that extends the GPL code or produces its variations. In AOP the key unit of modularity is an aspect. Aspects can integrate to GPL modules at runtime or compile time. An aspect enables the modularization of concerns, such as transaction management, that normally cross-cut multiple modules and objects [18].

UI development also employs such decompositions, but data presentation makes the decomposition process more challenging, especially when using DSL 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,

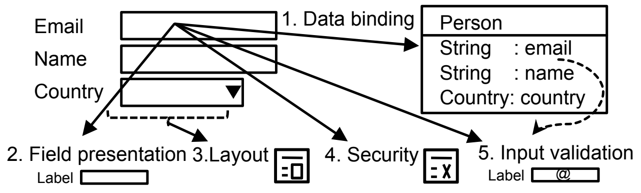

Figure 1. UI form decomposition

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>
```

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 with 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 in Listing 1. When an application adjusts a form layout at runtime based on a given condition, it is possible that multiple cloned variants of the same form must physically exist. For example, consider a slight modification of layout in Listing 1 for a user with wide screen. We would place the `name` to top-left, `country` to top-right and the `email` to bottom spanning both columns, in this case only the layout concern changes, but other concerns are unchanged, but having all the concerns at the same place limits the reuse and we end up with two forms. Next, `Arrow 4` indicates that form fields should consider additional UI conditions such as security or visibility. For example, some fields 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 sim-

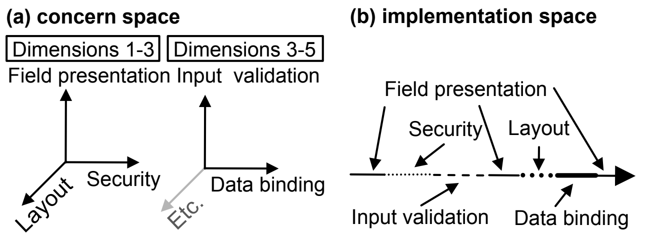

Figure 2. (a) Concern / (b) Implementation space

plified implementation of Figure 1; this JavaServer Faces (JSF) DSL 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 or layout. The reuse of individual concerns in such UI fragments is very limited since it only allows slight variations of concerns. CUI design only compounds the problem since it typically increases the number of concerns.

With the AOP approach [18] this same problem can be seen as an n-dimensional *concern space* that is expressed in the *implementation space* using a one-dimensional language. Unfortunately, orthogonality of concerns in the concern space gets lost (collapsed) 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.

So far we looked over a basic UI design example. Next, let us consider our expectations of an effective UI design. In order to design a CUI effective from the development perspective, we must consider multiple quality attributes. A good CUI design should allow designers to capture the expected functionality but also other non-functional attributes. First, it provides multiple presentations, different layouts, easy addressing and integration of various concerns, adaptive to application runtime context, third-party integration (security), etc. Second, it should be easy to develop and maintain these concerns with low coding effort, while preserving development approaches already known to the developer. Third, a good design should reduce information restatement/duplication across the application and, if possible, mitigate errors caused by UI inconsistency. Fourth, while reducing restated information a single focal point of information should exist to reduce multi-location changes. Fifth, a good design separates tangled concerns [11] into readable code fragments to support their reuse and maintenance.

When we consider conventional approaches and look back to Figure 2, we should note that multiple other concerns may exist for CUI, thus growing the concern space. For example consider concerns such as user's location, data submission error-rate, age, temporal information or layout adjusted to the user's screen size, etc. With no doubt, since the number of concerns in Figure 2 (a) grows and the complexity represented by Figure 2 (b) becomes even greater, it is not reasonable to keep concerns tangled together. Such tangling is directly responsible for increased development and maintenance efforts, diminishing readability, limiting reuse, higher possibility creating errors, etc.

3. RELATED WORK

We split the description of related work onto two parts. The first introduces existing development approaches that could be used to improve the UI design. The second part describes existing UI design approaches and CUI.

3.1 Development approaches

Model-Driven-Development (MDD) [8] argues that a design should involve models that act as a centralized location for information and design decisions. Such a model is then transformed into application code. The idea of MDD is promising regards reduction of restated information, on the other hand we must consider that majority of productions systems involve coding and to transform existing systems to MDD would be expensive. For instance, consider a situation when an application applies pure Object-Oriented Design (OOD) to its backend, while the frontend is designed in a MDD way. Such model would most likely use a custom DSL language for its description. As a result, the model must restate information from the backend. In addition, maintaining the connection between UI and backend requires significant effort. Furthermore, in MDD, problems such as cross-cutting concern applicable to code development are present as well [25]. Although we could use multiple models to capture various concerns and integrate them together, the appropriate generic integration mechanism is missing [29]. Another issue is that MDD suffers during adaptation and evolution management [25]. 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 [8]. Designing a system that considers multiple variations with MDD often leads to a compile time allocation of many states and configurations that can grow exponentially [25]. A runtime solution is less common and such use can degrade performance [30].

Generative Programming (GP) [9] emphasizes specific domain methods and their integration with OOP. GP can be seen as programming that generates source code through domain-specific code fragments or templates to improve designer productivity. Authors of GP [9] define it as a design approach to combine and generate specialized and highly optimized systems fulfilling specific requirements. The goal is to address the gap between program code and domain concepts, support reuse, adaptation, simplify management of component variants and to increase efficiency. It addresses *separation of concerns* [11] that propose dealing with one issue at the time and avoiding code that mixes multiple concerns. It further addresses *parameterization* or *separation of the problem space from the solution space*. Such separation splits the problem space and its domain-specific abstractions and maps them to the solution space with available implementation components (similar to what we show in Figure 2). GP uses DSL with the loss of language generality and emphasizes automatic configuration and generation at the compile time, similar to MDD. While there are substantive differences between MDD and GP, we note many similarities. While the MDD uses models and GP uses OOD and DSL, a model is often designed and described through a DSL, thus the parallel is close. GP directly addresses separation of concerns, which is not the primary goal for MDD. On the other hand, both lack the ability to use runtime information and effectively perform at runtime. The GP integration of concerns does not have a generic format, which exists in AOP.

The features lacking in MDD and GP are addressed by AOP [16, 18, 32]. AOP provides methods that allow us to capture different concerns separately in independent code fragments, as well as enabling runtime weaving. The mechanism to integrate concerns is well described and generic. The problem is decomposed to functional OOD units and aspects, and these weave together to obtain system implementation. In the OOD units, we indicate the possible aspect integration location through join points. These join point are recognized by the aspect weaver, a compiler that integrates aspects to the resulting code/behavior. The product of aspect weaver can have the same execution properties as tangled code, but with the advantage that all the concerns can be defined separately to support readability and maintenance. In [16] authors shows reduction of the total of lines of code (LOC) from 30,000 LOC to 1,000 LOC for the AOP version with the same properties. Compared with AOP, GP [9] has larger scope, involves DSL, and emphasizes the automatic configuration and genericity at the compile time. AOP weaving can be made at compile time or at runtime. The weaving process often involves meta-programming [9, 12] introduced next.

Meta-programming (MP) allows programs and languages to modify their structure and behavior at runtime. Many contemporary, statically-typed programming languages have the ability to describe themselves, which is called Reflection [12]. Reflection is based on an architectural design pattern [5], which gives us an opportunity to inspect classes, their fields and methods while not knowing their names at compile time. This mechanism allows programs to dynamically adapt the program to different situations. MP is a great instrument for code inspection and information extraction. On the other hand, we must consider the impact on performance. To face the performance bottleneck, it is possible to use cache or to involve code generation at application deployment time. Another issue for MP is its testing, because MP programs are not type safe and thus its maintenance becomes complex.

We summarize these approaches in Table 1. We compare the selected abilities such as, whether it applied to runtime, addresses separation of concerns, reduces restated decisions and information restated from existing structures, whether it applies inspection, how well it does for evolution management, if it is adaptable towards changes or existing structures and its compatibility with OOD.

Table 1. Comparison of design approaches

Ability/Approach	OOD	MDD	GP	AOP	MP
Compile time approach	yes	yes	yes	yes	yes
Runtime time approach	yes	slow	no	yes	yes
Separation of concerns	no	no	yes	yes	no
Reduces restated decisions	no	yes	yes	yes	no
Reduces restated information*	no	no	no	no	yes
Model or Code inspection	no	yes	no	no	yes
Evolution management	good	bad	good	good	bad
Adaptable towards changes*	no	no	yes	yes	yes
Transformation based	no	yes	yes	yes	no
Synergy with OOD	-	no	yes	yes	yes
**(from existing code structures/application backend)*					

3.2 UI design approaches

UI design approaches could be divided into three groups, as suggested by Kennard et. al. [15]: approaches using interactive graphical specification, model-based UIs, or language-based UI generation. The first group allows developers to sketch UIs on the screen with the corresponding source code automatically generated in the background. While this approach works for toy applications, it does not consider the maintenance and advanced manual code changes required by enterprise applications. Model-based UIs use a model to describe the UI, which is then transformed to an appropriate UI presentation considering various conditions. Unfortunately, when a model binds to an existing application backend, the model must restate information from it. The language-based tools derives the UIs from the language and domain objects. The advantage is that the UI is always consistent with the application backend, a feature missing from the previous groups, but domain objects are somewhat weak with respect to UI description and this only basic UIs can be derived [8].

In our research, we consider another view of UI design classification by classifying approaches into either *restate-to-extend* or *inspection-based*. The first approach requires that the same information in a system is 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, UI model-based generation tools [19, 25], external models for UI representation [22], and DSL tools [13]. The main drawback of this approach stems from the duplication of source information and maintenance efforts when source information changes.

Inspection-based approaches use existing information accessible by code-inspection. The main effort is placed on the information source that must capture sufficient information to derive a specific concern. Design 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 [7, 14, 15] utilize automated UI generation by applying code-inspection. These approaches inspect previously captured information, build an ad hoc structural model, and transform it to the UI. This simplifies both the 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 [8]. Experience from industrial standards such as Java

Table 2. Comparison of related work

Features	[28]	[22]	[3]	[4]	[19] [24]	[25]	[14] [15]	[8] [7]	[27]
Model-based	-	-	+	+	+	+	-	+	-
Runtime approach	+	+	+	+	-	+	+	+	-
Adaptive UI	+	+	+	+	+	+	-	-	+
Reduces code	-	-	+	-	+	+	+	+	+
Restate-to-extend	-	+	+	+	+	+	-	-	+
Addresses cross-cutting concerns	-	-	+	-	-	+	-	-	+
Code-inspection	-	-	-	-	-	-	+	+	-
Uses enterprise technology standards	-	-	-	-	-	-	+	+	+

EE [2, 10] shows that the data-model already capture additional markup for persistence and validation constraints, and the same approach can be applied also for presentation [8] and security. Our classification allows combining inspection-based approach with model-based design [21], which is not possible with the Kennard's [15] classification.

However, both *restate-to-extend* and *inspection-based* approaches need to deal with information transformation to the UI. Often we see hardcoded transformation rules, such as in interactive graphical specification tools. Generic and configurable rules allow designers wider options and adaptations. Based on [20], generic mapping rules allow easy reuse among systems. In our work, we provide such generic transformation rules while considering aspect-query-based features that involve data structures and field extensions. Furthermore, none of the above approaches directly addresses cross-cutting concerns, although related research on this topic exists for model-based [25, 21] and GP approaches [27].

3.3 Context-aware UI

A basic overview of adaptability and adaptivity is provided by [31]. Both terms refer to knowledge-based self-adaptation, but in the case of adaptivity, it relates to interactive session and adaptability that can be deduced before the interactive session. CUI may address both these features. The idea of CUI is studied in multiple domains. For example, we can see its application to [28] electronic cooking assistants in kitchens to adjust layout, in a hospital navigation case [22] and in a house control unit example [3]. Multiple CUI design methods require the target environment and possible variations of the user interface at design time [19, 24], but in [4] the authors argue that future adaptive systems need to consider runtime information to adapt, while design time approaches are not sufficient. [25] 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, on the other hand they require to restate information. Although, most of the CUI work focus on presentation, adaptability and adaptivity features of UI, they typically apply model-based approaches and restate information from application backend. None of the related approaches provide evaluation regarding runtime performance and production experience, they rarely consider maintenance efforts, and only indirectly address specific concerns such as layout [28]. We provide the summary of selected related work regarding the UI design in Table 2. In our approach, we address all the mentioned features and elements as well as avoid the *restate-to-extend* approach.

4. READ : RICH ENTITY ASPECT/AUDIT DESIGN FRAMEWORK

As shown in the related work, in order to design a CUI with low development and maintenance efforts, we should avoid definition of an additional model that restates information captured elsewhere in the application. Instead we should consider a *code-inspection* approach (MP) and synergy with knowledge about transformations (MDD & GP) as well as to address separation of concerns (GP & AOP).

First, we specify information that we want to reuse such as data structural information and their constraints. All these can be found at the application data-model. Assuming that the data-model design uses OOP and the language supports reflective mechanisms, we gain access to data structures. Besides this we need an access to the application context at runtime. Thus when we need to display data in the UI, we can inspect the given data class and get its structural model, that captures information about the class, its fields and field constraints.

Application context and structural model is then the subject of transformation to the UI. In order to effectively handle both adaptivity and adaptability, the transformation takes place at runtime and uses generic, easy-to-extend transformation rules. In order to design such rules, a single rule instance cannot bind to an individual data or data field but to something more general. In our approach, each rule instance consists of a query part and a suggestion, in the AOP terminology a pointcut and an advice. The query part is an evaluable boolean indicating whether the rule applies for a given context (given data field in given context). If so, the rule's advice is given; if not a next rule in the list is considered. The query can question a data field structural model, application context or both using logical and arithmetical operations. The advice provides the integration DSL template that is used for the data field.

A collection of customizable DSL templates is associated with the transformation rules. Such template uses the target presentation language and integration rules in it, to integrate additional concerns. An integration rule again consists of a pointcut and an advice. The pointcut is uses the same query constructs to question the data field structural model and context. An advice is different; it is a DSL content that can reference the structural model properties or context variables. All integration rules are considered for given template if a rule pointcut holds, then the advice content embeds to the template or resolves the reference to the structural model (such as field name, type, etc.). The result of the template interpretation is a UI code fragment in the target DSL language representing given data field considering all concerns, but layout.

Right after all data fields process through the transformation, then a proper layout template integrates. This process is similar to XSLT. The resulting output is a DSL fragment reflecting data, context and integrates all considered concerns. The last part of our approach is runtime integration of the resulting DSL code to the application UI. This involves the DSL code compilation and UI embedding.

4.1 Introduction to READ conceptual model

Our framework involves AOP and, in order to describe an AOP framework, [32] 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 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 we identify two sources of join points: the 1) structural data model and 2) application runtime context (a subset exposed to the READ process). A structural data model provides entity and field names, data types and field meta-instructions with their parameters [10, 2]. Application runtime context can consist of any kind of information, such as user access rights, geo-location, local context for presentation, device screen size, etc. We could even count user error-rate throughout the application interaction and based on that show a tooltip or help upon page load. Both sources provide us join points that can be considered in the transformation process. More specifically these join points allow us to support generic/reusable transformation rules.

The AOP terminology deals with two types of join points [32], static and dynamic. *Static join points* can be characterized as a location in the source code; the selection criteria refers to static structure, and it is known at compile time where and how to enhance the code for all invocations. *Dynamic join points* correspond to elements in code, but at the same time to a runtime condition that specifies the selection only to certain invocations. While the application context corresponds to the dynamic join points the structural data information consists of both types of join points. For instance, the field name is static, while field access rights denoted by field annotation can be dynamic.

The *pointcut language* defines the query language to select a subset of join points. READ uses an expression language known 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 elements of the structural model (from the field perspective) and to dynamic context variables that are populated by application designer. The pointcut language can query all information in the EL context. It is also possible to define custom utilities or functions that integrate third party libraries and pass them to the context and thus expose them as dynamic join points. The language uses both the *state-based* and *specification-based* constructs [32]. Later, in the next section we show how to access elements for the structural model and context from the EL.

The *adaptation* mechanism uses the above described join points, and based on their association to a particular data field or global context it selects an appropriate UI transformation rule instance that suggests an integration template. An integration template applies the same join points for integration rules. In both cases, pointcuts query the join points to get an advices, either for the transformation or concerns integration. We give an example of both rules later. The integration template uses an aspect language for concern integration but at the same time uses the target UI language. The layout integration is the last part of the adaptation mechanism. It is similar to the integration template in that it uses a DSL language from the target domain language to describe the layout and an extra markup to locate specific or anonymous fields in the template.

In order to an a new concern to the system, we either need to expose it to the READ context, or to extend the data structure through new annotations. This way the novel concern becomes accessible by EL, thus by both the transformation or integration rules.

4.2 READ lifecycle

Next, we briefly look at the READ lifecycle in Figure 3 that denotes the main stages (a-f). In the UI, we aim to display a given data instance (a) in the target UI language. In order to do that, we use a custom component that is associated with a specific component handler (b,c) and the displayed data instance reference. The responsibility of such a handler is to provide the content for the component. Thus this handler is the connection between the target UI language and integration of our approach. The custom component takes as an input the data instance and considers other context information. For instance, the context can be an indication that the aimed content is a read-only presentation, fields named *"notes"* are ignored, etc. First, the handler aims to get the data structure, the data structural model. Either, this structural model is found in the cache from a previous use or the data instance goes through an MP inspection (d) and the result is passed to the cache. A cloned instance of the structural model, which is the result of the inspection (d1), is interpreted in a given context using the Annotation Driver Participant Pattern (ADPP). This may result in modification of the structural model. Such a context-aware structural model is then passed to the transformation phase (e) together with the context. In the above sections, we mentioned three phases of transformation. Each data field from the context-aware structural model of given data is the subject of transformation and concern integration (e1*). This results in a UI code representation in the target UI language for each field. After all fields process the layout is integrated (e21) receiving the entire data UI representation as result. The last stage interprets the resulting UI fragment and integrates it to the UI (f).

4.2.1 READ UI integration

Consider the process of designing a web page where we want to display application data in the main panel. Normally, such a main panel contains code similar to Listing 1 to describe the data. Instead a custom component is used as shown in Listing 2. A component prefixed *"af"* takes as an attribute a reference to a data instance accessible through a controller (in our case called a bean and a local con-

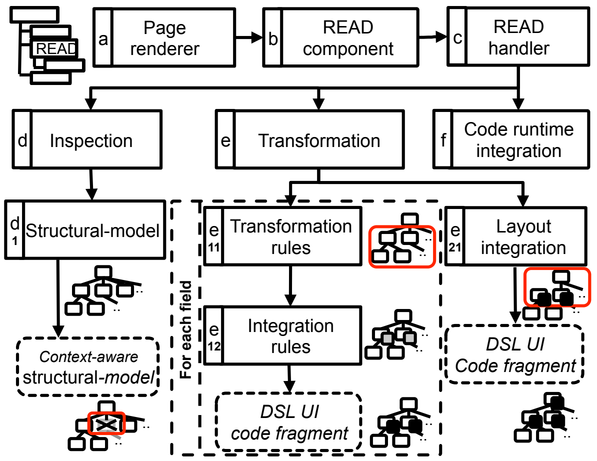

Figure 3. READ lifecycle

Listing 2. Example use of READ UI component

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

text). This component is associated with custom handler that pushes the local context to be considered in the READ context and issues the phases described in Section 4.2 to receive the CUI for the given data instance.

4.2.2 *Inspection phase*

The inspection phase is rather complex; thus we provide more details. It audits classes of the data-model (entities). It specifically looks for class name, class restrictions, its fields and field constraints. While aiming to build on existing industry standards, we look into the following profiles for persistence and input validation. Java EE considers Java Persistence API data-model profile [10] to support object-relational mapping and Java Beans Validation [2] to validate input of the persistent objects. Both of these standards are applied to existing systems, and we consider them in the inspection. Model-based profiles reflecting the above data-model extensions were introduced in [8]. The same approach can be further extended for role-based access control and for presentation [8]. Table 3 shows the class structure and field elements and a subset of selected extensions applicable to data-model class fields. The table describes extension names, denotes their applicability and also highlights the name of a variable under which it is accessible as a join point. The inspection phase considers all such extensions and it is further possible to consider other custom ones. The result of the inspection is a structural model, a three-level composite structure reflecting the class-level, field-level constrains or extensions. The structural model provides these properties through the variable name (the right most column in Table 3). Furthermore, in this phase, it is possible to apply runtime context to modify the structural model. For example, it is possible to locally modify structural model fields based on a given condition such as ignore a field, change field constraints or to expose a new variable/object in the context and make it available as a join point.

Table 3. Subset structural model elements accessible as join points

Extension	Description	Data type	Context variable
Class-level attributes			
-	class name	-	entity, Entity
-	full class name	-	fullClassName
Field-level attributes			
-	field name	-	field, Field
-	field type	-	dataType
Field-level constraints			
1. Persistence profile			
Column	DB table column props.	Any	notNull,required, maxLength,unique..
joinColumn	DB table column props.	Any	notNull,required, unique..
Temporal	Date, Time, TimeStamp	Date	temporal
	..		
2. Validation profile			
Length	Value length in the range	String	minLength, maxLength
Min, Max	Value in the range	Number	min, max
Email	Match email	String	email
Pattern	Matches the reg-exp	String	pattern
Future, Past	Future/Past date	Date	past, future
NotNull	Not null value	Any	required,notNull
NotEmpty	Not empty value	Any	required,notEmpty
	..		
3. Presentation profile			
UiLink	Web link expected	String	link
UiText	Long text expected	String	text, cols, rows
UiParam	Any Param expected	Any	param (name, value)
UiHtml	Html expected	String	html
UiPassword	Secret text expected	String	password
UiType	Type of widget to use	Any	type
UiOrder	Order in view	Any	order
UiTableOrder	Order in table	Any	tableOrder
UiIgnore	Ignore field in UI	Any	ignore
UiPattern	UI Script regular expr	String	uiPattern
UiProfiles	To support grouping	Any	Profiles
	..		
4. Access control profile			
Restrict	Third parti restriction	Any	restrict
UiUserRoles	Values specifies user role	Any	roles
	..		

4.2.3 *Transformation phase*

The transformation phase consists of three parts. This section provides details to build the whole picture. We have introduced how to apply a READ component to the UI DSL, how to receive the data instance, and process the inspection. The result of this is a context-aware structural-model of the data instance. Such a structural model is accessible to the transformation through a EL context using variables described in Table 3. The application context and any third-party elements or properties specified together with the READ component or in its handler are passed to the EL context. For example, consider the entity described in Listing 3 (it uses annotations [10] and [2]) and context specified in Listing 2. The structural model reflects information about the data, its fields and field properties. The settings in Listing 2 modify the model instance, and in our case fields password and notes are ignored. It also exposes the aim to edit the data to the context, as well as specific layout to use for this particular UI page.

The first stage applies transformation rules to data structural model fields. An example of a subset of such rules captured in a DSL is shown in Listing 4. Consider the *first* *name* field from the Listing 3. Based on the type, we use the String group of the rules, and since none of the rule pointcuts (expr attribute) apply, we use a default advice a *textTemplate*. For *email*, we pick a *emailTemplate* since the second pointcut applies. The pointcut could use any variable available in EL context (consider context variables in Table 3 from the structural model, or any variable exposed to the component handler). For example, we could ask whether a field of type String is short and required and whether it is Monday and user is from Prague. Such pointcut would look like this:

```
maxLength<100 and required
and timeUtil.getDayName() eq 'Monday'
and locationUtil.city.toLowerCase() eq 'Prague'
```

Each field from the structural model of the given data instance gets advice from the transformation rules. The advice is a DSL template that describes a basic presentation in the target UI language, with integration rules to integrate various concerns, such as binding, help, validation, etc. An example template is shown in Listing 5. Note that this template consists of many references to the structural model through the context variables. These references are part of the integration rules. In order to understand the mechanism, we show three variants of integration rules in Listing 6. It shows (a)-full/ (b)-brief/ (c)-shorten version of integration rules. It integrates join points from a given field. The full version separates the pointcut and advice part; when the pointcut evaluates to true, then the body applies. Brief version provides the same result but needs less code. The shorten version fits to common cases and needs the least code.

The last phase is layout integration. Layout is given by the READ component or deduced dynamically using the handler. This process is similar to XSLT, but it can express anonymous fields and iterate over repeating layout pattern. Consider Listing 7 that shows an HTML table decorating data fields. The layout has a repeating code pattern of two columns for anonymous fields with up to 100 iterations, and a reference to an explicit field called *notes* that spans over two columns in the last row. If fewer fields exist than specified, only the given amount applies. The specific fields take precedence in resolution. The result is a CUI code fragment that the READ handler integrates to the UI.

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 [10] for the ORM, which extends the class model with additional markup. Similarly validation [2] can be added. Generalization of such extensions and further enhancements are suggested by [8]. READ inspection uses all of this information for the structural model composition and for join points. Besides the data model, READ can also integrate business rules defined in the above layer. Preliminary work in [6] shows that business rules can be inspected and their definitions reused. This can be integrated into the READ context.

Listing 3. 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 @Email
  @Length(max=100) @Column(nullable=false, length=100)
  public String getEmail() { return email; }

  @UiOrder(3) @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 4. Example transformation rules

```
<mapping>
 <type>String</type>
 <default tag="textTemplate" size="20"
   javaPattern="" minLength="0" maxLength="255" />
 <var name="Person.username" tag="emailTemplate"/>
 <cond expr="${email == true}" tag="emailTemplate"/>
 <cond expr="${link == true}" tag="linkTemplate"/>
 <cond expr="${maxLength>255}"tag="textAreaTemplate"/>
</mapping>
```

Listing 5. Example template for inputText widget

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

Considering common development approaches, 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 build 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 transformation rules that generalize mapping among entity fields and presentation widgets. Transformation rules are generic and can be reused among projects. The developer then designs integration templates that are used by the READ weaver (component). These templates are also generic and can be reused. While developing such templates is time-consuming, 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 main benefits? First of all, the system presentation reflects the actual state of the software system. All data definitions, 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

Listing 6. Pointcut strategies for templates

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

Listing 7. 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>
```

an individual concern is easy to locate and modify. Third, READ reduces errors because the entity becomes 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 efforts since a new entity presentation does not require any coding. In case a new presentation is needed for a given field, it is possible to define new transformation rule or design a new template. Fifth, READ naturally supports adaptive UI design because it evaluates conditions at runtime and separates concerns. Sixth, READ is open for integration with third party frameworks through the context or data-model extensions. READ templates can integrate any DSL. 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, etc.). Seventh, READ does not bind the developer to a single-use approach; alternative approaches can be applied at the same time.

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

5. EVALUATION

In this section, we provide an evaluation of our approach. First, we consider a UI that provides a single presentation and compare the manual approach with READ. Next we consider UI extensions to support adaptive features such as adjustments to access rights, users location, age, capabilities or screen size. In this evaluation, we compare the development costs for both approaches. We also consider a few maintenance scenarios. In the second part of the evaluation, we consider runtime performance. Third, we evaluate an existing production system that uses READ and provide our evaluation statistics. In the evaluation, we consider an existing application, which is a registration system for the worldwide competitions, the ACM-ICPC registration system (available at http://icpc.baylor.edu)

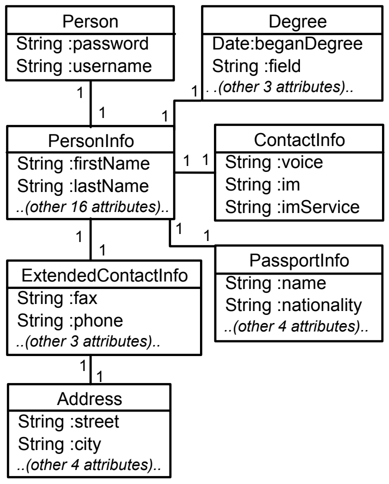

Figure 4. Evaluated application data-model

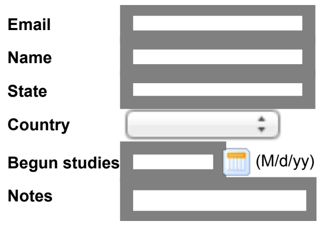

Figure 5. Sample simple UI Form

5.1 Development and maintenance impact

In this section, we consider a subsystem of an existing ACM-ICPC system used for the registration of users and user account management. The considered subsystem has the data-model illustrated in Figure 4. For brevity, the class attributes are abbreviated, and the class model does not list all attributes. 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, 10]. 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 with a single UI. The UI part of the application contains search with result listing plus a detail and modification page. The presentation covers the entire data-model in Figure 4. Illustration of a page fragment, a form, is shown in Figure 5. 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 development 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 of application configuration. The type-unsafe XML presentation exhibits 564 occurrences of restated information from the data-model and its constraints [2, 10]. Next, we implement the same application using READ. The data instance source code is extended with additional presentation marks [8] extending the field

Table 4. 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

constraints (see example in Listing 3). The main difference is that READ composes components presenting data. They combine information from data instance inspection, transformation 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 XML. This results in 1530 LOC of Java, including the additional data-model marks and a 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 consider the maintenance impact. In the manual approach, we are directly responsible for restating information from data model in the UI, where READ handles this for us. With READ we avoid inconsistency and errors, while reducing development time. Even greater code reduction effect can be achieved on larger projects. Note that presentation templates and transformation rules can be reused among projects. In this 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 4 (denoted by simple features). The aspect weaver itself is not included in the evaluation because it is a generic, reusable and external library (reasoning in [16]).

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. 6-8).

The application following the mainstream development applies field restrictions, such as user rights or locations 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, which includes 240 UI conditionals and 6768 restated information from the data-model. Consider that with this approach, developers follow the implementation in Figure 2 (b).

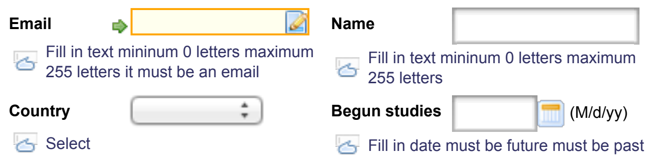
Email
Fill in text mininum 0 letters maximum 255 letters it must be an email
Name
Fill in text mininum 0 letters maximum 255 letters
Country
Select
Begun studies (M/d/yy)
Fill in date must be future must be past

Figure 6. Sample form for confused student

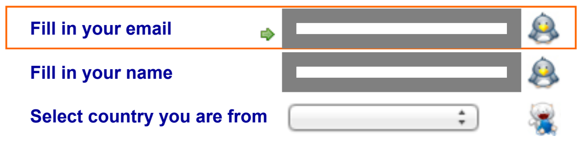
Fill in your email
Fill in your name
Select country you are from

Figure 7. Sample form for child

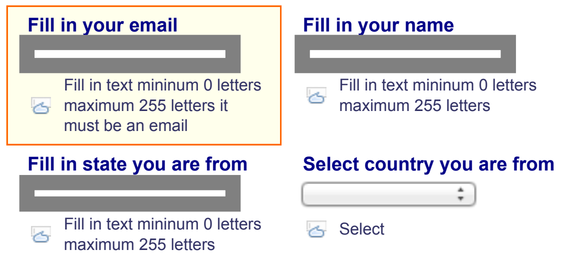
Fill in your email
Fill in text mininum 0 letters maximum 255 letters it must be an email
Fill in your name
Fill in text mininum 0 letters maximum 255 letters
Fill in state you are from
Fill in text mininum 0 letters maximum 255 letters
Select country you are from
Select

Figure 8. Sample form for elderly

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 weaver 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-right restrictions are captured in the data-model, which reduces them to 20. Furthermore there are no occurrences of restated information in the XML, although field names can be explicitly captured as attributes the UI component Listing 2 to ignoring specified fields. The overall summary of the evaluation is provided in Table 4. 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 code readability, reuse and maintenance more effectively.

Next, we evaluate basic maintenance scenarios. With manual development, the UI is fragile because of its coupling to the data-model in the environment with type safety. 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 addressed by text search. With READ approach, there are fewer UI references to the data elements; thus it does not require many UI corrections. When we want to globally

Table 5. Performance comparison

	Avg. page load time	Std. deviation
Manual approach	545ms	47
READ approach	539ms	41

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 require to application of conditionals in the UI or at controllers. Since multiple presentations exist for a single field, this can impact a significant amount of UI code. In READ, such change takes place at controllers and then only one time in the data-model, in a single location. The addition of a new form layout may require a new copy of the form with tangled layout, in the common approach. In READ, the layout is a separate fragment, thus only a new layout template is designed.

For the performance evaluation, we consider 5 forms with total of 21 fields. We evaluate the time needed for the page to render using both the manual and READ approach as shown in Table 5. 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.

5.2 Case Study : Production Experience

In order to demonstrate a large scenario, we provide a study that applies the READ framework in production use. The entire user registration and contest management system described in Section 5 is used. The goal of this study is to show applicability of READ to production environment, its impact on development and maintenance, statistics resulting from the approach, and generalization of its impact.

A subset of this system is evaluated in Section 5.1, using a prototype applying various adaptive features. In the production system we only consider single presentation and multiple screen layouts. The entire application is complex and builds on a large data-model (70 data entities). UI development and maintenance makes up a significant portion of the overall development effort. Recently, the application migrated to a new version that includes changes of the presentation framework. Thus we changed all the UI components in the entire application. Since both versions apply READ for the UI forms, only a few changes were required to support new form components. For the entire application, only 25 integration templates existed; these were reused for all forms in the application. Changes to support new form widgets took place in these templates. There were no changes needed for the other concerns (e.g., layout templates, or transformation rules). This migration was done in a very short time, compare to what it would be in the manual case. If had used the manual approach, each form would combine multiple concerns and thus the change would impact up to 21451 LOC of XHTML. Instead with READ, we could solely focus on a single concern (a presentation), which is a change in the UI templates, and this impacts only 288 LOC. While porting forms required little time and effort, the migration of the UI tables, which did not apply our READ approach, took considerably longer because it entangled multiple concerns that were reused in many locations.

Table 6. Case study summary

Application	Java	77394 LOC	
	XML	2380 LOC	
	XHTML	41473 LOC	
	Generated UI	equiv. to 21451 LOC (XHTML)	
Estimate	Savings on restated inf. in UI		15592
UI	Data entities	63 (70 total in the application)	
	Data fields	473	
Average	Entity	7.5 fields	per UI form and layout
	Entity in UI	82.5 restated inf.	
	Entity in UI	113 LOC	
	Field in UI	15 LOC	

Our code measurement in the production system provides the following results. Out of the 70 entities in the data-model, 63 of them are referenced in the UI as forms. All of these forms are generated at runtime based on data inspection. These forms are rendered in three different layout widths according to the user's screen size. In order to apply the READ approach, we must define 28 transformation rules (only 101 LOC), integration templates for UI components (288 LOC) and layout templates (367 LOC). We also need to apply additional 545 annotations to the Java classes. The view part of the application, including XHTML and XML, consists of 41473 LOC and 2380 LOC. The entire Java code has 77394 LOC. The approach brings the reduction in UI forms for 63 entities in three different layouts, which represents up to 21451 LOC of XHTML code. This represents approximately 32% of the entire UI code for the application.

The following is the summary on these measurements. There are 63 entities, with total 473 fields, that are represented in the UI. Each field may have multiple constraints defined by field annotations for object-relational mapping, validation, security or presentation (see Listing 3). This represents 9-13 references per field in the UI component (see the Listing 5); the exact number depends on a particular widget type and the field. We also measure the average number of fields in the UI form per a given entity. The result shows that our system has the mean value 7.5 fields per entity (median 6) with a standard deviation of 4.85. When counting that an UI widget has approximately 11 references to the data-model, it results in 82.5 occurrences of restated information in the XHTML per the average data entity in a single layout form. Consequently, READ prevented an estimated 15592 occurrences of restated information in the application UI. The measured statistics to render data entity in a UI form in a single layout results in 113 LOC with standard deviation 63.77. This is caused by the deviation of class fields. Thus each time we use READ in the UI we save around 113 LOC. On average this represents 15 LOC for an individual field in UI for a single layout.

The measurements are summarized in Table 6; it shows that the use of READ framework reduces the amount of UI code. Significant portion of the UI code is generated. Doing this manually require us to handle significant coupling and many occurrences of restated information; therefore, future evolution management would result in high maintenance efforts.

Table 6 gives our estimate on an average entity its field count, UI references, and LOC required for UI presentation with contemporary approach. The project statistics shows that the UI part of the system is significant, which confirms the estimate from [14]. Regarding the performance, there is no performance reduction exhibited and the community has reported no performance issue. The broader variety of adaptivity of the system to the international audience is currently under development.

6. CONCLUSION

Despite many benefits of CUIs, few production systems support employing them. The reasons behind this include the excessive cost of CUI development and maintenance as shown in our case study. We provide an approach that considers existing standards for application frameworks, aspect-oriented programming and employs code-inspection to face the complexity and efforts related to CUI design. Our READ technique considerably reduces the cost involved in the development of CUIs. While time-consuming to initially develop, all the work related to templates and transformation rules amortizes over the size of the software application. Our approach is implemented in a production-level library, called AspectFaces. It is currently used in production at the ACM-ICPC system.

In the future, we plan to generalize our approach to inspect and reuse application business rules. Our preliminary results show that such an approach will provide more options and variety of adaptivity and further code-reduction for business rules-aware UI.

7. ACKNOWLEDGMENTS

This research was supported by the Grant Agency of the Czech Technical University in Prague, grant No. 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] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[6] K. Cemus and T. Cerny. Aspect-driven design of information systems. In *SOFSEM 2014: Theory and Practice of Computer Science*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Novy Smokovec, High Tatras, Slovakia, 25, January 2014.

[7] 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.

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

[9] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

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

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., Oct. 1976.

[12] I. R. Forman and N. Forman. *Java Reflection in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[13] M. Karu. A textual domain specific language for user interface modelling. In T. Sobh and K. Elleithy, editors, *Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering*, volume 151 of *Lecture Notes in Electrical Engineering*, pages 985–996. Springer New York, 2013.

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

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

[16] 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.

[17] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

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

[19] 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.

[20] V. López-Jaquero, F. Montero, and F. Real. Designing user interface adaptation rules with t: Xml. In *Proceedings of the 14th international conference on*

Intelligent user interfaces, IUI '09, pages 383–388, New York, NY, USA, 2009. ACM.

[21] M. Macik, T. Cerny, J. Basek, and P. Slavik. Platform-aware rich-form generation for adaptive systems through code-inspection. In *Human Factors in Computing and Informatics*, pages 768–784. Springer Berlin Heidelberg, 2013.

[22] 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.

[23] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[24] 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.

[25] 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.

[26] 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.

[27] M. Schlee and J. Vanderdonckt. Generative programming of graphical user interfaces. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '04, pages 403–406, New York, NY, USA, 2004. ACM.

[28] V. Schwartze, S. Feuerstack, and S. Albayrak. Behavior-sensitive user interfaces for smart environments. In *Proceedings of the 2nd International Conference on Digital Human Modeling: Held as Part of HCI International 2009*, ICDHM '09, pages 305–314, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] J.-S. Sottet, G. Calvary, J. Coutaz, and J.-M. Favre. A model-driven engineering approach for the usability of plastic user interfaces. In *Engineering Interactive Systems*, pages 140–157. Springer, 2008.

[30] J.-S. Sottet, G. Calvary, and J.-M. Favre. Models at runtime for sustaining user interface plasticity. In *Models@ run. time workshop (in conjunction with MoDELS/UML 2006 conference)*, 2006.

[31] C. Stephanidis, A. Paramythis, D. Akoumianakis, and M. Sfyrakis. Self-adapting web-based systems: Towards universal accessibility. In Waern, editor, *In 4th ERCIM Workshop on "User Interfaces for All"*, 1998.

[32] 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.