Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

On Concern-separation of Data Presentations in User Interfaces

Doctoral Thesis

by

Tomáš Černý

A dissertation thesis submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

PhD programme: Electrical Engineering and Information Technology
Branch of Study: Information Science and Computer Engineering

Prague June 2015

Thesis Supervisor:

Doc. Ing. Jan Janoušek, Ph.D.
Department of Theoretical Computer Science
Faculty of Information Technology
Czech Technical University in Prague
Thakurova 9
160 00 Prague 6
Czech Republic

Thesis Co-Supervisor:

Prof. Michael "Jeff" Donahoo
Department of Computer Science
School of Engineering & Computer Science
Baylor University, Texas
One Bear Place, P.O. Box 97356
Waco, TX. 76798-7356
United States of America

Abstract

The User Interface (UI) of software systems plays an essential role in usability. Users place ever-expanding expectations for a good UI, e.g., dynamic responsiveness to partial input. Unfortunately, development frameworks typically treat UI development as a completely independent unit, ignoring the global application perspective. This results in information from the data definitions to be restated in the UI description (i.e., vertical repetition). Specification of expected type, client-side validation, and even input widget selection represent restated information that must be maintained over the evolution of the application, often with little language support as UI descriptions usually lack type safety. In addition, this UI description often tangles a multitude of concerns such as layout, data binding, input validation, etc., which makes it hard to reuse a particular concern. This tangling results in a particular concern definition being distributed to multiple UI fragments, thus changing a given concern requires significant effort. Making matters worse, systems are now expected to personalize for individual users (e.g., expert vs. novice), contexts (e.g., country-specific postal information), and even physical presentations (e.g., desktop vs. mobile). UI development solutions often fail to provide the flexibility to easily handle variations of the UI for such personalization. Current solutions with tangled concerns result in multiple, substantially-similar UI fragments that repeat information on input type, constraints, etc. and differ only in details (i.e., horizontal repetition). Clearly, in such a situation modification of a particular UI concern becomes tedious. The high-level of replication and restatement makes it error-prone and fragile towards application changes. The negative impact of the repetition and concern tangling goes beyond higher development and maintenance costs. UI descriptions bloated with repetition require more server-side processing and bandwidth to deliver to clients, negatively impacting UI responsiveness and server scalability. Even worse, such tangled information disables caching capabilities and reuse at the client-side.

This thesis considers existing UI software design approaches from the perspective of UI development, maintenance, responsiveness, delivery, integration with existing enterprise frameworks, ability to support personalized presentations reflecting user and system context, etc. We show that conventional approaches only partially or ineffectively address the above concerns for data presentations. To deal with these sorts of problems, we introduce a novel approach that addresses separation of cross-cutting concerns. We investigate approaches from Model-Driven Development (MDD) and leverage existing approaches to decrease information restatement at the code and delivery level. Our approach reduces the frictions related to UI development and maintenance efforts, while providing easy integration of extended UI capabilities such as context-awareness. The maintained separation of concerns applied to the remote client UI delivery extends the benefits to improved UI responsiveness, resource reuse, and reduction of repetitions as well as performance improvements in server-side UI description processing.

We demonstrate the efficacy of our approach through various case studies. Finally, we present an evaluation of our approach based on its production-level use in a large enterprise system deployed for the ACM International Collegiate Programming Contest (ACM-ICPC), used by tens of thousands of participants from more than 2,534 universities in more than 101 countries.

Keywords:

User interface, Metaprogramming, Aspect-Oriented Programming, Maintenance, Model-Driven Development, Model Transformations, Separation of Concerns

As collaborator of Tomáš Černý and a co-author of his papers, I agree with Tomáš Černý's authorship of the research results as stated in this dissertation thesis.

.....	.....
Michael Donahoo	*Eunjee Song*
.....	.....
Miroslav Macík	*Jan Janoušek*

Acknowledgements

My sincerest thanks and appreciation to those who have made my life richer from a wide range of perspectives. This work would never exist, unless I had studied abroad at the Baylor University in Waco, Texas. At the same time, to become a student at Baylor, I had to have solid technological background, which I received from the Czech Technical University in Prague. Many people put their trust in me and this acknowledgement is just a small piece of my gratitude. I would first like to recognize Boba Mannova, Michael Jeff Donahoo and William Bill Poucher for making my studies and life rich, as well as for finding me all the possible opportunities in order to reach my fullest potential.

This thesis would never be written without great support, advices, mentoring and energizing from Michael Jeff Donahoo, who supervised me through my masters and doctoral studies and found time for consultations, brought new ideas and pushed me towards the right direction. I would also like to extend my deepest appreciation to my adviser Jan Janousek, who is perhaps the most energetic and positive person I ever met.

My gratitude further extends to all my research colleagues, co-authors and mentors. Namely I would like to recognize Eunjee Song, Miroslav Macik, Pavel Slavik, Karel Cemus, Lubos Matl.

Dedication

To my lovely wife and wonderful daughter, my colleagues and friends.

Contents

List of Figures

1

Introduction

No matter how beautiful, no matter how cool your interface, it would be better if there were less of it.

-Alan Cooper

Father of Visual Basic

Software engineers always aim to design systems that not only function well and attract users but also require low development and maintenance efforts and easy integration. A User Interface (UI) plays the role of the ambassador of an application, as it is the portion of the system experienced by users. In order to do this, the UI provides mechanisms to control events, submit/retrieve data to/from the system as well as influencing how to display information. Given this, the UI is one of the most critical parts of an application; therefore, the UI graphical design aims to make user interaction simple and efficient for all tasks.

In the area of UIs, we often see complex and sophisticated features that greatly attract users, but at the same time negatively impact development efforts. This work deals with the UI design from the software engineering perspective and considers "how it works,"

> **To visually design a data form is one side of the coin**

Figure 1.1: Conventional-manually made web form

Figure 1.2: Automated data-driven web form

which components are involved and what is the interaction of particular components, what is the complexity regarding comprehension, development, maintenance[1], reuse, module responsibilities, performance, extensibility, efficacy regards capturing various system aspects, concerns or goals, as well as other design considerations.

For a demonstration consider the following motivational examples. A rather simple UI fragment, such as these shown in Figs. 1.1 and 1.2, might require considerable development and maintenance efforts. The source code behind these forms might be quite complex and hard to read due to various goals, amount of information or usability strategies considered all at the same time by developer. Clearly, both of the forms from Figs. 1.1 and 1.2 are visually the same, although their design, i.e., the structure of their source code, may significantly differ.

Consider the design used in contemporary UI frameworks, such as Swing from Java Standard Edition (Java SE), JavaServer Faces (JSF) [2] from Java Enterprise Edition (Java EE), etc. Such *conventional design* (see Chapter 2) involves basic building blocks in the form of pre-defined components to represent information, knowledge and

[1]Most of these terms are defined in Section 2.1

data stored in the system. Usually, it expects the developer to describe all the information involved in a particular UI page all together. This makes it easy to build the whole picture. On the other hand, there is no explicit boundary among different information purpose-sets. The information usually tangle together, which makes it difficult to maintain [A.4].

The pre-defined components connect to a component tree that represents a particular UI fragment; a design pattern, called composite [3], is often used to represent such a structure. For instance, the UI fragments in Figs. 1.1 and 1.2 may be represented as a component tree, which contains a form component connected with three panel components that consist of different data-presenting components, called widgets. The tree can be expressed and described using a General-Purpose Languages (GPL), such as Java, or it may involve Domain Specific Languages (DSL) description such as the one used by JSF. Although it is easy to describe a particular UI through a tree, the problem is that this format of description is limited to a particular situation.

When considering multiple system concerns[2] that impact the UI presented to the user, it might be necessary to capture certain concerns in a way that they tangle across multiple components or through other concerns. This decreases the ease of comprehension, reuse of components and increases the development and maintenance efforts. The situation becomes worse when some concerns vary over time or at runtime based on application/user context. A component that tangles together multiple concerns becomes specialized; its volume may expand with repetitions as the result of tangling and its reuse is limited. Thus a situation with varying concerns usually results in duplication of the original component and slight modification. Unfortunately, this results with two or more components that must be maintained.

Consider a conventional design applied to Fig. 1.1 in the dimension of information arrangement that involves the presentation concern, as well as security, input validation, data-binding and layout. Each UI component that considers this information must capture them in the source code to make it self-descriptive. This results in source code

[2]How to interpret system concerns? The literature [4] suggests that *information set* that affects source code is known as *concern*. Dijkstra [5] recognized the importance of separation of concerns as an effective technique to organize thoughts. Tangled concerns in design or in source code worsen the readability, because an individual *concern* spreads throughout the code and cross-cuts other *concerns*. Thus localization of a single *concern* in the code becomes non-trivial. In Fig. 1.3 notice the concerns tangled together. Obviously a design that tangles various *concerns* together becomes complex, hard to read or maintain, and it often increases the volume of code because of low reusability of given *concerns*. Further demonstrations of tangling concerns will be given in this work. Another illustration of tangling is provided by Fig. 2.13.

Less coding efforts and loose coupling is the key

Listing 1.1: Illustration of a JSF code behind the conventional design of form from Fig. 1.1

```
<p:panel>
 <table><tr><td>
   <util:inputUsername label="#{txt['user.username']}"
      value="#{reg.user.username}"
      required="true" size="30" maxlength="50"
      title="#{txt['txt.user.username']}"
      edit="#{reg.edit}" id="username" />
  </td></tr><tr><td>
   <util:inputPassword label="#{txt['user.password']}"
      value="#{reg.user.password}" required="true"
      title="#{txt['txt.user.password']}"
      maxlength="255" rendered="#{empty reg.user.id}"
      edit="#{reg.edit}" id="password" />
  </td></tr><tr><td>
   <util:checkBox label="#{txt['user.confirmed']}"
      id="confirmed" rendered="#{security.admin}"
      edit="#{reg.edit}" value="#{reg.user.confirmed}"
      title="#{txt['txt.user.confirmed']}" />
  </td></tr></table>
</p:panel>
<p:panel>
 <table><tr><td>
   <util:selectMenu label="#{txt['personInfo.title']}"
      edit="#{reg.edit}" collection="#{enums.title}"
      value="#{reg.user.personInfo.title}" type="enum"
      title="#{txt['txt.personInfo.title']}"
       id="title" />
  </td></tr><tr><td>
   <util:inputText label="#{txt['personInfo.firstName']}"
      value="#{reg.user.personInfo.firstName}"
      edit="#{reg.edit}"  required="true" size="30"
      title="#{txt['txt.personInfo.firstName']}"
      maxlength="255" id="firstName" />
  </td></tr><tr><td>
   <util:inputText label="#{txt['personInfo.lastName']}"
      value="#{reg.user.personInfo.lastName}"
      edit="#{reg.edit}" required="true" size="30"
      title="#{txt['txt.personInfo.lastName']}"
      maxlength="255" id="lastName" />
  </td></tr><tr><td>
   <util:inputText label="#{txt['personInfo.email']}"
      value="#{reg.user.personInfo.email}"
      required="true" size="30" email="true"
      title="#{txt['txt.personInfo.email']}"
      edit="#{reg.edit}" maxlength="255" id="email" />
  </td></tr><tr><td>
   <util:inputNumber label="#{txt['personInfo.age']}"
      value="#{reg.user.personInfo.age}" size="10"
      required="false" edit="#{reg.edit}" maxlength="50"
      title="#{txt['txt.personInfo.age']}" id="age" />
  </td></tr><tr><td>
   <util:inputDate edit="#{reg.edit}"
      label="#{txt['personInfo.born']}"
      value="#{reg.user.personInfo.born}"
      title="#{txt['txt.personInfo.firstJavaProgram']}"
      required="false" id="firstJavaProgram" />
  </td></tr></table>
</p:panel>
<p:panel>
 <table><tr><td>
  <util:inputText
      label="#{txt['branch.organizationName']}"
      value="#{reg.user.branch.organizationName}"
      edit="#{reg.edit}" required="true" size="30"
      title="#{txt['txt.branch.organizationName']}"
      maxlength="255" id="branchName" />
  </td></tr><tr><td>
   <util:inputText label="#{txt['branch.addressLine1']}"
      value="#{reg.user.branch.addressLine1}"
      edit="#{reg.edit}" size="30" maxlength="255"
      title="#{txt['txt.branch.addressLine1']}"
      id="addressLine1" />
  </td></tr><tr><td>
   <util:inputText label="#{txt['branch.addressLine2']}"
      value="#{reg.user.branch.addressLine2}"
      edit="#{reg.edit}" size="30" maxlength="255"
      title="#{txt['txt.branch.addressLine2']}"
      id="addressLine2" />
  </td></tr><tr><td>
   <util:inputText label="#{txt['branch.city']}"
      value="#{reg.user.branch.city}" edit="#{reg.edit}"
      title="#{txt['txt.branch.city']}" id="city"
      required="true" size="30" maxlength="255" />


  ...
```

Listing 1.2: Illustration of the JSF code behind the aimed approach from Fig. 1.2

```
<p:panel>
 <af:ui   edit="#{reg.edit}"
      instance="#{reg.user}" />
</p:panel>
<p:panel>
 <af:ui edit="#{reg.edit}"
  instance="#{reg.user.personInfo}"/>
</p:panel>
<p:panel>
 <af:ui edit="#{reg.edit}"
    instance="#{reg.user.org}"/>
</p:panel>
```

Listing 1.3: Illustration of a reusable field presentation template in the target language for the aimed approach from Fig. 1.2

```
 <util:inputText
     label="#{txt['$fieldPath']}"
     value="#{$value}"
      edit="#{reg.edit}" size="$size"
 required="$required"  email="$email"
 maxlength="$maxlength"
     title="#{txt['txt.$fieldPath']}"
        id="$field" />
```

Listing 1.4: Illustration of a reusable layout template in the aimed approach from Fig. 1.2

```
<table>
 <af:iteration-part maxOccurs="100">
  <tr>
   <td>$af:next$</td>
   <td>$af:next$</td>
  </tr>
 </af:iteration-part>
</table>
```

Listing 1.5: Illustration of generic field-to-template mapping rules in the aimed approach from Fig. 1.2

```
<mapping>
  <type>String</type>
  <default tag="inputTextTag.xhtml"
    maxlength="255" size="30"
    required="false" />
  <var name="username"
    tag="inputUsernameTag.xhtml" />
  <condition
    expression="${email == true}"
    tag="emailTag.xhtml" />
  <condition
    expression="${not empty maxlength
                  and maxlength > 255}"
    tag="inputTextAreaTag.xhtml" />
</mapping>
```

Listing 1.6: Sample Java Entity class specifying data for Listing 1.1 and Listing 1.2

```
public class User { ..

 @NotNull @Length(max=50)
 public String getUsername() { .. }

 @Column(nullable=false) @UiPassword
 public String getPassword() { .. }

 @UiRestrict("s:hasRole('ADMIN')")
 public boolean getConfirmed() { .. }
..
```

that tangles together concerns for a particular data presentation. To locate and isolate a particular concern in the code becomes hard due to tangling, but also a particular concern definition is distributed and restated across multiple locations because of its limited reuse. Next, the part of code belonging to a particular concern cannot be directly determined. Listing 1.1 presents sample source code for Fig. 1.1 in JSF technology with various concerns shown in different colors. The purpose is not to understand the content of the code, rather to illustrate the problem. To visually demonstrate the issue, consider the abstraction in Fig. 1.3. In order to capture multiple concerns at the same time for a particular UI page, conventional design cross-cuts and tangles concerns together, as well as a particular concern cannot be easily isolated from others. Isolation of a particular concern would improve the cohesion of the description; instead, the concern is distributed across other concerns. Consider what happens when the same system data needs to be represented in the UI using different widget types, layouts, input validations for given users, varying security rules, etc.? Most likely it would be necessary to copy the component code and design another component with slightly modified target concerns. The tangled concerns directly limit their reuse.

Next, consider the co-existing data definition in Listing 1.6. This data definition may determine how the UI description in Listing 1.1 should be structured, which components it should use, and what settings and values apply. In case when data definition changes, e.g., the *username* field *length* restriction extends to *255* characters and a text pattern enforcing an *email* value applies, the UI description in Listing 1.1 must conform to the changes. This directly involves a change of the *maxlength* value. The developer decides to replace the entire component for a given field based on the *email* restriction. When we consider that data definition changes, but the change propagation to the UI description delays or gets lost, weak type safety of the UI description does not indicate any error; consequently, the inconsistency may occur at runtime in production. Let us summarize the example highlighting individual problems using markers[3]. Listing 1.1 entangles multiple concerns together; this results (i1) with low cohesive[4] source code of the UI description. We see that the description in Listing 1.1 mixes all different kind of ideas together. The consequence is that finding the correct location/concern to apply a particular change requires high concentration, while being distracted by other concerns. Moreover, the description in Listing 1.1 is highly coupled[4] to the data definition at Listing 1.6. We can see that the type of the data field and its constraints

[3]Markers (i1), (i2),... are used for problem references in later text

[4]Cohesion & Coupling [6] are measures used to compare quality of source code. See Section 2.1.

influence the selection (i2) of a particular UI component that is, in fact, repeated over and over again by the developer for different fields. The coupling is also apparent through the information restatements (i3), such as the maximum length restriction of the field applies at both the data definitions and the UI description in Listing 1.1 and Listing 1.6. The way Listing 1.6 designs the data presentation is through entangled concerns (i4), which leads to repetition. For instance, consider the repeating text for inputText components, or the table row separators, etc. Even worse, when we aim to provide the same form in different layout or using different presentation components, we most likely end up with two very similar descriptions, entangling multiple same and unchanged concerns with only single concern changed. The result leaves us to maintain two highly similar descriptions that look like Listing 1.1. Any time the data definition changes, both descriptions must update. The inability to address varying concerns may lead, in the worse case, to exponential growth [7, A.8] of the UI descriptions. The number of combinations in the concern space influences the number of the descriptions; this would result in an overwhelming amount of code. Next, assume we aim to change a particular UI concern (i5) in the entire system. When you consider the outcome from the above text, the difficulty demonstrated in Fig. 1.3, or think of multiple situations described by components following the description in Listing 1.1, you notice that a particular concern may be captured multiple times, at many different physical locations, tangled with other concerns. Thus it may be replicated, and the change may result with a complex, long lasting update. When you add possible weak type safety, the process may become very error-prone. As an example, consider replacing all text components in the presentation with a new version from another library. The change may be as simple as the change of an attribute name. In order to execute the component change, we need to find all occurrences of the old component and modify the attribute name. This shows the concern distribution across the system.

A better arrangement of information would positively impact development, maintenance, reuse, comprehension, flexibility, variations, etc. For instance, compare Figs. 1.3 and 1.4. Fig. 1.3 shows tangled concerns that are distributed in a particular UI description; this may reflect the situation in Listing 1.1. Fig. 1.4 gives an abstraction representing an arrangement that separates individual information sets, concerns, and centralizes concern description. In order to provide a UI design that is in the abstraction similar to Fig. 1.4, it should avoid tangling concerns together. Naturally, multiple concerns play a role in the UI design, mostly when considering data presentations. The organization given in Fig. 1.4 brings better readability in the code, but does it support

> To develop and maintain it is the other side of the coin

Figure 1.3: Possible visualization of conventional UI design with cross-cutting concerns in source code

Figure 1.4: Possible visualization of improved UI design centralizing cross-cutting concerns in source code

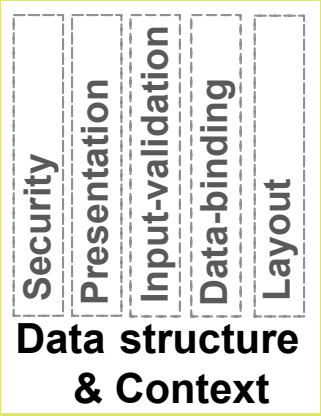

Figure 1.5: Possible visualization of aimed UI design addressing cross-cutting concern reuse in source code

concern reuse? If we leave all the concerns together in a particular UI description, it makes the description specialized to a given combination of concerns. Concerns are still captured repeatedly with each UI presentation description.

In order to support concern reuse, we should consider indirection [6]. Having a separate description for concerns, as suggests Fig. 1.5, provides the benefit of their reuse across different UI presentations. Although we feel that concern separation is a good step towards a design that supports better reuse, we need to raise the question about how to compose a UI presentation for particular data. With no doubt, we loose the single, self-descriptive UI description that connects all the pieces and decisions together, and we need to find another mechanism to connect them.

In the previous text, you may notice that the self-descriptive UI descriptions restate existing information. This information usually comes from application data and its structure, although in later text we will see that application context is similarly important. If we consider the data structure and context to be the conductor that directs the composition of UI concerns for its UI presentation, then we reduce restatements. Fig. 1.5 shows the situation described so far. Both data structure and context influence the selection of particular concerns that are used to derive the UI presentation. The consequence is that presentation of another application data type reuses the already defined concerns. This also means that growing the amount of application data does not directly impact the volume of UI description, which was the case with conventional approaches. Although we sense how the design is structured and the information is

captured, this thesis also describes the process that composes all these parts together to derive various UI presentations.

One possible result of such a concern separation approach is depicted in Listing 1.2. This example represents the form shown in Fig. 1.2. It shows three components that process data definition given by the *instance* attribute. The result is equivalent to the target form in Fig. 1.1 that in the conventional approach requires much more effort (Listing 1.1). Each component only binds to a particular data instance for which it produces the desired UI fragment (form, table, etc.). Internally it weaves together considered, separately defined concerns (Listings 1.3-1.5). Listing 1.3 shows a presentation template that represents a text field. Notice that there is no binding to particular data information; instead information is populated from the data instance passed to the component in Listing 1.2. Listing 1.4 represents a layout whose content is populated by the resolved field presentation from Listing 1.3. The resolution that determines a presentation template (Listing 1.3) to which binds a given data field (Listing 1.6) is achieved through generic field-to-template mapping rules in Listing 1.5. These rules reference and query field information to find the appropriate presentation template. There is no coupling to particular data field, which makes rules reusable across different data. All the mentioned concerns from Listings 1.3-1.5 are reusable throughout the application as well as they can be ported to different systems; thus they are not necessarily the subjects of development efforts.

To highlight a few of the advantages that are later explained in the thesis, consider points (i1) - (i5) from the text above. The previewed approach separates considered concerns to distinct locations. Their centralized definitions are easy to comprehend, have high cohesion (i1), which makes it easy to apply changes in a single location; furthermore one change (i5) can easily influence all its occurrences in UI presentations across different data and situations. The UI component selection (i2) is determined by generic field-to-template mapping rules; thus the developer does not need to repeatedly make the selection. Information restatement (i3) is reduced, since particular data instance can determine the data structure that is used, together with application context, to determine the UI presentation. The data structure information propagates to the presentation through an automated process. Listing 1.3 shows an example where is a particular template aware of information that may be captured together with a particular data field; the resolution process propagates the information from data structure given by data instance to the result following the template. The information in the

template relates to the data structure metamodel. Next, the previewed approach targets separation of concerns (i4). A novel concern can be integrated to any part in Listings 1.2-1.5. A situation when presentation template changes based on context can be handled by extended field-to-template mapping rules resolved at runtime. A wide layout can be selected through a novel layout definition (e.g., specified as attribute of the component in Listing 1.2). A conditional rendering of a particular field based on security rules and context can be pushed down to the data structure as its extension information or through field-to-template mapping rules, or even through the presentation or layout templates. The code presented in Listings 1.2-1.5 is actual code that applies a library integrating the design approach brought by this thesis.

1.1 Problems Specification

The previous section provides the sense of the problem with which this thesis aims to deal. At the same time, there are multiple other issues to address as well, and this section provides their brief specification. The detailed explanation brings the consequent chapters. For the broader understanding of the context and notions, refer to Chapter 2.

Problem 1. Information restatement. Enterprise Applications (EAs) and information systems provide mechanisms to manage, store, and retrieve data typically stored in a relational database [6, 8]. Application data definitions are used in object-relational mapping, and the definition usually extends with additional information that conducts the mapping [9]. It is important that data being stored to the system follow integrity constraints and are valid [10] towards their definition specification. The UI part of such systems reflects data specification, constraints, security, and other concerns.

In the conventional UI design, the presentation components provide mechanisms to specify input validation and constraints, which restrict user input. From the UI perspective, there does not seem to be anything wrong with that, but from the system global perspective, the constraints are specified and captured twice: once when specifying data definitions for the storage and retrieval and again in the UI when specifying the presentation components. Such duplicated specification must correlate to avoid inconsistency errors, although this extends the development and maintenance efforts.

Problem 2. Repeated decisions. A similar issue occurs when binding system data to particular components (widgets) of the UI. Usually, each data field must explicitly bind to a particular component. The UI description language associates particular fields with particular components; however the field and its constraints determine the selec-

tion of which component to use and its validation. For example, a person's *username* is a *text* field, and the UI description specifies a component that considers text input for the field; obviously there is restatement in the UI description, since it explicitly specifies a particular component based on information from the data definition[5] . The developer should not be responsible for picking the component for a particular data field because it must be maintained. Since the UI description usually uses language with weak type safety, it is tedious, error-prone, and presents a challenge to avoid inconsistencies [A.2]. Once the data restriction changes for the *username* to become an *email*, different UI component might be required, although the UI change requires manual modification of the UI description.

Problem 3. Issues with tangled code. Next, we want to describe a UI data presentation, but in order to do that in the conventional approach, we must tangle various concerns together. It should be possible to effectively decompose cross-cutting concerns that appear in the UI. The issue is demonstrated in the motivation example, for instance in Figs. 1.3-1.5. Conventional design does not posses a mechanism that would effectively deal with cross-cutting concerns [4] and their separation. The inefficiency is apparent in both GPLs and DSLs. As described in [4] and [12], even the object-oriented mechanisms and techniques posses such ability. The result of the inefficient separation of concerns is the tangled-code (concerns), low reuse, low cohesion, complex readability, increased volume of source code, increased development and maintenance efforts, etc. In the worse case, the amount of duplication grows exponentially with the considered concern space and individual concern variations.

Problem 4. Centralization of information. Another problem related to the separation of concerns is to find an appropriate location in the system to centralize information. There exists a large community behind the MDD [A.2], which suggests designing a model that is the source of information. Such a model is then transformed to code that defines various components that consider the information. This way the model can be transformed into the component for information storage and retrieval, as well as to the UI. At the same time, there exists a large community and industrial standards that suggests capturing information in the source code. To avoid reinventing the wheel [3], the best practice should be considered irrespective of the particular approach.

[5]This can be seen similar to functional dependency in normalization of database management systems schemas [11]

The MDD benefits should be considered from the perspective of the UI design. The nice benefit brought by MDD is the well-defined location for information and the direction towards platform-independent description of the UI. On the other hand, the model must be extended to consider industry standards [A.2]. MDD is capable to address cross-cutting concerns, although there is no generic and broadly accepted mechanism for this [A.4]. At the same time, system evolution management is impractical [A.3], and when considering context-sensitive UIs, the model-based transformation must be aware of the runtime-context, and thus either operate at runtime or prepare all possible states at the compile time, which could occupy disk space for hypothetical and never reached UI states [A.4]. As stated in Problem 3, there might exist exponential growth of the states, related to the combinations in the concern space, which only extends the difficulty in generating all possible states at compile time.

Problem 5. Benefits of MDD in code-based approaches. MDD brings significant advantages regarding centralization of information, reduction of restated information, etc. Although MDD approaches are very common for scientific research [A.2, A.3], the industrial community usually uses code-based application development [2, 6, A.4]. The difficulty with MDD is that developer usually uses it to build application skeletons, but later modification are applied to code instead of the model [A.2], which consequently disables the later MDD use, since all code-level changes would be erased. The idea of partial MDD use, for instance for UI, is common in the Human-Computer Interaction (HCI) discipline [A.3], although when the UI presents data that come from a code-based subsystem, the model suffers from Problem 1. In this work, we aim to still use the advantages of MDD for code-based approaches. The information that is in the MDD captured by model could be represented by code. The model can be derived from code, which avoids the Problem 1.

Problem 6. Deriving UI with process supporting the separation of concerns. As emphasized earlier, neither the GPL nor DSL solutions are sufficient since they do not effectively address cross-cutting concerns. Thus concern-separating techniques, such as Aspect-Oriented Programming (AOP) [4] and Generative Programming (GP) [12] can be considered. Both are examined in more detail in Chapter 2. For instance, AOP suggests capturing different concerns separately, which increases the information cohesion and reuse. Although the contemporary use of AOP [13] targets method invocation, the initial published work [4] provides a wider perspective. At the same time, the terminology and mechanisms to address cross-cutting concerns are well

described and accepted. It is possible to base an approach on AOP-mechanisms, even for the UI design.

This work explores the benefits of applying AOP in the area of UI design. AOP suggests that there exist components that are being extended with the separately defined concerns and that such components should indicate the location of the join point between the component and extra concern. When considering the AOP-mechanisms in the area of UI design, concern integration is not considered upon method invocation, but as an extension to the transformation process. Perhaps the general idea of MDD and AOP can coexist when finding the process to present data information in the UI. Thus the model, no matter whether designed through a dedicated graphical model, DSL or GPL, can be the subject of an information inspection that can be based on well-defined join points. Such inspection derives the structure of the UI presentation. Once we resolve the data and information need for the UI derivation, then it is up to the transformation process to reuse the inspected information, weave together UI concerns, and derive the UI presentation for given data. Presuming this is possible, then the questions become 1) what is the impact on development and maintenance efforts, 2) what are the design limits and benefits, and 3) does it help with performance, UI variability, etc.

Problem 7. High costs and efforts related to design of Context-aware User Interface (CaUI). What do we do when the UI needs to change with the context, or when it is sensitive to particular user, location, time, etc.? In the motivation section, we show that concern tangling in conventional approaches is the main inhibitor of concern reuse, and thus varying concerns may result with multiple copies of UI descriptions for particular data. With context-awareness the concern space grows, and thus the development and maintenance efforts, and consequently the costs, grow as well. Although there exists a large volume of research in this area, the software engineering perspective is rather limited [A.3]. For instance, existing research mostly focuses on usability, efficient features, distribution of the UI to multiple devices, but the development or maintenance efforts are left behind, as is the UI performance and responsiveness. There are user studies regarding usability and user satisfaction, but the code volume comparisons with existing conventional approaches are not addressed. Although some results of graphical design features might be stunning, a production deployment might be impractical due to increased costs related to the development or maintenance efforts [A.4].

In order to apply context-aware UIs in practice, the costs should not increase significantly. If we solve Problems 4 and 6, the concern definition becomes defined once and easily reused. Thus the growth in concern space does not impact the development and maintenance efforts as critically as with conventional approach. To prevent the tight dependency between data definitions and UI description, the Problem 6 description suggests deriving information from existing code, which is the code defining application data. If it is possible to reuse existing information captured in code for data definitions for the UI derivation and the transformation process can easily integrate new concerns, then the growth in the amount of application data or UI concerns does not directly impact the volume of UI description for data presentations.

Context-aware UIs for future systems [A.3] must consider runtime application context, and in conjunction with Problems 3 and 6, the compile time derivation seems impractical, thus the derivation process must take place at runtime, although for practical use, it cannot impact UI rendering performance and responsiveness.

Problem 8. UI delivery to remote clients, performance and responsiveness. When we look at existing solutions for UI remote client delivery, specifically to web-based applications, remote clients request the UIs from a distant server. Web-based servers provide clients Hypertext Markup Language (HTML) content together with other, usually static, resources. When we look to the HTML content, it reveals that UI description is tangled together. Information tangling, as mentioned earlier, disables reuse of individual information concerns. If the server sends to the client the form at Fig. 1.1, and then aims to change the layout since user re-sized web-browser window, the entire tangled block of information must be retrieved again from the server, even though only a single concern changed. Even that many concerns are unchanged, they cannot be cached at the client-side because the tangling disables it.

Even though solution to Problems 6 and 7 would separate concerns, the separation is not maintained for the UI delivery, because the integration process takes place at the server-side, and thus the output is what the server sends to the clients, a tangled UI description in the HTML format.

If we separate various concerns and maintain the separation for the server-client delivery, the client can become responsible for the concern weaving, having full control over what has changed and what needs to be requested from the server. This pushes the UI rendering responsibility to the client, but consequently it supports extended caching options with concern reuse. The separated concern delivery reduces the vol-

ume of transmitted information, and furthermore, it reduces server involvement in the UI presentation and thus reduces required server-side resources to process a particular client.

Problem 9. Platform-specific UIs. Even though web-based applications are very common, the user experience is usually better using a native application [14]. Consider for instance a mobile application and web application that both present a selection component. The native component extends to the entire display, which makes it easy for the user to tap on the screen. Although the benefits are undeniable, the efforts and costs related to application support for various platforms are unbearable. When we put next to each other a data presentation rendered on different platforms, we must note that the structure, constraints, input validation, security, data values, etc. are the same, only the presentation and layout changes. Thus using the concern-separating UI delivery solution to Problem 8, while separating out the platform-independent concerns, opens the server-side application for reuse across clients that base on various platforms, while reducing both the development and maintenance efforts as well as involved costs.

1.2 Contributions of the Thesis

This thesis brings contribution to multiple areas of research as well as shows the practical use in production-level environment that allows for easy transition to the industry [15]. It addresses problems specified in Section 1.1. In the following text is given the summary of the thesis contribution, although details are given in Chapters 4 through 9. All the presented content was published in various international conferences as research papers, SCI-E journals or as extended versions of conference research papers. The content of Chapters 5-8 correspond to the content of published journal papers [A.1, A.2, A.3, A.4]. The refereed publications are listed in Bibliography B.

In order to reduce data information restatement (Problem 1) and restated decisions (Problem 2), this work considers two possibilities. First, the MDD approach is applied using the Unified Modeling Language (UML) Class model [A.2] transformation. Although the UML Class model does not provide all the information needed to derive data UI presentation, there exists an extension mechanism through UML profiles. Existing code-based industrial standards are considered and provided through UML profiles with stereotypes for presentation, user input validation, and security. This extension allows deriving the UI from data elements described through UML Class models. The model becomes the source of information (Problem 4). The information

advances to the UI as well as to the data storage components through MDD transformations. The benefit brought by MDD is the abstraction, formalization, and easy transition among various platforms (partial solution to Problem 9).

Next, the code-based perspective is evaluated [A.15]. It is possible to capture UI-specific information together with the data definition at the code level. In fact multiple industrial standards [9, 10, 16] provide similar information extensions to the domain model [A.2]. Having information captured at the code level allows us to derive data-based UI components directly from the existing code (Problem 5). The derived UIs components are equivalent to these that can be received from the UML Class model through MDD. The advantage of the code-based approach is that there is no need for any external model. Such an external model used solely for UI derivation greatly increases development and maintenance efforts in case the rest of the system uses mainstream EA development [3, 6, 8, 16]. The code-based approach is more familiar to developers aware of conventional development standards used for EA development [A.4], thus there is no significant push to learn a novel approach. It is fairly easy to extend existing frameworks or even reuse already existing information that already applies to other system concerns. Since developers tend to apply changes to code [A.2], the above MDD approach would have to apply reverse engineering [A.13] in order to apply such changes to the model and preserve them upon the next model-to-code transformation.

As briefly mentioned in Section 1.1, neither GPL nor DSL effectively address cross-cutting concerns, which leads to code tangling. MDD can consider multiple models [A.17], although a general integration mechanism is missing [17]. Thus AOP concepts are considered since they effectively address the separation of concerns. UI data presentations consist of multiple cross-cutting concerns that are usually tangled together [A.17]; this is apparent in their source code. To address this, an AOP-based, UI design approach is proposed [A.4] (Problem 3). It considers the domain model to be the subject of an inspection and an AOP-based transformation that processes the result of the inspection and determines a particular UI presentation, while integrating UI concerns (Problem 6). Since it operates at runtime, it considers runtime context and adapts to context-aware UIs (Problem 7). Practical benefits of the approach are provided based on an empirical study conducted [A.4] at a production-level application. Among the advantages are the separation of UI concerns, reduction of restated information, high concern cohesion, operability at runtime, context-awareness, reduction of development and maintenance efforts, reduced code volume, single focal point of information, full

control over the resulting presentation and the target UI language description, etc. At the same time, the approach is limited to data presentations, thus it is expected as an extension to third party UI frameworks that provide the target UI language, components, navigation, event handling, etc. The approach computational performance is evaluated [A.4], and shows similar results when compared with JSF (Problem 7). The approach is deployed in the production-level application for the ACM-ICPC[6] contest management.

The AOP-based, UI design fits well to the context-sensitive environment. Not only can it handle application runtime context, it can integrate third-party, context-aware UI approaches [A.3, A.21]. Since the approach applies inspection to application domain models, considers their context, and involves the AOP-based transformation, it can produce input to third-party approaches. This brings a synergy and simplification. For instance, a third-party, context-aware UI approach normally must restate information from the domain model that is designed in a mainstream framework. Restatement brings the disadvantage of tedious and error-prone work [A.2]. Instead, input to the third-party approach is derived through the AOP-based UI approach (Problems 1 and 2). From the other perspective, this extends the capabilities on both sides. For instance in [A.3] such synergy brings the capability to stream the UI in a platform-independent format (Problem 9), which allows composition of the UI on various clients using native platform components to improve usability. Additionally, advanced features are provided, such as automated layout allocation of fields, based on usability metrics. Future changes to the domain model (in the mainstream design) are immediately reflected in the third-party approach, which avoids inconsistencies and errors and reduces manual efforts.

Although the above text mentions performance of the AOP-based UI design, it can be considered in much deeper detail. In web-based systems, the UI is delivered from a server to remote clients. The UI page load time is influenced by network conditions, although the load time can be further influenced by other factors, such as the transmitted content size and client-side caching. Conventional UI designs deliver clients the UI description in a tangled format. Thus field presentations, layout, data presenting component structure, presented data values, etc. tangle together in the HTML. Although the AOP-based UI design separates concerns at the server-side, it weaves them together before they leave the server, thus there is no difference in the transmission

[6]The ACM International Collegiate Programming Contest (ICPC) is a multitier, team-based, programming competition. http://icpc.baylor.edu; 2015

when compared to conventional approaches. Since these concerns tangle together at the client, the concern reuse as well as the client-side concern caching is limited. This leads to situations that are quite inefficient with respect to transmission and thus UI responsiveness. For instance, when a single concern changes in the UI, the entire UI must reload from the server-side. Since the AOP-based UI design already separates concerns at the server-side, it can be extended [A.6, A.1, A.12] in a way that it provides UI concerns separately to a particular client, and the concern weaving becomes distributed between the client and server-side (Problem 8). Consequently, clients can reuse particular concerns and request only these concerns that have changed. In addition, the client can request various concerns concurrently in parallel. We present a case study that applies the distributed, AOP-based UI design and compares it with the conventional approach regarding the page load time. The study demonstrates reduced load times for the presented approach. At the same time, the information volume processed by the server-side reduces for each request, which reduces required resources.

Dividing the distributed UI concern delivery particles to platform-independent and platform-specific allows us to reuse the platform-independent information about UI across different client applications that base on different platforms [A.7]. This brings the advantage of reduced efforts related to maintenance and development of such applications. Furthermore, the application becomes flexible to server-side changes to data definitions that are immediately reflected by all clients (Problem 9). The limitation is that the approach deals with data presentations, thus controller, navigation and page flow are left for conventional development approach.

1.3 Organization of the Thesis

This thesis is organized as follows. Chapter 2 provides basic notions and background materials utilized in this work. Chapter 3 introduces related research and provides literature overview. The summary or related work and the roadmap to the research in the subsequent chapters are given in Chapter 4. A model-driven approach is presented in Chapter 5, together with definition of UML profiles that extend the UML Class diagram to capture information necessary for diagram transformations to the UI. Chapter 6 presents an alternative approach based on code-inspection. It is discussed and presented together with AOP-based transformation process. In this chapter, an evaluation is provided; it shows a proof-of-concept, production experience as well as samples of integration with other projects. In Chapter 7, the integration with a third-

party framework is discussed in the area of context-aware UIs. The synergy provides advantages brought by both approaches. Chapter 8 discusses UI transmission to remote clients as well as caching. Future works are presented in Chapter 9 before the conclusion of the thesis in Chapter 10.

2

Basic Notions

Most people make the mistake of thinking design is what it looks like. People think it's this veneer - that the designers are handed this box and told, 'Make it look good!' That's not what we think design is. It's not just what it looks like and feels like. Design is how it works.

-Steve Jobs

Co-founder, Apple Inc.

UIs play an important role in interaction between software systems and users. Users often judge the quality of the entire system based on the quality of the UI, and thus it may significantly impact market success. For this reason, there is a significant aim to provide sophisticated UIs to satisfy high user expectations; however, UI design typically involves significant complexity for development and maintenance. The complexity has many sources, such as large volume of UI source code, but also the logical location of the UI in the system. UI is built on the top of other software components, which makes its code vulnerable to changes in the underlying code blocks and components.

As an example, consider a data UI presentation, i.e., a data form through which users submit data to the system. Such a form must reflect underlying system data and their constraints. This is, although, only "the tip of the iceberg" of concerns that should be considered. Proper presentation widgets must be selected as well as an appropriate layout, and it must consider input validation, business rules, systems access rights, etc. Often developers design such forms in a manner that mixes and tangles various concerns together. The motivating example in Chapter 1 shows a problem commonly known as a *cross-cutting concern* [4, 12]. This term [18] describes parts of a software system that logically belong to one single module but cannot be modularized due to limited abstractions of the underlying programming language. Cross-cutting concerns are responsible for code tangling, fragment complexity (readability), and decreased cohesion [6]. Often, such code fragments exhibit high coupling [6] to considered system aspects; thus at anytime a system aspect changes, it possibly impacts the fragment and forces its changes. The situation becomes worse when we build a software module (in our case UI) in a framework that uses a DSL with limited type safety. This means that when a single system aspect changes in the underlying code, there are limited mechanisms that would indicate the inconsistency with its references in the DSL, unless we deploy the system and test it.

From another perspective, UI code is hard to maintain because it exhibits information restated from other system components/modules [19]. For a good example, consider the Java EE architecture [20] that suggests capturing persistent data with Java objects, called *entities*, and describing UIs through component-based JSF technology [2]. JSF uses Extensible Markup Language (XML) UI descriptions and provides a templating framework. Naturally, when we deal with two different languages, such as Object-Oriented (OO) Java and XML UI descriptions (JSF), we face multiple complications. For instance, JSF code fragments have to restate information already defined by entities (data properties, constraints, validation) [A.15]. Next, XML has weak type safety; thus a typographical mistake can cause inconsistency, which might be discovered too late after system deployment and result in system error. The impacts of information restatement are increased development and maintenance efforts [A.4]. [A.11] suggests that such approach makes the development unnecessary tedious and error-prone.

From both above examples, we see that conventional UI design exhibits multiple development and maintenance deficiencies, which may become a burden for developers, especially when we consider that statistically [21] on average 48% of the application

code is devoted to the UI. The average time spent on UI development is 45% during the design phase, 50% during the implementation phase, and 37% during the maintenance phase [21]. Similar numbers are observed in a study made with the ACM-ICPC registration system [A.4] where 44% of the code is devoted to the UI-part; furthermore 15% of the code is devoted to forms. Such statistical information provides us with insight on how significant the UI is from the development and maintenance perspective. Improvements and simplification to the UI may have significant impact to the entire project; thus exploring a novel approach to reduce development and maintenance efforts is well justified. At the same time, it is important to consider contemporary industrial standards for enterprise system development [22] and to provide easy transition into existing systems and processes, while avoiding reinvention of the wheel.

2.1 Basic Notions

This section describes basic notions used throughout the thesis.

Notion 1. Coupling: Coupling [6] is a measure of how strongly one element connects to, has knowledge of, or relies on other elements. An element with low (or weak) coupling is not dependent on too many other elements; "too many" is context-dependent. These elements include classes, subsystems, systems, and so on.

Notion 2. Cohesion: Cohesion [6] (functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are. An element with highly-related responsibilities, and which does not do a tremendous amount of work, has high cohesion. These elements include classes, subsystems, and so on.

Notion 3. Maintainability: Maintainability [6, 23] is a measure of how easy it is to maintain a software system. Preserving low coupling positively impacts maintenance. It reduces the probability that a maintenance change in a single component needs to be propagated to other components. Preserving high cohesion, gives a higher probability that a maintenance change in the system requirements only affects a small number of components.

Notion 4. Development efforts: Fowler et al. [8] suggests that lots of people write computer software, and we call all of it software development. Effort is understood as the amount of exertion expended for a specified purpose, in this case software development. Development efforts usually grow with the complexity of considered tasks, situations, requirements, etc. but are also influenced by the software architecture [23].

On the other hand, reduction is brought by the use of design patterns [3, 8]. Bass et al. [23] also suggest its close relation business quality called *cost and benefit*, when stating that "The development effort will naturally have a budget that must not be exceeded."

Notion 5. Maintenance efforts: Maintenance is an aspect of maintainability [23]. It can be understood as the phase of a software development process [6, 23], when the system is extended with error-fixes, performance improvements, functionality, changes in requirements, etc. Effort is understood as the amount of exertion expended for software maintenance. Note that even authors such as Dijkstra were concerned with maintenance. For instance [24] suggests partitioning and structuring software to as oppose to producing correct results; consequently [23] suggests that an elegant software design increases the ease of development and maintenance. The creation and maintenance of large systems presents a substantial software development challenge in designing for the performance, modifiability (accommodation of changes in requirements), scalability of functions needed to extend these systems to provide more functionality, fidelity, automation, etc. The aim of OO or AOP, introduced later in this chapter, is to simplify software development and maintenance [13].

Notion 6. Responsiveness of UI applications: Responsiveness is related to system performance; it is the system ability to complete a given task within a particular time. Responsiveness of UI applications [13] can be understood as the time needed to accomplish a particular operation provided by the UI. For web-based systems, many factors influence responsiveness, including system design, amount of transmitted information, network conditions (latency, bandwidth, etc.), current system load, hardware performance, etc.

Notion 7. Readability: Readability [8] is a measure of how easily [13] a human reader can understand the purpose, control or data flow, and operation of source code. Low coupling simplifies readability, since comprehension of the interaction among components requires studying fewer components. High cohesion improves readability since all related information is aggregated at the same location. Readability is important since developers spend most of the time reading and comprehending the code. Unreadable code is hard to maintain, leads to errors, poor reuse, and often leads to duplication.

Notion 8. Reusability: Reusability [3] is a measure of how easy it is to reuse a component in a different system or for a requirement variation. Preserving low coupling positively impacts reuse. Generic components tend to be reusable since they are less dependent on other system components, on the contrary specific components tend to

be hard to reuse. Reusability improves readability. Preserving high cohesion of a component gives complete and well-defined functionality with easy reuse.

Notion 9. Extensibility: Extensibility [3] is a measure of how easy is to extend system with new functionality, or to allow incorporation of relevant, new technologies as they are developed [23]. Preserving low coupling positively impacts introduction and integration of a new component. Preserving high cohesion simplifies development and design of new components, since we do not need to be concerned with aspects not related to a given functionality.

Notion 10. Usability: Usability [23] considers how easy it is for a human user to interact with software system. It studies user satisfaction, experience, usefulness, clarity and the ease to achieve a given task. Usability is often associated with UIs where it measures how effective it is in a given domain to prevent errors and guide user to achieve a particular goal, etc. In the context of UI, context-awareness and good responsiveness improves usability [23].

Notion 11. Concern: A concern [4] is a set of information that influences the source code of a particular program or a component. A concern could be a performance perspective of a program, its ability to present data, to secure them, to verify consistency, and so on.

Notion 12. Separation of concerns: Separation of concerns [4, 5, 13] is an approach for modularization that brings a reduction of design complexity. When we have multiple concerns that tangle in the same section of source code, we call it highly-coupled or tangled. We usually aim to separate concerns to support readability and maintenance, but sometimes such separation is not possible in conventional programming languages (see cross-cutting concerns).

Notion 13. Cross-cutting concerns: Concerns that cannot be easily separated and tangle together in a given source code section are called cross-cutting concerns [4, 13]. These concerns cannot be cleanly separated from each other, which causes code duplication, hard reuse and has the effect of a "spaghetti code" with difficult maintenance. Existing programming paradigms that address cross-cutting concerns include AOP [4] and GP [12]. Object-Oriented Design (OOD) has limited constructs to effectively address cross-cutting concerns [4]. Similarly MDD can address cross-cutting concerns, but a generic integration mechanism is missing [A.3, A.4].

Notion 14. Metamodel: A metamodel [6] is a model that is used to specify and define other models. It usually defines elements from which to form a model and their constraints. It may define a language for expressing a model. It gives an explanation and definition of relationships among various components of the applied model.

Notion 15. Aspect: AOP considers units such as GPL components and aspects [4, 13]. An aspect is a feature of a program that can be linked to the other parts of the program, although it does not need to be the program's primary functionality. An aspect can be multidimensional and can bring both functional and non-functional properties to other concerns in the program. In AOP an aspect consists of a pointcut (see join point) and advice that defines the extension to the core program. As suggested by [13], aspect is a new unit of modularization in the AOP methodology that provides separation of cross-cutting concerns.

Notion 16. Join point: A join point [4, 13] is a point in the control flow of a program. It specifies a location or a point of execution in a program where an aspect (its advice) can be executed. A set of join points is called a pointcut. When a program execution reaches a join point defined in a pointcut, advice can be executed.

Notion 17. Object: The Object-Oriented Programming (OOP) introduces the concept of an object [23] that consists of fields and methods. An object has properties defined through fields and also defines behavior through methods and interaction with other objects. Objects usually hide data designed through its fields and other objects that want to access the data must interact with given objects through its methods. An object is similar to an Abstract Data Type (ADT) [25] with the addition of inheritance and polymorphism.

2.2 Enterprise Application Design

The term EA is often mentioned in literature [2, 6, 8], although consider that various authors may mean different things by this term. This work uses the meaning suggested by Martin Fowler [8]. Fowler provides a detailed description of EA. In his book [8], he suggests starting with examples, such as systems dealing with payroll, patient records, shipping tracking, cost analysis, credit scoring, insurance, supply chain, accounting, customer service, and foreign exchange trading.

An EA usually involves persistent data, which needs to be persisted over multiple runs of the program or even forever. Since there is usually a lot of data, they need to be

organized, indexed and easily accessible. This involves persistent storage, databases, or more abstractly a repository [25]. EAs provide access to data concurrently to many users and must handle transactions and data integrity. Since the '90s these systems tend to be web-based and work over Internet, utilizing client-server architecture [25].

Fowler also mentions that since there is so much data, many UI screens exist to handle it. At the same time, users come with different levels of technical expertise and, for some users, low technical expertise. This means that the data has to be presented in lots of different ways for different purposes. EA usually supports batch processing over the data and focuses on interaction with users.

EAs are rarely isolated and usually integrate other EAs, which requires them to be based on common communication technology. EAs usually separate component responsibilities into distinct layers [6, 25]. So far we mentioned the presentation on the top and persistence at the bottom. In the middle layer, EA handles business processes over the data. Although we often read in literature the term business logic, Fowler considers this to be often "illogic". This layer is very individual for each organization. It is very common that many strange conditions interact with each other, usually to handle special cases. Management of such thousands of conditions is often a burden.

The Enterprise Application Architecture (EAA) can be summarized as follows: It uses various architectural styles [25] such as client-server for interaction with users, repository for persistence, and three-layered architecture style [25] to divide responsibilities of the EA. The client-server interaction involves network communication that may be based on Transmission-control protocol (TCP)/Internet Protocol (IP), but more likely on the Hypertext Transfer Protocol (HTTP) involving HTML. The repository involves databases or persistent storages such as relational databases, message queues, distributed databases, etc. EAA has the high-level components divided into persistence, business, and presentation layers. Practical examples of EAA are applications built with Java EE or .NET frameworks.

In relation to EAA, it is important to mention that for the last two decades the most dominant programming languages used to design software systems are OO languages, [3, 6]. OO languages fit into the category of GPLs [4, 12]. OOD applies to all three-layers of EAA, although it is very common for the presentation layer to involve Component-Based Design (CBD) [2] and DSLs [2, 26] to describe the UI. In case of web systems, these components or DSL descriptions often translate to the HTML, which is the result provided to clients.

The basic rational for components in CBD is generalization, easy extension and simplification of the client code. The concept is introduced by [3] through the composite design pattern. DSLs, as oppose to GPL, focus on a specific domain, and thus may capture the domain concerns more effectively [27] than GPLs. On the other hand, we often need to integrate information already described in GPL into DSL. Thus, we cannot decouple the integration of DSL and GPL structures. One issue presented by [28, A.4] suggests that DSL descriptions often need to restate information from GPL components, in order to extend them. This introduces extra development and maintenance efforts and may become a source for errors.

2.2.1 Object-Oriented Programming/Design (OOP/D)

Historically an object is an evolutionary step from an ADT [25]. In the OO style, we define the concept of a class and its instances called objects. A class defines a set of properties or fields and a behavior in the form of its methods. An object usually encapsulates and hides its properties from other objects and lets its methods provide the aimed behavior on the properties. Objects interact with each other over method calls.

When designing class models, for instance to capture the application domain, it is useful to divide properties into attributes and associations [29]. An attribute has a common, basic data type; an association is a property, whose data type is defined by the class model in the considered domain. For instance, consider a class that has a String property; this property is an attribute. If a property has the data type Person or Set of Persons, where Person is defined by the class model, then the property is an association. Object interaction then often uses associations to reach other objects or an indirection [6]. An object has knowledge about other domain objects, introduced through method parameters, associations, variables and so on.

Any kind of object knowledge about other objects can be understood as *coupling* among the objects. This is evident for example when an object changes its name; all coupled objects must reflect the change (apparent in their source code). In OOD we aim to keep the coupling low [6].

Next, the OOD brings the concept of inheritance and polymorphism among classes. This means that a class can extend another class, shadow its methods and extend the original behavior. In the other direction, a class can be generalized with a parent

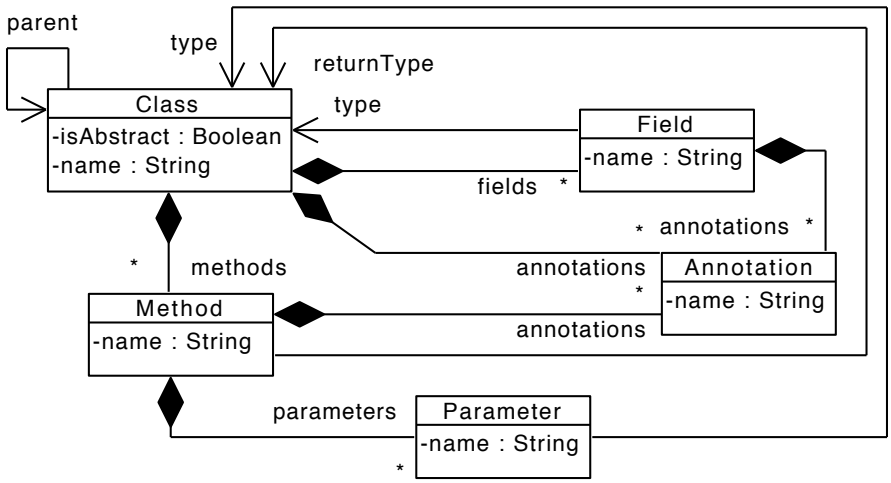

Figure 2.1: Example object-oriented design metamodel

class exposing the generalized idea/purpose, while child classes capture variations and custom details.

When we deal with a programming task/problem, there are two basic approaches to decompose it: through a class composition that uses associations or through the inheritance [3, 8]. The composition and inheritance are orthogonal approaches, and they can be combined together as well. As can be seen from the above or from OOD literature [3, 6], it provides multiple mechanisms to deal with software problem decomposition. Since no one wants to reinvent the wheel, OOD best practice solutions exist, known as design patterns [3, 8].

Class constructs, semantics and constraints are defined by a metamodel. For instance Fig. 2.1 shows an example class metamodel deduced from the Java programming language. It shows the concept of classes with relation to fields, methods, method parameters, and annotations.

Each class, field, method or its parameter can be extended with a set of annotations. An annotation extends the information captured relative to its location. In the past, such an extension would be captured through XML descriptors, but later such approaches become known as XML-hell [2] because of difficult maintenance. Example use of annotations are class extensions for input validation [10], persistence [9], security [30], dependency injection [8, 30], etc.; broad examples are provided later in this work.

Before we move to the description of various layers, we look at a few basic parameters that impact code maintenance and readability. We have already introduced coupling [3, 6, 31]. Elements that introduce high coupling in their design are hard to read and maintain. Thus in order to keep the design well-readable, reusable and maintainable, the coupling among classes (or components) should remain loose.

Another important parameter the keeps classes well readable is *cohesion* [6]. *Cohesion* from its original meaning predicts that a class (or component) should direct its functionality to a single goal rather than to many different goals. A class (or component) with low cohesion has too many responsibilities, and tries to solve many unrelated things. Such a class (or component) is hard to maintain, read/reuse, and comprehend by developers/designers.

Classes that together design a cohesive and coupled solution to a sub-problem can be referred as a component or subsystem. The overall system is often composed out of multiple components that may introduce various levels of coupling. Often we hide the component under a Facade design pattern [3] that only introduces to the outside world the component's public functionality and, at the same time, hides its internal complexity.

In order to define a high-level system abstraction, the concept of layers was introduced. A component, whether large or small, gives us a sense of a structure. In case we need to logically separate/aggregate classes with different/similar aims, we can place them into a package, which is a hierarchical logical structure that may contain other packages, classes or artifacts.

2.2.2 Three-layered Architecture

Contemporary development frameworks usually apply a three-layered architecture [6, 8, A.4]. This section introduces the responsibilities of the layers. As a reference, we use the Java EE [16] architecture.

The lowest layer of the architecture is responsible for persistence and typically uses OOP [3, 6]. This layer must deal with mapping of objects to the relational database or other storage. In the case of relational databases, the mapping is called Object-Relational Mapping (ORM). In Java EE exists a standard for the ORM called Java Persistence API (JPA) [9]. The JPA specification states its objective "to provide developers with an object/relational mapping facility using a Java domain model to manage a relational database." This bottom, persistence layer consists of an object-oriented domain model representation that reflects the entity-relationship database model[1,2]. A class of the domain model usually consists of fields (attributes and associations) and getter/setter

[1] Alternative terms are stated in [29] where they refer to an object model representation that reflects the relational data model representation.

[2] In the following text, I refer to the OO domain model representation as the domain model [8]. Some researchers also refer it as a data model.

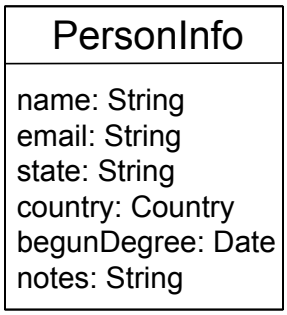

Figure 2.2: Example PersonInfo Entity

Listing 2.1: Example Java code fragment for Fig. 2.2 with JPA and Bean Validation annotations

```
@Entity
@Table(name = "PersonInfo")
public class PersonInfo {
 private String name;
 private String email;
 private String state;
 private Country country;
 private Date begunDegree;
 private String notes;
   ..
 @Column(name="name",nullable=false,length=100)
 @NotEmpty @Length(max = 100)
 public String getName() { return this.name; }
 public void setName(String n) { this.name = n; }

 @Column(name = "begunDegree", nullable = false)
 @Temporal(TemporalType.DATE)
 @NotEmpty @Past
 public Date getBegunDegree() {
    return this.begunDegree;
 }
 ..  /* other getters/setters */
```

pairs to access fields. It can be seen that this layer captures the static structure of a system, mostly in the form of the data that the system manages. In JPA terminology, each class that is persistent is called an *entity*. A class becomes an entity when it is annotated with an entity annotation [9]. Each entity field is persisted to a database, unless explicitly signified that it should not be. An example is shown in Fig. 2.2 where the example class, PersonInfo, is captured in the UML class diagram. This class can be transformed to an entity in Java code as shown in Listing 2.1. Notice that this Java class uses multiple annotations, these are explained in Chapter 5.

Each entity class may capture further constraints, such as the set of unique fields; an individual field can be constrained as well. The JPA standard defines annotations to conform data integrity (nullable, unique, read-only, etc.) as well as to apply persistence. Thus in order to define persistent data, we define an entity class, and its fields represent data properties or table columns. This approach becomes the best practice for many other frameworks such as .NET, PHP, etc.

Furthermore, over time it was found useful in design to use field annotations for entities beyond persistence. For instance, a Java standard called Beans Validation [10, 32] suggests that a set of validators applies to entity class fields and enforces validity of data properties received from upper layers of the system. This approach allows designers to

non-forcibly extend field meaning and seamlessly integrate validation. Such addition of validation is easy to understand, maintain, and locate. Listing 2.1 shows annotations for both JPA and Bean Validation. Clearly, the persistence layer is pretty straightforward with respect to development practices, when considering the above standards.

The business layer is, in comparison to the persistence layer, the least standardized among different frameworks. The intent of this layer is to capture application business rules and elaborate interaction among entities in the application workflow and business processes [A.10, A.18, A.23]. It can be seen as the dynamic part of the application. This layer is dependent on the persistence layer, in particular the domain model. Thus local changes to entity classes may require changes in other entities and also in business layer in relevant business rules.

The most notable issue with this layer is weak standardization of business rule management, which leads to issues with maintenance, specifically integration or overlapping of business rules that impact each other [8]. It can be observed that with OOD we cannot effectively capture a all the business rules that should apply for multiple entities [A.10, A.18, A.23]. [A.18] shows an example of a rule that triggers multiple entities to be read-only upon a given date. A base OOD approach leads to multi-location restatement of the rule and its distribution over the business layer makes such a rule hard to maintain. Such tangled code can be addressed via design that involves AOP [A.10, A.18, A.23], which we mention later in Chapter 3.

In a large application, it could be reasonable to use rule/reasoning engine for rule evaluation. An example of such an engine is framework Drools [33]. This framework builds on a pattern matching rete-algorithm [34]. Drools uses DSL to describe business rules and patterns that refer to entities and enforce system actions or complex inter-object validations. The DSL is translated to Java code and applies throughout the application.

Besides business rules, this layer applies security, transaction boundary, and in some cases session state of services. Various services are provided to the above layer. As a best practice, it provides Service Facades [3] that present service API to the above subsystem, so that no internal complexity exposes to other parts of the system.

The presentation layer presents both lower layers to users. It considers that end users use devices, such as personal computers, notebooks, tablets, cellphones, etc. Notably, it presents coupling to lower layers. The layer often divides workflow or task-flow through

various pages that present particular data in various forms [2]. It provides navigation among pages and control over system actions enforcing the workflow.

It is a common practice to build pages from hierarchical composite components [2, 3] that provide action listeners. These actions are being interpreted and controlled by controllers [6]. A UI component that provides visualization is known as a widget. Examples of widgets are buttons, combo-boxes, input fields, panels, etc. This thesis focuses on data presentation, which in EAs [8] represents significant part of the UI. Data presentation might be also referred as data visualization. Examples of these UI fragments are forms, tables, data iterations, data reports, etc. Besides these parts, the presentation layer of EAs also offers menus, navigations, buttons, action triggers, window layout, controllers, etc.

From the considered scope, the main efforts are placed on components that request and present data from/to user, forms. These components are very complex as they integrate data references, field presentation, user input validation, security concerns, form layout, business rules, context, etc. A particular user interaction with the system may also influence the resulting data representation in the UI. For example, a user that experiences difficulty in the interaction might be provided suggestions or hints on how to fill in data values.

When we look at this architecture from a high-level perspective, we want to see the interaction among the layers. The most common "happy path" life-cycle scenario is when an authorized user submits a page containing a data form. The submission is processed through a controller that converts data submitted through the HTTP POST/GET via widgets to object fields at the server-side and makes a decision on data/action validity; if the decision passes, then it initiates a service call to the business layer. The business layer may apply security rules, business rules and perhaps make sure the data fields are valid using complex validation rules. Then it persists the data object based on the entity class/field annotations. All entity fields have appropriate ORM annotations, which determine data persistence and constraints. Upon successful return, the controller decides where to take the user next and most likely redirects him to another page or presents him a success message. Failure usually results in an exception. Based on the exception, an exception handler either leaves the user on the same page indication the error or in a severe case redirects to a default page presenting the error message.

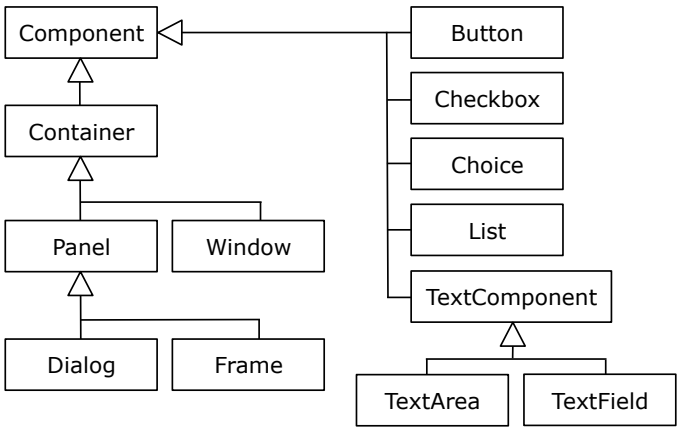

Figure 2.3: Subset of AWT Component hierarchy

2.2.3 Model-View-Controller Design Patterns

Many contemporary frameworks that deal with UI advertise the use of the Model-view-controller (MVC) pattern [3, 8], although they usually employ a variant of it. The aim of the pattern is to separate presentation of information/data from its definition and support reuse of the definition.

The idea behind MVC is to divide components into a model (in the context of 3-layer architecture represents persistent and business layer), a controller that mediates actions from the view, and a view that provides a presentation of a data from the model. In the context of the 3-layer architecture, both the controller and view are part of the presentation layer. When the user invokes an action in the view, the controller handles the action and may pass it to the model. The model accepting the action that results with data modification notifies all views about the change.

Fowler [8] points out that the pattern usually degrades to a passive version of MVC, where no notification from model to view happens on its update. Although, the MVC is common design pattern, many alternatives to it exist such as Model-View-Presenter, Presentation-Abstraction-Control, Model View ViewModel, Hierarchical-MVC, etc. [6, 8]. The pattern involvement does not impact results of our work, and we mention it mostly because of it usual involvement in the context of UI.

2.2.4 Component-Based Design (CBD)

The presentation layer in EAs involves both OOP practices and CBD, which is often further simplified with the use of DSLs. Plain development of a UI in OOP would be complex, low-level and impractical. Instead the UI is usually designed from existing predefined components, also known as widgets. The aim of every contemporary devel-

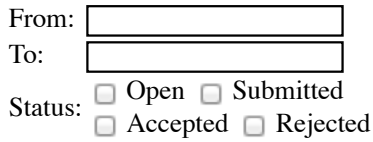

Figure 2.4: Sample JSF fragment [1]

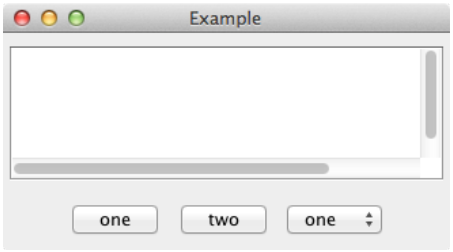

Figure 2.5: Sample AWT fragment

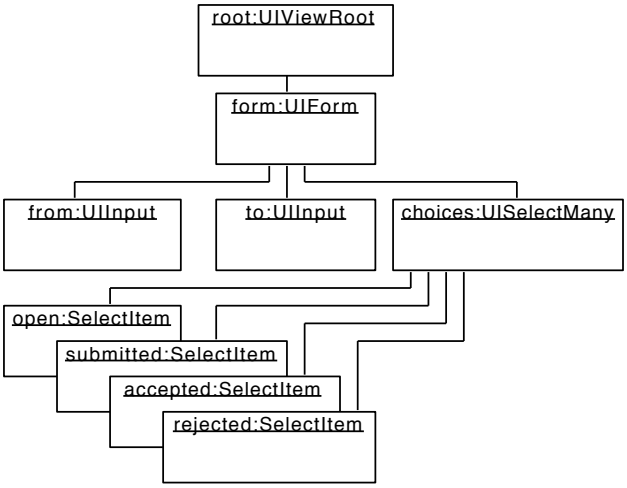

Figure 2.6: Component tree for Fig. 2.4 [1]

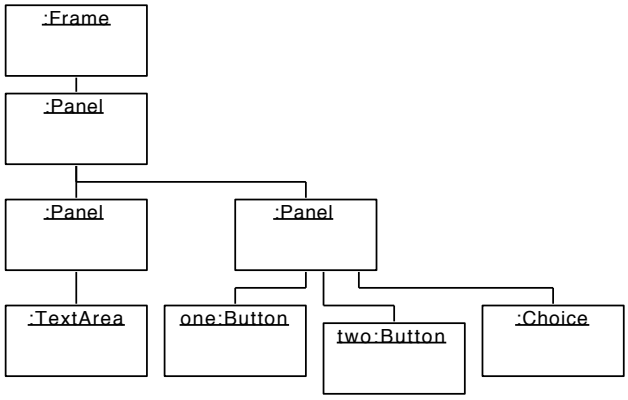

Figure 2.7: Component tree for Fig. 2.5

opment framework [1, 2, 3, 6, 8, 20, 29, 30, A.18] is to provide composition mechanisms and mostly simplification for the development and maintenance. The aim of CBD is to simplify the design through the use of generalized widgets that can be combined together to form the entire UI page or window [2].

Nowadays CBD-based frameworks, such as Java Swing, Abstract Window Toolkit (AWT), JSF, Wicket, Flex, .NET, etc., offer large collection of widgets, with various aims of presentation or composition. Each component has a given presentation purpose, although alternative components may exist with different appearance for user satisfaction. Components usually provide action listeners to propagate invoked events to page controllers [6, 8]. The advantage brought by CBD is that we can combine all sorts of components together, in order to achieve the expected presentation result.

For demonstration purposes we use two frameworks: fat-client presentation framework, AWT, and web-presentation framework, JSF [2]. Consider a subset of the component hierarchy for AWT in Fig. 2.3; similar hierarchy exists also for JSF. Note the inheritance is based on the composite design pattern [3], which allows component composition into a component tree. The component tree then represents a particular UI page or window.

Next, consider a basic example of a JSF page in Fig. 2.4 and an AWT window in Fig. 2.5. The component trees of these presentations are available in Fig. 2.6 for the JSF page and in Fig. 2.7 for the AWT window. For fat-clients, it is common to use GPL

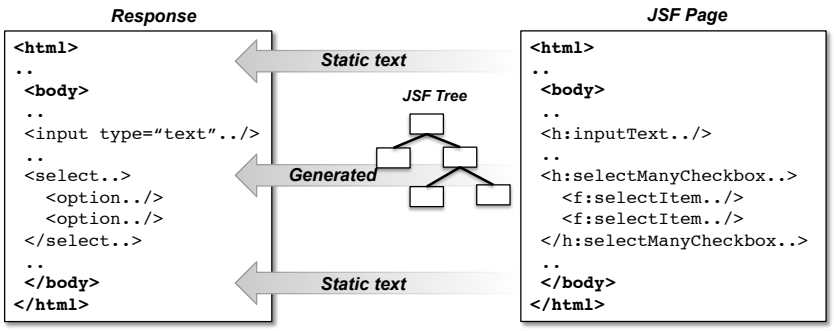

Figure 2.8: Rendering of a JSF Tree to the HTML response [1]

for component composition and specification. The web-based technology usually uses DSL descriptions for the composition and often allows its constructs to combine it with HTML. Google Web Toolkit (GWT) provides another approach for web frameworks. GWT, although a web framework, offers design similar to fat clients; it uses the GPL description, which it compiles into a JavaScript (JS) representation.

While the DSL tends to be more effective, developers might dislike its possible limited type safety as well as restricted options for inheritance and decomposition. At the same time, we cannot expect the GPL approach to provide full type safety. For example, we can reference objects, methods and fields, but we cannot reference annotations. In Section 2.2.2, we mention that contemporary standards for enterprise application development capture data constraints and restrictions through annotations over its fields. This plays an important role in the UI, because the widget selection is influenced by the constraint constellation, and the constraints must be restated in settings of each component in order to preserve the consistency of data requested from users and data persisted to the database. Consider for example an error in text length specification where the persistence class field allows persisting 50 characters while text length specification is omitted in a Text widget. The user assumes that he/she may provide unlimited description that is later transparently shorten throughout the persistence, and in the worse case, no error is reported to the user and important information gets lost.

Specifically for DSL, web-based frameworks consider the translation mechanism similar to one presented at Fig. 2.8 [2]. This figure presents a JSF code snippet at the right side that is parsed to produce a component tree (in the middle). This component tree is the subject of a transformation into the target format, HTML. The HTML is then interpreted at the client-side by web browsers. Such a mechanism allows us to take into consideration the target web-browser type and its capabilities and influence the transformation.

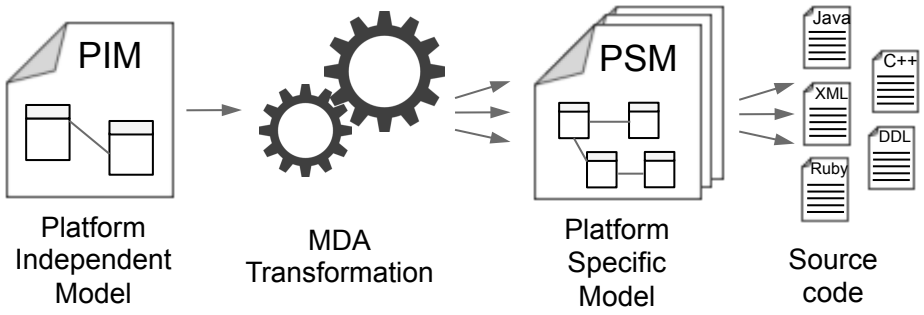

Figure 2.9: Model-driven architecture transformation

2.3 General Design Approaches

Besides the contemporary 3-layered architecture and CBD manual development introduced in Section 2.2, multiple alternative approaches exist. Namely, we look into Model-Driven Development (MDD), Generative Programming (GP), Domain Specific Languages (DSL), Meta-Programming (MP) and Aspect-Oriented Programming (AOP).

2.3.1 Model-driven Approaches

The first considered alternative to conventional design utilizes a model-centric approach called MDD. The model becomes the primary artifact of system development; application source code is generated directly from this model. There exists a standardization attempt that uses UML models called Model-Driven Architecture (MDA) [35]. The main advantage of the MDA approach is platform/middleware independence. We can capture the problem domain through platform-independent models, such as UML, and transform them to platform-specific models that reflect a particular domain in given platform. From these we can derive source code for the given platform. For example, consider Fig. 2.9 that sketch the MDA phases. When we try to apply the MDA approach, we use the UML model semantic and syntactic descriptions and their metamodels. Metamodels were already mentioned in relation to object concepts in Section 2.2.1. The UML metamodel defines properties and characteristics for every model language element.

In order to extend UML diagram functionality, an extension mechanism called UML profile exists [36, 37, 38, 39]. A UML profile is a package with stereotypes for specific meta-classes and tagged values for meta-attributes. Stereotypes in UML reflect the concept of code annotations, and tagged values reflect annotation attributes. The majority of statically-typed OOP languages have some form of annotations for class extensions that allow us to capture extra information or constraints in it.

The MDA suggests that all information should be captured in the model, and the code itself is solely generated from models. Unfortunately, there is a lack of production experience [A.2]; the majority of productions systems are code-based. It is obvious that a model that captures the entire system becomes too complex and with no doubt may include cross-cutting concerns on the model level [A.17]. Although we could use multiple models to capture various concerns and integrate them together, the appropriate generic integration mechanism is missing [17]. Furthermore, MDD approaches do not naturally fit to adaptive systems [40] because in such systems runtime information is needed to influence model-to-code transformation and MDD runtime transformation might be performance inefficient [41], which leads to compile-time derivation of large amount of possible states [40]. It may statically generate all possible application states and configurations for hypothetical/possible situations. In complex systems, this can grow exponentially. In [40] authors observe that MDD suffers during adaptation and evolution management, that minor changes to the system require complex redeploy.

With MDD we receive a system that fits to most of the situations, but for the edge cases, we may need to apply slight modification, which often takes place in code rather than in the model itself. Code re-generation from the higher abstraction model can become impractical since the manually added information gets lost [A.2, A.17]. A reasonable compromise can be seen in an approach where a certain subsystem is based on MDD and the rest follows the mainstream code design. Unfortunately, such an approach presents a burden when existing backend code contains information that are being restated in the model, leading to introduction of inconsistencies. In addition, often no verification mechanism exists that would inform us about inconsistencies. For example, consider a UI designed with the MDD approach using information from business and persistence layers. This approach will expect all information to be restated in the model; thus all later changes in these layers cause inconsistency with the MDD UI.

2.3.2 Automatic/Generative Programming

Another idea called automatic programming goes back to 1940 [42] when punched cards were used. Parnas, the main contributor of the concept, explains that the main idea is that a programmer defines a specification of what he wants and a computer produces the program in the language of the given machine. Thus automatic programming always has been a euphemism for programming with a higher-level language. Parnas concludes that research in automatic programming is simply research in the implementation of higher-level programming languages. The primary input to automatic programming

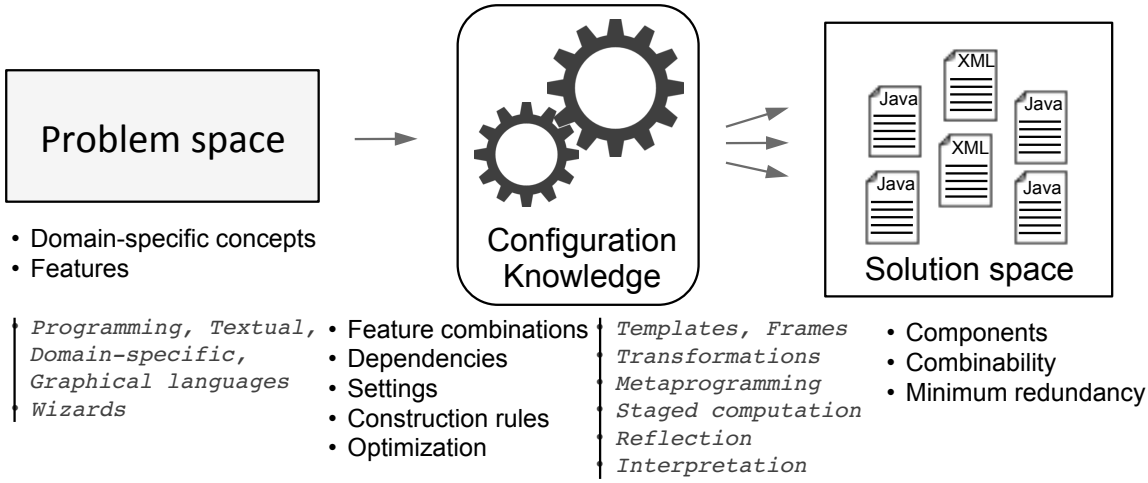

Figure 2.10: Sketch of generative programming concept

was supposed to be a problem "specification," but this over the time degraded to description of an algorithm, for efficiency reasons. Thus writing the specification is really writing a program. Parnas also raised a question on improvements, and believed that improved languages can lead to a reduction in the amount of detail that a programmer must handle and thus improvements in reliability. This must mean that the input specification is in non-algorithmic form. Although, even a mistake can be made in such non-algorithmic specification, thus automatic programming will not help with flawed specifications. Parnas indicated that the missing element in automatic programming is verification. This, although, cannot be solved by mathematical models that we would use for automatic programming.

The idea of automatic programming is not far from the alternative and later idea of GP [7, 12, 43]. GP emphasizes specific domain methods and integration to OOP. GP can be seen as programming that generates source code through generic code fragments or templates to improve a designer's productivity. Authors of GP [12, 43] define it as a design approach to combine and generate specialized and highly optimized systems fulfilling specific requirements. A GP sketch is provided in Fig. 2.10. The goal is to address the gap between program code and domain concepts, support reuse and adaptation, simplify management of component variants, and increase efficiency. Furthermore, it addresses principles such as *separation of concerns* [5] that suggest dealing with one issue at time and avoid code that deals with multiple issues at the same time. Next, it addresses *parameterization*, *separation of the problem space from the solution space*, which splits the problem space and its domain specific abstractions and maps it to the solution space with available implementation components, etc.

Authors of [12, 43] relate GP to DSLs and AOP, which we address later. DSLs focus on a specialized description of a given domain, which give designers concepts to work with the domain, but with loss of language generality. AOP addresses problems through decomposition to functional OOD units and aspects, and these weave together to obtain system implementation. Weaving takes place at compile time or runtime. Weaving often uses meta-programming, also described later. The aim of GP is wider than AOP; it emphasizes the automatic configuration and genericity, but applies at the compile time.

2.3.3 Domain Specific Languages (DSL)

The involvement of DSL for application design is remarkable and more details should be given. When we focus at a particular domain, we often find a language efficient for the domain specification and description [7, 27]. These DSLs are often taken as a counterexample to General-Purpose programming Languages (GPLs). Historically it is also known as application-specific, special-purpose, specialized, task-specific or application languages [27]. The use of a particular DSL is often limited to a given domain, while providing suitable constructs. The expected benefits are increased productivity and reduced maintenance. It may open the application specification to domain experts, who have no explicit programming skills. Such a language often builds on the top of XML. We may find their use in many domains, such as presentation description, with examples such as JSF or HTML, business rules [33], modeling, etc. In fact these languages may capture models introduced in Section 2.3.1. Unlike GPL, DSLs do not need to be executable.

In [27], authors suggest that DSL brings possibilities for analysis, verification, optimization, parallelization, and transformation in terms of their constructs that would be much harder or infeasible in GPL. They also emphasize that it is common to provide a visual model to represent the DSL, while the textual description makes program and specification composition easier for large systems. When aiming to design a new DSL, either it is possible to build on existing languages or it is necessary to determine new DSLs relationship to the applied GPL, define constructs their composition mechanisms and their semantics. To execute the specification given by DSL, it needs to be compiled, interpreted [3], and the output can be generated or the result can be embedded (in such cases it is better build on the top of GPL - host language). When DSL embeds its result, the developer must face issues with error reporting and with performance. Furthermore, we must beware that when we aim to apply DSL together with underlying

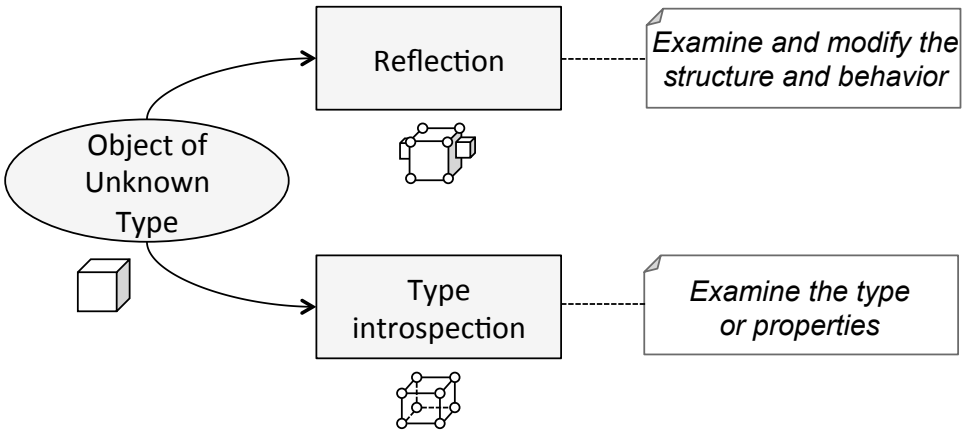

Figure 2.11: Observing and object of unknown type through reflection and introspection

system, there is limited type safety, and thus changes to the underlying system does not directly propagate to the DSL specification. Information are often restated [28], and DSL does not address cross-cutting concerns, although it is commonly used with AOP to describe aspects [18].

2.3.4 Metaprogramming (MP)

So far the approaches did not address information reuse from existing structures, although reuse would be very beneficial for maintenance. Assume that we use the design approach from Section 2.2 and we want to reuse information captured in low layers. MP gives a program or a language an ability to examine or modify both its structure and behavior at runtime. Many contemporary, statically-typed programming languages have the ability to describe them, called Reflection [44], based on an architectural design pattern [45]. It is further capable to observe object properties, methods, annotations, etc. (see Fig. 2.1).

For object examination, there is only a subset of the reflection ability needed, called introspection. Introspection is capable of examination of object properties at runtime. Fig. 2.11 gives a basic overview. This mechanism allows the program to dynamically adapt to different situations. Among the disadvantages, we can emphasize that performance becomes an issue for this type of approach. Another issue is code readability as we loose type safety or harder maintenance of reflective programs. MP is a great instrument for code inspection and information extraction. This approach is commonly used by contemporary frameworks, for instance for ORM. The way it deals with performance bottlenecks is that it uses a cache or code generation upon application deployment [46].

2.3.5 Aspect-Oriented Programming (AOP)

Features we discovered as missing or complex for OOD, MDD, GP, etc., are addressed by the AOP approach [4, 13, 18]. It suggests to capture different concerns in independent code fragments called aspects and considers their compile time or runtime weaving. The runtime weaving may depend or be influenced by the runtime context. The integration mechanism is well described and generic. A given problem is decomposed to functional OOD components and aspects, and these are joined together through aspect weaver. The integration of aspects into in the OOD components considers dedicated locations in the OOD components called join points.

For a demonstration, consider Fig. 2.12 that on the left side shows the usual compilation of the OOP code and the alternative with the AOP on the right side. Notice that the left side OOP code has larger volume and interleaves various colors, consider that these colors are various concerns that this case captures tangled together. The developer is responsible to maintain the entire tangled code fragment. On the other hand the AOP aims to strip out the concerns from the OOP code into aspects that are separately defined in GPL or DSL. Such separation supports their reuse across different OOP components. Consequently the volume of the OOP code decreases and becomes easy to read, highly cohesive. The aspect weaver is instructed by OOP code join points how to integrate aspects, defining other concerns, into the OOP component. A join point can be for example an annotation or a method name. The result of weaving, as shows Fig. 2.12 on the right side, can be an OOP component that is further compiled by regular compiler. The advantage for the developer is the concern separation, code reduction, readability and the same compiled result as when is all concern tangling done manually.

Join points are crucial elements of the AOP as they indicate concern integration. They are recognized by the aspect weaver either dynamically upon code invocation or at the compile time (as demonstrated by Fig. 2.12). Based on a given context, an aspect may or may not integrate to a given location at the OOD unit. 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 good readability and maintenance.

To provide a more specific example of the AOP consider Fig. 2.13. It shows 3 methods on the left side. Each method tangles concerns such as main method logic, security, synchronization and logging. When considering the left side image we see tangled concerns in the method source code when using OOP. Such method is hard to read

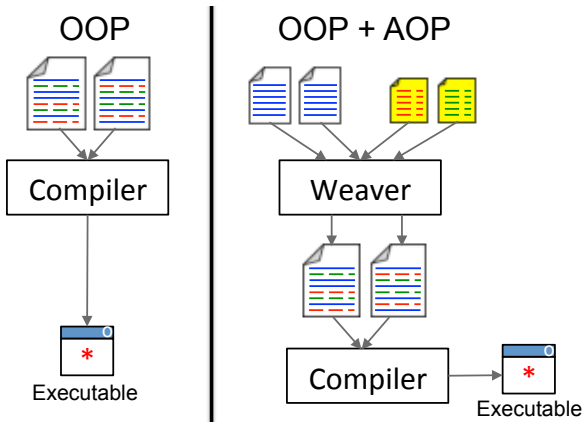

Figure 2.12: Compilation of a program that involves aspect weaving

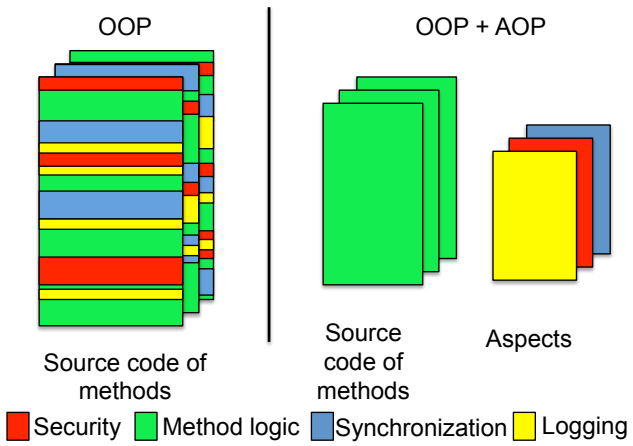

Figure 2.13: Demonstration of cross-cutting concerns tangled in OOP and their untangling in with AOP

and its cohesion is low. The right side of Fig. 2.13 uses AOP and shows the separation of concerns as well as use of the OOP for method logic and AOP. The example well demonstrates the advantages but it omits the fact that sometimes the OOP component requires a slight extension with join points, such as additional method annotations.

In the primary AOP research [4] authors demonstrate that AOP design has the ability to significantly reduce the application total Lines Of Code (LOC) and development efforts. In their example, they reduce a complex, optimized application that tangles code with 35'213 LOC into an AOP-based application with the same features and optimization with fewer then 1'039 LOC.

Contemporary AOP frameworks can be described with a conceptual model. Such a model has three main components [18]:

Join Point Model: defines available join points for adaptation

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

Adaptation Mechanism: allows to add or modify 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. We can recognize static and dynamic join points. A static join point can be characterized as a location in the program's source code [18] and thus characterizes application static structure. Each static join point can be reached multiple times upon program execution, and the dynamic join point then represents a single hit to this join point during the program execution [18]. The difference becomes more obvious when we consider a changing application runtime context that plays a role in the adaptation. Whether an aspect adaptation should take place or not is decided based on evaluation of an expres-

Table 2.1: Comparison of design approaches

Ability/Approach	OOD	MDD	GP & DSL	AOP	MP
Compile time convenient	yes	yes	yes	yes	yes
Runtime time convenient	yes	slow	no	yes	yes
Separation of concerns	no	no[2]	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[1]
Adaptable towards changes*	no	no	yes	yes	yes
Transformation based	no	yes	yes	yes	no
Synergy with OOD	-	no	yes	yes	yes
Platform independence	no	yes	no[3]	no	no

*when considering existing code structures/application backend
[1] the MP code itself, but enables good management to the rest of the system
[2] it can address them, but a generic integration mechanism is missing
[3] DSL can express a model for MDD

sion in the pointcut language that may involve the structure and also the application context. Example adaptation mechanisms are advices in AspectJ or composition rules in Hyper/J.

To receive the static structure of a program, often the weaving involves metaprogramming mentioned in Section 2.3.4. The difference with GP [43] is the scope. GP scope is larger than AOP and involves DSL, model, etc., although AOP can use DSL for aspect description as well. AOP is more general and has generalized integration constructs. GP emphasizes compile time automatic configuration; however, we must also see a wide intersection in addressing cross-cutting concerns, adaptation and configurability.

2.3.6 Summary

Let us summarize mentioned approaches in Table 2.1. We compare the selected abilities such as whether it applies to runtime, addresses separation of concerns, reduces restated decisions or information from existing structures, applies inspection, handles evolution management, adapts easily towards changes or existing structures, and is compatible with OOD. It can be seen that none of the approaches addresses all considered criteria. In our consideration, we mostly aim for a runtime solution, addressing restated decisions and information, easily applying existing information with proper evolution management, while addressing cross-cutting concerns, and at the same time synergy to OOD to enable easy integration to current mainstream design of information systems and EAs. As oppose to other approaches MDD brings the possible benefit of platform independence.

3

Related Works

Don't reinvent the wheel, just realign it.

-Anthony J. D'Angelo

The area of UI development is broad, and thus it includes many related works based on the approaches introduced in Section 2.3. This chapter is divided into three sections. The first section introduces and classifies UI design approaches. The second section introduces Context-aware User Interface (CaUI) in the area of HCI. The third section describes approaches that deal with content delivery that may improve UI performance and responsiveness for web-based applications.

3.1 UI Design Approaches

This section describes existing UI design approaches and provides their classification. First, the manual approach is introduced. Next, widget builders are discussed followed by model-based UIs, DSLs and GP for UI, and MP approaches to UI. All approaches are also compared based on selected criteria in the summary.

3.1.1 Manual Design for Data Presentation in UI

A nice introduction to EA development is provided by Fowler et. al. [8] and mentioned in Section 2.2. Fowler suggests that EA usually involves a large volume of persistent data, “illogic" business rules, and UI screens to handle the data workflow.

In case UI fragments for data representation are designed manually, the developer must inspect application data, their fields and constraints by hand and select appropriate components/widgets for data property representation based on a given selection criteria. Each component has additional settings influencing its representation or behavior and reaction on user input.

When data are represented as objects, appropriate components are selected for their fields. An individual component setting representing the field also relates to field constraints. The restatement in the UI representation is caused by the incompatibility of the data object with the UI language, possibly DSL. This might seem to be a minor issue at first; in fact, it is a significant problem. Consider what happens when the system evolves and data change, a common scenario with application development; all UI components referencing the data must change as well. When we use a type-safe language, we might receive immediate feedback from the compiler, although some component properties are text or numeric rather than verifiable references. For example, consider that a static reference to a field annotation or its parameter is not available in GPL; such information is then restated in the UI component. The situation becomes worse when the UI framework uses a DSL to describe the UI, for example JSF. Such an inconsistency between the data object and the UI part might be discovered after production deploy.

Information restatement brings negative consequences. When the restatement requires manual efforts, the process becomes error-prone and tedious and the maintenance becomes difficult. Next, there is an increase in coupling among application elements, and there might not be a mechanism to prevent inconsistency for languages with weak type safety or to verify the integrity.

Further details concerning to conventional UI development are provided in [A.4, A.17]. The main problem arises from tangled concerns in the UI fragment source code. As mentioned in Chapter 1, to design an appropriate presentation we must provide data-binding and consider user input validation that plays a role in the component selection and component setting. What if there exist different validation rules for the general

public and an administrator in our application or when validation rules change with the time scope? It must be considered that different users might have different rights and perhaps can see different data fields. Furthermore, what if we want to provide a user-friendly UI such that we offer a wide layout for users with wide screens and a standard layout otherwise? We have explored in Chapter 1 that these concerns are cross-cutting, and we do not have effective mechanisms in OOP [18] that would allow us to capture and consider these concerns separately, which greatly decreases fragment reuse, since the component tree considers all these at once. As an example, consider multiple layout variations for the same form. It is apparent that we would need to copy the entire component code fragment and replace parts representing layout.

A skeptical assumption is that Extensible Stylesheet Language (XSL) transformation can solve the problem with layout, but the CBD framework mechanism may prevent it. For example in JSF, the component tree renders to HTML (Fig. 2.8); thus any Extensible Stylesheet Language Transformations (XSLT) process would have to be done internally, i.e., an internal pre-component tree would have to be composed, transformed through the XSL to the component tree and then rendered to HTML. Instead, we need to consider that the layout is a single concern and other concerns may be changing as well. Later in Chapter 6 more details are provided along with a solution to address separation of UI concerns.

To summarize, there exist two core problems with conventional development of UI fragments representing data. First, we must restate information from the data elements in the UI elements. This presents a considerable problem for UI inconsistency; for languages with weak type safety, this becomes a burden. Second, UI fragments representing data combine multiple concerns at once, which prevent it from reuse and flexibility to adjust the UI to various conditions or situations. As a result, we must copy the original code fragment apply changes to it and keep up with multiple similar code fragments varying in details.

3.1.2 UI Widget Builders

Specifying UI through code descriptions may be hard and complex, especially for developers new to this kind of UI framework. Very often a simplification is provided in the form of UI widget builders [47, 48]. Such builders are tied to the component development framework and provide an easy visual editor with drag-and-drop functionality, composing the component tree, while seeing the visual representation of the UI that is

being designed. Very often we may find these editors to be an extension of Integrated Development Environment (IDE) such as Eclipse, NetBeans, IDEA, etc.

When dealing with data representation, we can select components to represent data fields, drop them to a given layout, and even select component constraints and set data binding. The main focus of these builders is a one-way approach from the visual editor to code, although, basic code constructs can be also recognized in the visual editor [48]. The reverse approach is limited to base constructs that lack custom, parameterized constructs or dynamic UI code based on runtime calculations. More sophisticated builders are capable of initial component generation based on given data reference [48, 49, 50]. We consider such generation rather basic or simplistic as the designer has to explicitly specify which fields to visualize, etc.

Widget builders certainly help with the initial design. The produced UI source code is often rather complex and its volume is large. Manual changes to the code and refactoring [6, 8] that uses more advanced, generalized or parameterized code constructs are not interpreted by the builder in the reverse direction. The maintenance of the UIs then becomes as complex as with manual design, or even more complex since the resulting UI code from the builder is extensive and lacks advance constructs. The builder-generated UIs couple together all concerns that we considered in previous sections; thus a single page integrates all the layout, field presentation, and constraints. In case we want to support conditional rendering, these builders do not help us, and we must touch the source code, which may disable future builder use. Even builders, upon code-changes, do not help with restated information, and thus data modification that implies changes to the UI must be propagated by hand. They neither help with system evolution nor support adaptive features for the UIs. Since these widget builders do build the UI representation as a component tree, they cannot handle cross-cutting concerns well. Furthermore, the generation is processed before or at compile time.

3.1.3 UI Model-based Approaches

The model-centric approaches are also used for UIs. For example, [51] provides a survey on model-driven tools for HCI and emphasize the impact on reduced efforts for design of interactive systems. They classify tools on how they capture metamodels and models, whether they use graphical or also a textual representation, and what they use for constraint definition (most of the time Object-Constraint Language (OCL) [52]). Authors of [53] suggest that the gap between HCI and System Engineering could be

addressed with MDA. They raise the argument that part of the scientific community suggests that relating design models to the user interface might be considered a mix of presentation and persistence logic; however, their observation is that HCI concerns cannot easily be described independently from other concerns. A skeptical view on MDD in the UI area suggests [54] that despite considerable research, model-based UI tools have not become common, in part because building models is an abstract process and better results are often achievable by a human designer in less time.

In Section 2.2.2, we introduce standards used by application frameworks such as JPA [9] for ORM or Beans Validation for User Input Validation (UIV) [10]. It seems that in MDD we would need to re-invent these. We could consider suggestions given in [22]: when we want to design an approach that would be applied for contemporary systems, we cannot expect industry to make very large changes to processes and methods. Research is expected to build on the work of others, using existing standards. Fortunately, is possible to extend the UML models by profiles [35] for situation like this. An Model-Driven (MD)-JPA UML Profile for class diagrams is provided by [55], which enables us to capture persistence aspects on the model level. Furthermore, in our work [A.2], we design UML profiles for validation, security, and presentation.

In [56], authors aim to design CaUI through MDD, but for such UI we need runtime information and UI generation at runtime [40], which may impact performance [41]. Static generation can lead to many states, and their variation can grow exponentially. Furthermore addressing cross-cutting concerns [40] may become an issue since generic, multi-model integration mechanism is missing [17]. [57] notices that MDD often uses specific or hard-coded transformation rules, which degrades its reuse; instead generic and configurable rules should exist [57, A.17], to support reuse among systems.

3.1.4 Generative Programming & Domain Specific Languages for UI

Pioneering work applying generative programming to UI [7] demonstrates the use of abstract specifications for UI generation. The authors suggest that such application consists of three parts: *a DSL for UI description*, *configuration generator* that automates the product assembly by taking the DSL specification and assembling the implementation components from it, and *an extensible collection of elementary components* available for the assembly. Such an approach then allows producing a large number of system variants based on specific requirements. The authors then suggest splitting the analysis of features and their simplified capturing in a DSL, which reduces the required

knowledge from the implementation language. They notice that it is highly possible to make mistakes, such as typing errors, while writing the specification. The main benefit of the GP demonstrated by a case study at [7] is the possibility to combine two hundred features in UI, which gives the variability of 5×10^{17} prototypes. On the other hand, it is questionable whether all such prototypes will ever be used and whether having statically allocated all these prototypes is reasonable. However, GP does not help with information restatement and lacks the ability to consider runtime information.

The DSL area itself is very broad and commonly used in the industry, e.g., consider JSF, JavaServer Pages (JSP), Wicket and many other DSLs for Java EE (and other languages). Although many researchers aim to simplify the UI description even more [58], the main drawbacks of these approaches stems from the duplication of source information, additional maintenance efforts when source information changes, limited constructs to capture cross-cutting concerns separately, or lacking support for variable UIs and often weak type safety.

3.1.5 Meta-programming UI

Multiple research proposals such as [19, 28, 59, 60, A.15] utilize automated UI generation by applying code-inspection or rather MP. These approaches inspect information captured by lower layers, then build an ad-hoc structural model, and transform it to the UI. This simplifies both 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 [A.2]. Experience from industrial standards such as Java EE [9, 10, 16] show that the domain model already captures additional markup for persistence and validation constraints. The same approach can be applied also for presentation [A.2] and security. Naturally, such an approach can generate basic UIs, but more complex UIs or even adaptive UIs can become difficult.

At the same time MP does not address an UI transformation or a proper context structure. Furthermore, it does not deal with output variations, cross-cutting concerns, or context-awareness, although, it can adapt the UI to changes in the underlying structures. Naturally, MP seems to well address information reuse from existing structures and can be used as a part of more complex solution [A.4, A.17].

3.1.6 Aspect-based UI

In [40, 61] the authors point out cross-cutting concerns that play a role in UI and attempt to apply aspect-oriented techniques together with model-based approach to support multiple degrees of variability that depends on user context. Although, only limited amount of information is exposed on the applied approach. Generally, the research of AOP applicability to the area of UI is rather sparse. This thesis and our previous research [A.1, A.4, A.17, A.21] explore more details and aims to formalize the AOP-based UI approach. More details are to be found in the upcoming chapters.

3.1.7 Classification Discussion

Kennard et. al. [60] suggests that UI design approaches could be divided into three groups: interactive graphical specification tools, model-based generation tools, or language-based tools. Interactive graphical specification tools are described in Section 3.1.2, and allow developers to sketch UIs on the screen while in the background they generate corresponding source code. Model-based generation tools use a model describing the UI, which is then transformed to an appropriate UI presentation (see Section 3.1.3, Section 3.1.4 and current related research in Section 3.1.6). The language-based tools suggest deriving UIs from the language and domain objects, as shown in Section 3.1.5 as well as Section 3.1.6.

In our research [A.17], we consider another point of view for UI design classification. We consider an approach that we call *restate-to-extend* and *inspection-based* approach. The *restate-to-extend* approach requires that the same information in a system be captured twice at different locations, to preserve its integrity. This rises from the technological inability to effectively reference information. The information duplicity is applied to a particular concern in the UI, such as presentation, validation, etc. Development using this approach typically involves interactive graphical tools, model-based generation tools [62, 40], external models for UI representation [14], but also DSL tools [58]. The main drawback of these approaches stems from the duplication of source information and additional maintenance efforts when source information changes. *Inspection-based* approaches use existing information accessible by code-inspection or MP. 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. However, both *restate-to-extend* and *inspection-based*

Table 3.1: Comparison of UI design approaches

Criteria/Existing UI Approaches	Manual	Widget builders	MDD	GP	DSL[@]	MP	AOP
Address cross-cutting concerns	no	no	weak	yes	no	no	yes
Reduces restated co-exist. information	no	no	no	no	no	yes	no*
Fast initial design	no	yes	yes	no[1]	yes	yes	no[12]
Full output customization	yes	no	no	yes	no	no	yes
Reduced system maintenance efforts	no	no	no	yes	yes	yes	yes
Reduced code volume	no	no	yes	yes	yes	yes	yes
Compatible with third-party	yes	no	no	yes	no	yes	yes
Customizable UI transformation	-	no	yes	yes	no	no	yes
Generic/reusable UI transformation rules	-	no	no	no	no	no	no
Runtime transformation	-	no	slow	no	yes	slow	yes
[@] *(assuming high-level DSL not low-level)* [2] *(non existing weaver - values below this row are assumed)* [1] *(unless a compiler/weaver exists)* * *(apart from MP)*							

approaches need to deal with information transformation to the UI. As suggested in [57] and [A.17], a generic approach should be used. Furthermore, neither of the above approaches directly address cross-cutting concerns, although related research on this topic exists for both model-based [40, A.21] and GP approaches [7].

3.1.8 Summary of UI Design Approaches

In the above section, we introduced contemporary UI design approaches. We looked at the component-based development of UI involving manual coding, widget builders, MDD, GP, DSL, MP, and AOP. We evaluate the existing work in Table 3.1, although a small variation may exist among existing work in given approach. As evaluation criteria, we consider whether cross-cutting concerns are addressed, reduction of restated information from co-existsing structures, fast initial design, ease of customization of the output, reduction of maintenance/code volume, compatibility with third-party frameworks, customizable/generic transformation rules, and (most important) runtime consideration. The columns of Table 3.1 represent existing work in given approaches, and rows are evaluation criteria. The AOP research is sparse; thus we estimate the result. We can see that neither manual development nor builders address many of our criteria. MDD and DSL address multiple of our criteria. More promising approaches although seem to be GP and MP. Theoretically AOP may have the most promising attributes for UI design, although weaver does not exist (not considering the one from our work introduced later). Also a given weaver implementation can strongly impact the evaluation. The ideal design may combine multiple strong parts from various approaches, such as MDD transformation, DSL description for UI, GP integration of the DSL, MP for code-inspection, or AOP for runtime integration of cross-cutting concerns.

3.2 Context-aware UIs

A basic overview of adaptability and adaptivity is provided by [63]. Both terms refer to knowledge-based self-adaptation. Adaptability can be deduced before the interactive session begins. On the other hand, adaptivity relates to an interactive session. CaUI may address both these features. The idea of such UIs is studied in multiple domains.

For example, we can see its application to [64] electronic cooking assistants in kitchens to adjust layout, in a hospital navigation case [14], and in a house control unit example [61]. Most of the existing work focuses solely on presentation, adaptability and adaptivity features. Typically they apply model-based approaches and thus restate information from application backend. None of the related approaches provides a study or an evaluation regarding runtime performance or production experience. In fact they rarely consider maintenance efforts.

Multiple CaUI design methods require the target environment and possible variations of the user interface at the design time [62, 65], but future, more advanced, adaptive systems need to consider runtime information as well as suggested by [66]. CaUIs may receive benefits from application of aspect-oriented techniques. The existing work [40, 61] extends a model-based approach to deal with multiple degrees of variability that depends on user needs and context.

From the architectural point of view, most of the related works in this area use model-based approach. Specifically they divide the model into an abstract and concrete part for the UI [A.21]. Sometimes there exists an interaction module [61]. Furthermore, there might exist inconsistencies among these parts. Bi-directional mappings between the abstract and concrete UI are suggested [67] to avoid such inconsistencies. To further avoid information restatement among models, [57] suggest using model-to-model transformations. In addition they suggest to avoid excessively detailed adaptation rules since they prevent reuse; instead a set of generic mapping rules should be used.

In [56] the authors suggest a task-based approach and use Concur Task Tree (CCT) [68] to describe tasks. Such an approach allows authors to distribute UI to multiple diverse devices. In [69] the authors divide the context into user, platform, and environment and then apply an ontology-based framework.

CaUIs may also deal with automatic field position optimization for concrete UIs [A.21, 70]. This can be seen as an optimal distribution of widgets on the screen to support usability, although such optimization requires addition criteria per field detailing its

Table 3.2: Comparison of related work

Features	[64]	[14]	[61]	[66]	[62, 65]	[40]	[19, 60]	[A.2, A.15]	[7]
Model-based	-	-	+	+	+	+	-	+	-
Runtime approach	+	+	+	+	-	+	+	+	-
CaUI	+	+	+	+	+	+	-	-	+
Reduces code	-	-	+	-	+	+	+	+	+
Avoids restate-to-extend	+	-	-	-	-	-	+	+	-
Addresses cross-cutting concerns	-	-	+	-	-	+	-	-	+
Ad-hoc model construction (code-inspection)	-	-	-	-	-	-	+	+	-
Uses enterprise technology standards	-	-	-	-	-	-	+	+	-

importance for given context, which increases maintenance and development efforts. Note that the optimization itself also degrades the application performance (in range of seconds) [A.21].

Related research addresses presentation, but also an interaction flow and navigation, although this is beyond the scope of the research presented in this thesis. For our approach, we consider that the interaction mechanism is managed by a host framework. For example JSF uses a DSL page flow, or a broad variety of navigation and interaction options are provided by specialized frameworks such as jBPM and Drools Flow [33] that comply with our approach and can be seen as the production-level alternative to the CCT.

The summary of considered related work regarding the CaUI design is provided in Table 3.2. The Table 3.2 evaluates [7, 14, 19, 40, 60, 61, 62, 64, 65, 66, A.2, A.15] from various feature perspectives. The first row shows whether the considered research bases on models. Next, whether it applies runtime UI generation. The below row shows whether it intends to address CaUI. The following row indicates whether the used approach aims to address code reduction. The next rows shows whether it avoids restating information. The perspective of separation of cross-cutting concerns is given next. The previous last row shows, whether the approach applies code inspection. The last row indicates whether it complies with existing developments standards such as Beans Validation [10], JPA [9], etc. It can be noted that none of the approaches has the ability to provide all the features that in consequence reduce development and maintenance efforts and give the flexibility to address CaUI.

3.3 Content Delivery in Web-based Applications

The standard client-server communication for EAs and web applications bases on the HTTP protocol. It provides core mechanisms to improve the transmission. First, the underlying TCP-based protocol supports connection persistence, so that multiple resources can be loaded from a single server using the same connection, to avoid connection reopening. Multiple parallel connections usually exist from a client to a single server. HTTP supports content compression to reduce the transmission content size. Furthermore, it supports resource caching at the client-side with time-based invalidation. The caching usually applies to static resources such as CSS, images, and JS. HTML5 brings an option to cache JS calculated results and results of web-services through a local caching mechanism called Local Storage. [A.19] shows that an average contemporary web system consists of about 90% static, cacheable resources. To further reduce the transmission, UI developers may apply resource content obfuscation and resource merging [A.20]. To mitigate the impact of client distance, servers may apply geo-distributed caching of static resources called Content-delivery Networkss (CDNs), such as Akamai [71].

The majority of existing UI approaches provide UI descriptions is a way that various UI concerns tangle together, which impacts the transmitted volume, limits reuse, caching and abilities of the client-side [A.1]. Alternative delivery options are considered by GWT [72] and AngularJS framework [73].

GWT brings an advanced idea in the way it works with actual data values. Instead of delivering data value tangled with the UI description, it streams them separately to the client in JavaScript Object Notation (JSON) format. The client-side integrates the received data to the UI. A similar idea is adopted in AngularJS framework [73], which separates data values from the delivered UI description and uses templates that integrate the provided data values at the client-side.

We look at the GWT in more detail. It suggests describing the UI using type-safe Java language (GPL). Such description compiles and generates all UI states to the JS UI description, although this may affect the transmitted UI volume. Consider the use of the MDD philosophy, and its benefits and limitations. The positive effect can be seen in the offline functionality, but the overall transmission volume might be extensive considering CaUIs [40]. The MDD nature of GWT brings the possibility to address incompatibilities among web browsers, and consider multiple transformations

generating different variants of JS descriptions. GWT as mentioned above considers separate delivery of data values, but also considers that part of the generated JS UI description is static and cacheable and part is dynamic. From the above is apparent that GWT application usually send the client UI description with all the states, which fits to applications such as interactive consoles, email clients, etc. But the use for EAs or information systems might not be the best fit [A.1].

Alternative option explored later in this thesis [A.1] considers separated delivery of UI concerns. This allows transmitting only the actual state needed by the client and this fits better to EAs. When dealing with exponential growth of UI states, GWT compiles them to a particular descriptions, the proposed alternative targets separation of concerns at the delivery level, which allows to provide concerns individually, and thus avoid the negative impact on the extended transmission volume.

Approaches addressing HTTP optimization exist. For instance Structured Hypertext Transfer Protocol (STTP) [74] extends HTTP to include new messages to control the resource transmission for a particular web page. A similar approach, HTTP-MPLEX [75], employs a header compression and response-encoding scheme for HTTP. Similar to STTP, it multiplexes multiple responses to a single, sustained stream of data to speed response times and improves application layer use of TCP. While experiments show performance improvements with these protocols, they do not consider resource distribution through CDNs, caching, or web-page variations.

An alternative to a CDN involves the use of a Cooperative-Web cache (CWC) [A.19, A.9]. The CWC involves a Peer-To-Peer (P2P) overlay network where clients interact together to redistribute cached resources among each other. Considering a situation with distant server and two clients relatively close to each other, this may bring benefits. Unlike CDNs, CWC supports natural scalability, resistance to flash crowds and free-of-charge P2P services; however, it must deal with content invalidation in the overlay and mechanisms to detect and disable malicious clients from sharing corrupted data. Since no centralized authority for content validation exists, distributed approaches must be used. Our preliminary results has shown that such an approach improves page loads, since it is possible to detect the closest client via anycast. Considerable reduction of resource allocation and Central processing unit (CPU) use can be seen at the server-side with this approach.

4

Research Roadmap

Programs must be written for people to read, and only incidentally for machines to execute.

- Hal Abelson

Structure and Interpretation of Computer Programs

The aim of this chapter is to analyze the related work and expose problems that are not solved. It also provides a roadmap to accomplished research and introduction to the next chapters.

4.1 Summary and Analysis of the Related Works

Chapters 2 and 3 provides a broad overview of the related work in the area of general design approaches plus UI design approaches as well as an overview of context-aware design from the software engineering perspective or the UI transmission to clients. The general design approaches compared in Section 2.3.6 at Table 2.1 show that none of given approaches provide all the considered abilities. It is possible to combine these ideas. For instance, consider that the idea brought by MDD with model and transfor-

mation could apply to the code-based approach as well. The model could be extracted from code through MP at compile time and then transformed. The transformation could involve templates described through DSL. Although the transformation usually follows specific transformation rules, they could be relaxed to become more generic. For instance, consider that each rule resolves a certain conditional clauses that determines whether it applies or not. Similarly, the template resolution can directly inject content from the model or indirectly resolve certain conditional clause that decides whether or not to extend the template with certain information. When we move to AOP, it can provide the mechanism and terminology for this indirection. The clause can be a point-cut, the decision an advice, and the clause term can be a join point received from the model. The same join point can also operate as the information extending a template. Notice in the summary in Section 3.1.8 at Table 3.1 that existing UI design approaches provide only a subset of possible the quality criteria; there is room for improvements.

The context-aware UI approaches in Section 3.2 give us a sense of what kind of concerns can be considered. The context-aware UI research deals not only with data presentation, but also with page-flow or task-flow, which is within scope of this thesis. A large amount of related work focuses on MDD, but most utilizes the restate-to-extend approach with only rare emphasis on the separation of concerns, although this should be essential. Additionally, the related work suggests that it is important to consider runtime context; thus a future approach should operate at runtime to adjust the output to the runtime to reflect context changes.

UI responsiveness and the performance related to page load time is an important quality that must be considered regarding the UI design. For instance, if a UI page loads in 7 seconds, the end user may decide to look for an alternative, more responsive application, willing to accept less functionality or features. Section 3.3 considers UI performance of web-based applications. It introduced existing approaches that usually deliver tangled concerns in the UI description in the server-client communication. Tangled description is usually compressed to reduce the transmission efforts, but when only single UI concern changes, the entire tangled description must reload.

4.2 Accomplished Research and Roadmap

This section provides a survey of conducted research that applies to the scope of this thesis. The research aims to present system data in UI in varying context. The UI data presentation is usually given by forms, tables, reports, etc. The initial emphasize

of the work is on the reduction of development and maintenance efforts, although as shown later, addressing these efforts through the separation of concerns brings further benefits to reduce repetitions, to provide CaUI features or an alternative UI delivery to remote clients, while improving client-caching options and UI responsiveness.

The first problem this thesis addresses is the UI design complexity related to data presentations. UI descriptions of data presentations are usually highly coupled to data definitions and have low cohesion. This leads to difficult readability and extended maintenance. Manually-designed UI data presentations are self-descriptive, but suffer from human errors introduced because of large information restatement [28]. Furthermore, often UI presentation of particular data differs for various users and contexts because of access rights, screen size, etc. This may result in multiple UI variants that have physical description and only differ in details, but all are sensitive to changes in data definitions. In order to address the issue with information restatement, we look in this early stage of the research, [A.2, A.11, A.13] to the advantages brought by the MDD approaches. In particular we consider involvement of UML class model as the source of information for UI data presentations.

Since the UML class model does not contain enough information to derive satisfactory UIs, an extension to the model through UML profiles is suggested [A.2]. A UML profile allows capturing addition information at the model level. Research [A.2, A.11, A.13] suggests defining reusable UML profiles for capturing presentation, validation, security and persistence in the UML class model. The advantage brought by the extension is the ability to describe the above information at the model-level and enable its derivation for the UI description-code transformations.

[A.2, A.11] and Chapter 5 demonstrate advantages brought by an MDD approach. The source of information is the model, and thus UI derivation from the model reduces information restatement. It is also possible to derive variations of UI descriptions. Another advantage is that all changes to the model apply in the UI upon consequent model-to-code transformation. The consequence brought by this is that UI description for data presentations may enforce security, avoid consistency errors, and reduce development and maintenance efforts. It is fairly easy to apply data constraints to the UI description for the client-side input validation, etc. Such a model is not specific to a particular platform, and thus it is possible to apply it to Java, C#, PHP, etc. The UI generation is done at compile time. Regarding the problems introduced in Section 1.1,

this approach addresses Problems 1 and 2, partially addresses Problem 4 and brings a base ground to Problem 9.

The disadvantage is that the derivation of the UI needs to use a model, which might require defining a new model for existing applications or for applications that use a code-based approach for data management. In such a case the model restates the information already captured in the application and thus does not solve Problem 1. The MDD approach could suffer from Problem 3, which is mostly evident when the UI data presentation produces multiple variants of the UIs for the same data. In case there are concerns that are dependent and their combination produce an exponential growth of data descriptions, it might require a large number of generated applications states that might be hypothetical. Thus there exists a space to address Problems 3 and 7. The MDD could possibly address cross-cutting concerns, although there is no well-accepted mechanism to do that, which leaves space for Problem 6. At the same time MDD approach does not address Problem 8. Most importantly a large community of code-based developers and architects might be skeptical of MDD maintainability in production [A.3, A.4] and might solely base their development on code. Alternatively, they use a MDD approach to build application skeletons, but later changes are applied to code rather than to model, which disables later model-to-code transformations, since all code changes would be erased. Thus we need to find a way to use the MDD benefits at the code-base level (Problem 5).

Although we start with MDD that addresses several problems from Section 1.1, we believe that model can be expressed in code. Next, we apply MDD ideas into GPL. The aim of Chapter 6 is to enable the above MDD advantages in code-based applications, while avoiding maintenance of an external model. If we are able to capture in the code the equivalent information as we did in MDD in the model, then similar mechanism can be used to derive UI descriptions that represent data. This is also described in [A.8, A.14, A.15, A.16] that suggests using metaprogramming for the derivation, which addresses the Problem 5.

At the same time, we want to address other, thus far unaddressed problems from Section 1.1. From the above text, we note that the compile-time approaches suffer from multiple issues introduced in Problems 3 and 4. Thus the UI derivation at runtime is researched considering the application context. The runtime derivation brings many advantages, but also may impact performance. For example, it allows deriving the UI and considering contextual information in the derivation process to influence the

result. On the other hand, with this we start to face the issue of cross-cutting concerns (Problem 6) and tangled code (Problem 3).

Thus the critical problem to solve is the separation of concerns. What are our options? We can design a custom, non-standard solution at the MDD level, or consider the use of GP or AOP described in Chapter 3. When considering research in GP, we may find out that the amount of work is limited in comparison with AOP. Furthermore, the scope of GP is rather large, and at the same time, existing work mostly targets compile-time process of transformation. Thus we consider the AOP in the next research [A.4, A.17] of Chapter 6.

The MDD transformation process usually involves transformation rules and templates specifying the target language. When deriving the model from code, its consequent transformation process could stay the same, although we aim to support concern separation and consider runtime application context, which leads to a runtime process. Besides the application context the process must be aware of data structure information. The transformation rules must extend likewise, and this is the location that we extend with AOP mechanisms [A.4, A.17]. The data structure information and the application context act as join points, the transformation rules query them through pointcuts providing an appropriate template as the advice. This step makes the transformation rules generic and reusable across different data and changing context. Novel concern can define new join points that may occur in the pointcut, and this easily integrate to the rules. Although the expressiveness of such AOP-based rules that are resolved at runtime is wide, we might need to not only change the resolution of transformation rules but also the resolution of template content. Thus the templates are extended correspondingly. Each template uses the target UI description markup/language as it would do in MDD, but in addition to this, it also considers conditional extensions of other concerns. Thus templates also use the pointcut mechanism introduced for the transformation rules. In case a pointcut activates, then the advice embeds to the template an additional concern, an additional description extension or information referencing the join points (contextual information, data field constraint, validation, property, processed field name, data type, etc.). Templates also apply for the layout integration. Its processing and resolution can use similar AOP mechanisms.

The result of the AOP-based extension [A.4, A.17] supports separation of concerns, easy concerns integration (Problem 3), reduces tangling (Problem 6), brings concern specification centralization (Problem 4), concern reuse, as well as the above mentioned

MDD-based benefits that altogether reduce development and maintenance efforts. The runtime processing opens the approach for support of CaUI (Problem 7), although a wide range of advantages is provided also for applications with context-insensitive UIs. On the other hand, the performance related to the runtime processing might deteriorate. [A.4] evaluates the performance impact of this approach on enterprise web system by sharing experiences from a case study. Although many design advantages are brought as well as considerable code and efforts reduction, the performance impact is minimal in the domain of Java EE. Details on this research and approach that we call Rich Entity Aspect/Audit Design (READ) are provided in Chapter 6 and additional explanation can also be found in Chapter 8.

When we consider the research area of CaUI from Chapter 3, we may find out that there exist a large number of CaUI approaches and even libraries that provide remarkable, advanced and sometimes even stunning features. Unfortunately, when we try to apply them together with code-based application backends or legacy systems, they are prone to restatements. Could READ ideas address the issues related to restatement, development and maintenance of such approaches? Chapter 7 discovers what we need to do in order do adapt such a restatement-prone libraries to code-based application backends, while avoiding information restatements.

Existing User Interface Protocol (UIP) third-party CaUI approach [A.3, A.21] streams CaUI descriptions to clients on various platforms. Each client constructs the UI using native platform components to support usability. Among its features are automated UI layout adjustment of data elements for various users and context, based on given usability metrics, multi-platform/native component support, simplicity to integrate UIP into existing systems, etc. The UIP itself requires developer to have a deep knowledge of a new DSL to describe a particular UI. The DSL has weak type safety, which is a source for errors. Since it tends to restatements, the approach extends maintenance efforts and has to deal with consistency errors.

Integration with READ may reduce the restatements when using existing application backends. Since READ centralizes concerns, it is fairly easy to specify the expected transformation output, which follows the expected UIP DSL to feed the library with a UIP description reusing the existing application information. This reduces the development and maintenance efforts, and makes it easy for the UIP to transit to other code-based systems. Later changes to data definitions in code-based systems directly promote to UI, since READ constructs new UIP description. A provided case-study

[A.3, A.21] shows the simplicity of READ integration with third-party libraries, which provides developers with the READ advantages.

The Problem 8 is so far unaddressed. Even though we use concern-separating approach, the separation drops when it goes to rendering. Consider web-based applications that use the concern separation for UI description. When a client requests a UI page, the server-side resolves the UI from separate concerns, tangles them together and produces a tangled client-readable description that is delivered to the client. The server must spend efforts to weave the data presentation concerns together. The volume of the client-readable description grows as it may contain repetitions, although compression can reduce it. It is apparent that the delivery process does not maintain the concern separation. The client-side then works with the tangled description that disallows to reuse a particular concern, as it is not separable from the tangled description. This reduces options for client-side caching; thus a small change involving a single UI concern requires re-transmission of the entangled description.

Is it possible to apply similar ideas of concern separation to UI delivery? The answers brings Chapter 8. [A.1, A.6, A.12] shows that such separation is possible, which brings multiple positive impacts. The concern delivery becomes distributed, and the client becomes responsible for the UI assembly, which gives the client-side higher responsibility and extended capabilities, such as making decision on concern reuse and caching; furthermore the server-side efforts for the UI assembly reduces. The content provided by the server-side reduces; furthermore extended client-side caching options push towards even greater reduction that results with improved UI page load times and responsiveness.

Chapter 8 describes an extension to the AOP-based UI design approach [A.1, A.6, A.12] that allows providing various UI concerns separately to clients, which partially delegates the data presenting UI component assembly to clients. Clients can request various concerns simultaneously from the server-side, which further impact the page load times. Chapter 8 provides a case study showing the impact on page load, caching, and the server-side. This addresses Problem 8. Furthermore, the distribution of UI concerns provided in a machine-readable way opens the possibility to reuse provided information by native clients and integrate native platform components to present given data [A.7, A.22], which addresses Problem 9.

Ongoing, preliminary research [A.10, A.23] focuses on system business rule inspection and its transformation to the UI. This extends the approach that derives data-driven UIs

with more contextual information and allows us to consider interaction with business flow. This direction of research of business rule inspection and transformation is left for future work, and mentioned with other directions of research in Chapter 9 on future work.

5

Extension to UML Models to Support UI Derivation

A journey of a thousand miles must begin with a single step.

-LAO-TZU

Tao Te Ching

This section presents the research that applies MDD to derive UI data presentations. This section bases on research published by [A.2, A.11, A.13]. First, a basic derivation from UML class diagram is sketched. Such derivation can produce basic UI fragments. In order to improve the derivable results, an extension to the UML class diagram is suggested to capture input validation, presentation, and security concerns. Such an extension allows designers to derive advanced UI fragments, such as rich forms, tables, and their variations. This chapter explores the benefits provided by MDD, such as reductions or restated information and repeated decisions. The advantage is also that such approach brings concepts that are platform-independent. Outcome of thus chapter is considered in subsequent chapters.

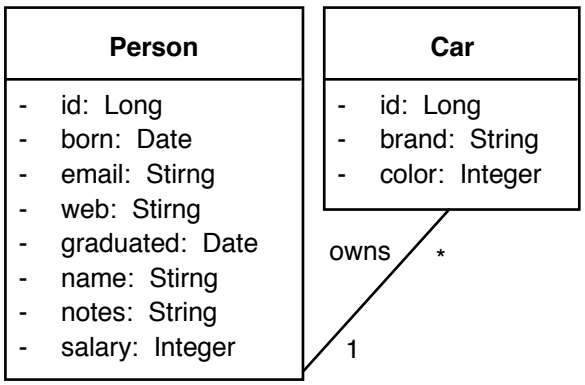

Figure 5.1: A UML class model example

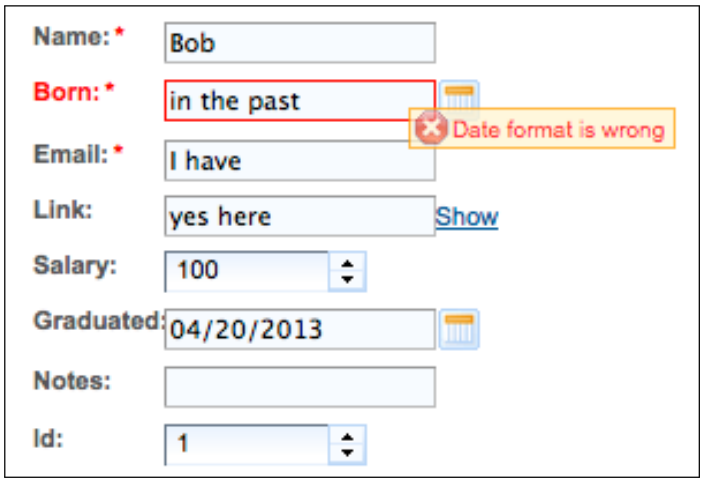

Figure 5.2: Generated *Person* form (model in Fig. 5.1)

5.1 Problem Description

Despite many advantages of MDD, model-based user interface development tools (e.g., [51, 53]) have not been widely used in practice [54]. In many cases, developers create UIs manually. They are usually very familiar with the current implementation. This way of UI development can work for small-scale applications initially, but is not desirable even for a small system development when the system should evolve. The problem with manual UI creation becomes worse when the code fragments have been generated using a model-driven tool. For example, manually added code makes maintenance difficult because MDD aims to have all changes in generated fragments propagated from future changes in design. Therefore, there has been a need to integrate the required UI code fragment information within design models. Many UML tools that are currently available, however, do not support the properties necessary for the UI code fragment generation in the underlying models.

For example, consider UML design model shown in Fig. 5.1 from which we want to derive a view form for a Java EE web application. Each field should be validated when it is collected through the form. A view form like one given in Fig. 5.2 can be generated for the *Person* class using the information currently given in Fig. 5.1. However, such a form has limited capability in detecting the meaning of fields and in User Input Validation (UIV), because the given design model does not provide any form generation rules with validation details other than field types. Another problems in the form can be summarized as follows. We might want to have fields in a different *order*, restrict that the provided birth date be in the *past*, an input to the *email* address field could be invalid, an *http* link can be malformed, there can be a negative value given to

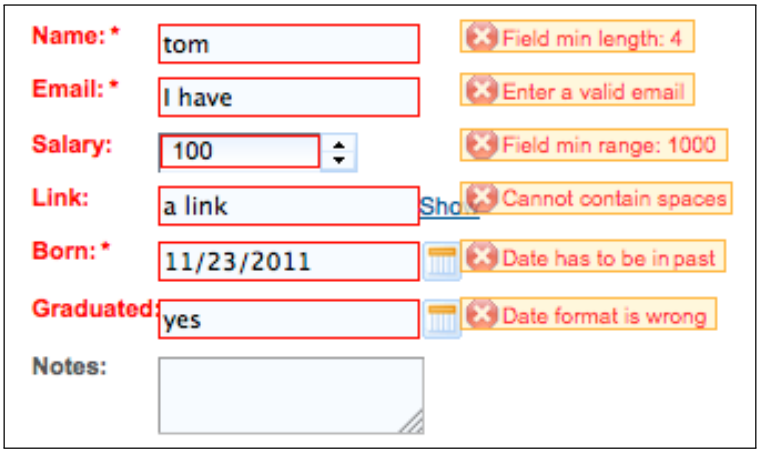

Figure 5.3: Expected *Person* form (full access with validation)

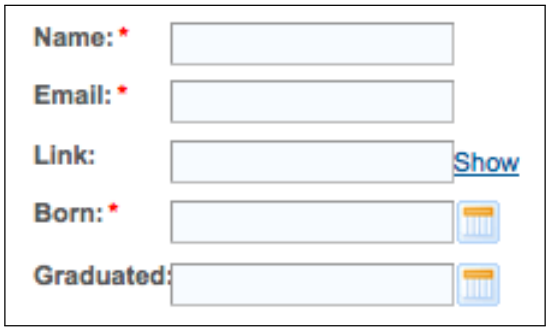

Figure 5.4: Expected *Person* form (restricted access)

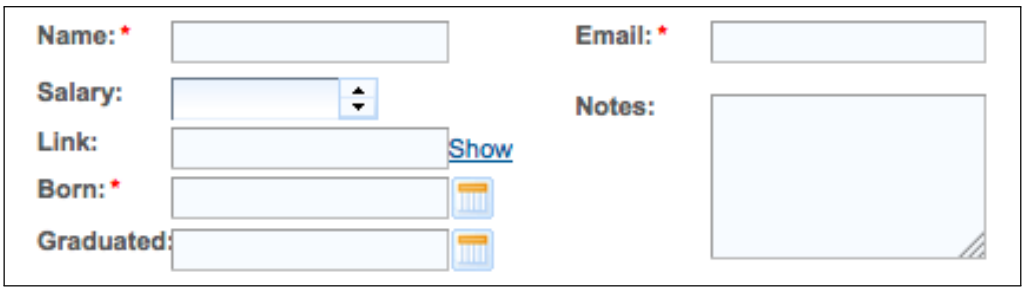

Figure 5.5: Expected *Person* form layout for a specific context

the salary field, etc. In addition, the current form includes the field *id* that should be assigned by the system, not by the user. Consequently this is not the view form that was expected. Therefore, we often generate an initial code for the form and then add additional constraints and validation rules to each field in the form and reorder the fields. Fig. 5.3 shows a form, even including required UIVs, that can be generated by this additional modification.

Another important requirement that is common in many applications these days is that it should be able to support multiple users who may have different access rights or user roles in the system [76]. This may also impact the forms that are provided by the system. A user with full access rights may see the form in Fig. 5.3 for a person selection, but a user with restricted rights would be allowed to see just a subset of this form, such as the one in Fig. 5.4. Having both forms supported within the system may require either the duplication of the same form for different users or embedding extra access control checking conditions to each field in the form. The latter approach with embedded conditions may work for the forms that basically share the same layout (like Fig. 5.4). However, it will not help when we need various form layouts to support more user context-specific needs (as shown in Fig. 5.5).

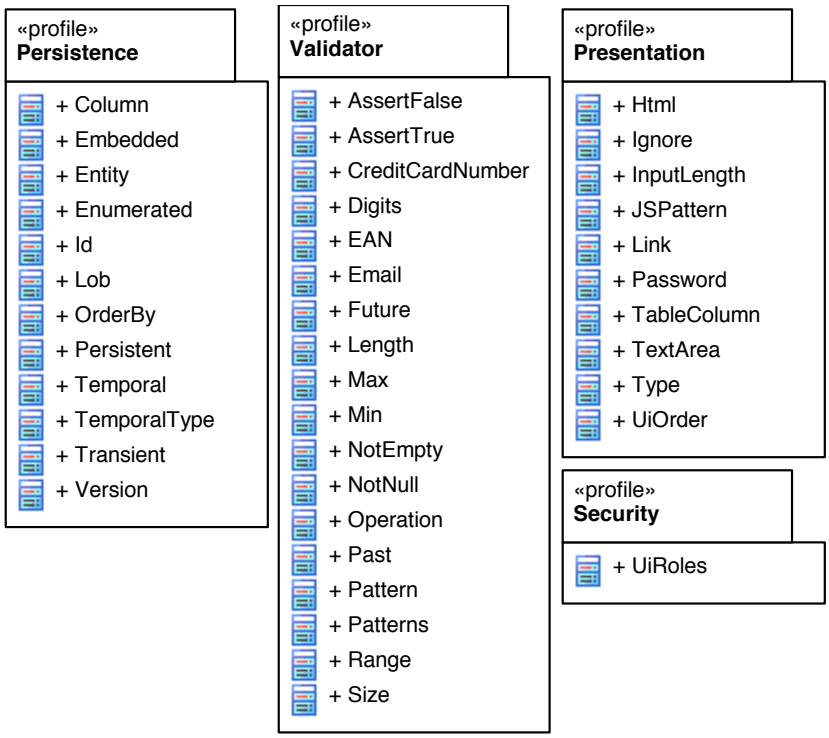

Figure 5.6: Example UML profiles supporting ORM, UIV, Presentation and Access rules

One of main problems in this kind of form generation process is that further changes made later on the design model cannot be automatically applied to the form that was originally generated. Therefore, one needs to modify all related view forms and their underlying entities, including the database. If we decide to change restrictions for user rights, we must again manually modify the original forms, which is an error-prone and tedious work. In the next sections, we show an approach that makes it possible to generate complex UI fragments and provide a tool that is capable of such generation. The main focus is put to fragments representing data. In addition to form and table generation, we consider UIV and the impact of access control [76] applied on fields.

5.2 Solution

Section 2.2 introduces approaches used in industry to support ORM or UIV for OOP using class and field annotations [9, 10]. An approach to persistence modeling [77] promotes JPA by an UML profile to UML models. This then allows us to use the model for MDD, which is the same way we can promote other concerns to models.

5.2.1 UML Profiles

To extend UML class models to capture ORM, UIV, UI presentation and access rights, four UML profiles are defined [A.2, A.11, A.13]. From the ORM metamodel defined

Table 5.1: Subset of UML profile stereotype details

Stereotype (Annotation)	Description	Appl. type
MD-ORM profile		
Column(..)	DB table column with props.	Any
Entity	Enable JPA for POJO class	Class
Enumerated	Persist ordinal value or string	Enum
Temporal	Date, Time, TimeStamp	Date
	..	
MD-Validation profile		
Length, Min, Max	Value length in the range	String
Email	Match email	String
Pattern	Value matches the reg-exp	String
Future, Past	Future/Past date	Date
NotNull,NotEmpty	Not null (empty) attrib. value	Any
	..	
MD-Presentation profile		
Link	Web link expected	String
TextArea	Long text expected	String
Html	Html expected	String
Password	Secret text expected	String
Type	Type of widget to use	Any
UiOrder	Order in view form	Any
TableColumn	Use field for table generation	Any
Ignore	Do not use field for generation	Any
InputLength	Input widget length	Any
JSPattern	JavaScript regular expr	String
	..	
MD-AccessRules		
UiRoles	Restrict access to roles	Any

by JPA standard [16], all rules can be transformed to UML stereotypes and tagged values to define the *MD-Persistence profile* [77]. An existing meta-data model for JavaBean validation (JSR 303) [16] is applied to define the *MD-Validation profile*. A *MD-Presentation profile* then addresses the missing elements to fully determine the form content and follows the metamodel defined in [A.14]. Furthermore *MD-Security profile* restricts access to given fields. Table 5.1 provides a subset of stereotypes from those profiles as well as their description and applicability to domain types. Fig. 5.6 shows an overview of profile implementations. These profiles describe the system structure fully from all four aspects and allow us to fully determine view form content.

In Section 5.1, we have shown that a form may be used in various mutations for users with different access privileges and context. To support such access control, we apply the Role-Based Access Control (RBAC) security model [76]. For the form generation, all required permission checks are applied to each form field. A field is rendered when the user has a role granting access to the given field. In our approach, we allow a user to have multiple roles in a given session. The user is provided with a form containing a particular field, if he has at least one role that allows access to the field. To deal with different fields rendered in the form for different users, we may apply multiple

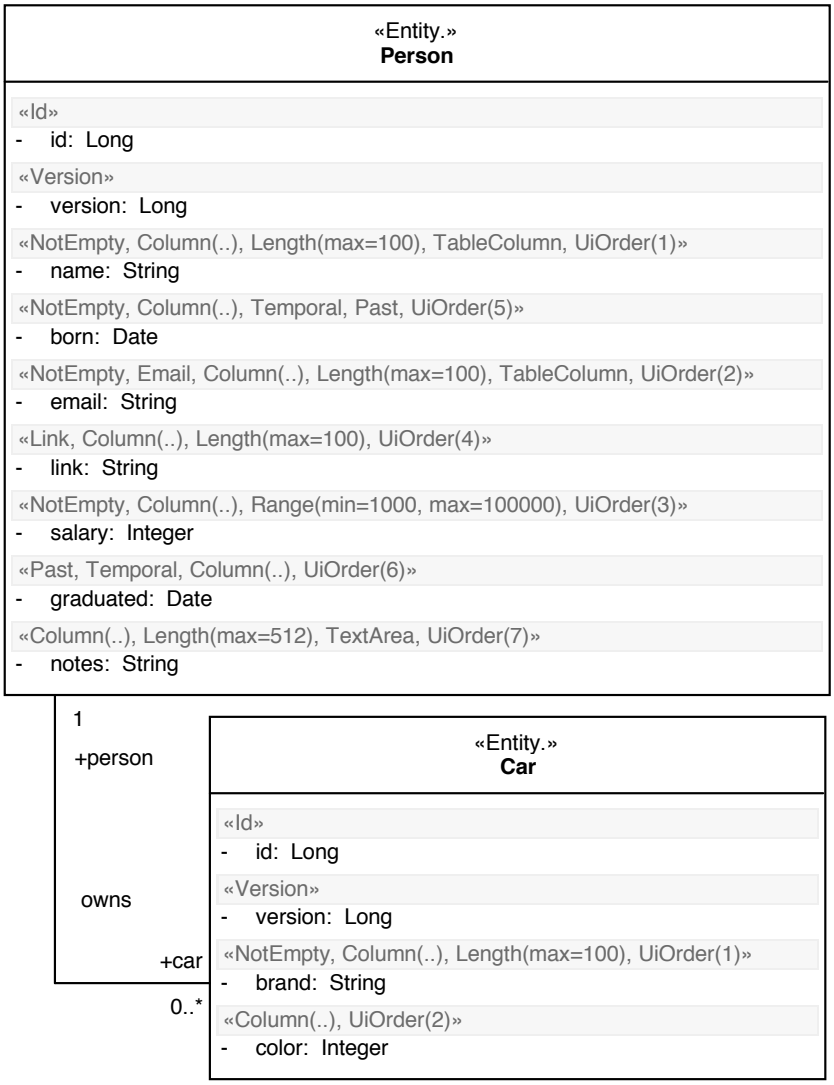

Figure 5.7: Example of rich design model

approaches. One possible approach is to design forms where each field can be selectively rendered based on supplied conditions that involve additional presentation logic. To avoid this addition in the presentation layer, we suggest that the design model captures these field access restrictions. This is supported through the *MD-Security profile* with a stereotype *UiRoles* with a tagged value specifying roles that has access to the field. Until now it was possible to generate one form that suited all the requirements, but for the field access control with logic kept in the model, we must either generate multiple forms for different contexts of their use or generate them at runtime as the user rights are evaluated. These strategies are discussed and evaluated later. Layout concerns can be thought as a decoration for defined templates that directs the transformation of entity fields to UI fragments. A specific layout can be selected based on given context or rules given by designer.

In order to verify models, UML profiles specify a set of constraints using OCL [77]. This language defines invariants that can be applied to stereotypes and verified in an integrated development environment that supports an OCL interpreter. Listing 5.1 shows selected OCL constraints.

```
context Length inv:
  self.max >= self.min and self.min >= 0 and self.type.name = 'String'
   and self.max = self.property.Column.length
context Size inv:
  self.max >= self.min and self.min >= 0
   and self.type.oclIsKindOf(CollectionType)
context NotNull  inv:
  self.property.Column.nullable = false
context Password inv:
  self.type.name = 'String'
```

Listing 5.1: Selected OCL constraints UML profiles

5.2.2 Model-Driven Fragment Generation Example

With UML profiles, complex UI fragments can be generated. The class model from our example in Fig. 5.1 can be extended using the defined profiles. Fig. 5.7 shows the extended design model. The *Person* entity generated from the model contains all information as annotations, which matches the manual development approach. The *Person* entity in Listing 5.2 is equivalent to the one from Fig. 5.1 with the addition of the profile stereotypes.

```
@Entity @Table(name = "Person")
public class Person implements Serializable {
  private String name;
  private Date born;
  ..
  @Column(name="name",nullable=false,length=100)
  @NotEmpty @Length(max = 100) @UiOrder(1) @FormTableColumn
  public String getName() { return this.name; }

  @Column(name = "born", nullable = false)
  @Temporal(TemporalType.DATE) @NotEmpty @Past @UiOrder(5)
  public Date getBorn() { return this.born; }
  ..  /* get/set */
```

Listing 5.2: Person entity fragment

We implement a tool that generates UI fragments for provided entities. It inspects entity attributes and their properties, and based on that, it selects an appropriate UI widget. It propagates all constraints and validation settings to the widget so that UIV can be applied. The designer can define mapping between the widget and entity field properties and also design his own library of widgets to allow full customization. A generated form example for *Person* entity integrating UIV and constraint restrictions is shown in Listing 5.3.

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

Listing 5.3: Person view form code

Our tool is based on the attribute inspection and supplies all the underlined texts. The design model in Fig. 5.7 captures all the information required to generate the targeted form in Fig. 5.3. The *Car* form in Fig. 5.8 is generated in the same way, providing an example of a more complex attribute *Person*.

UI fragments such as tables or reports can be generated as well. Tables are in fact not much different from forms in terms of their source code, even though they capture multiple elements. On the other hand, we often want to provide a table with fewer entity fields than what a form provides. In order to mark entity fields used for table generation, MD-Presentation profile uses the *TableColumn* stereotype. The generated person table code in Listing 5.4 is rendered as the table in Fig. 5.9.

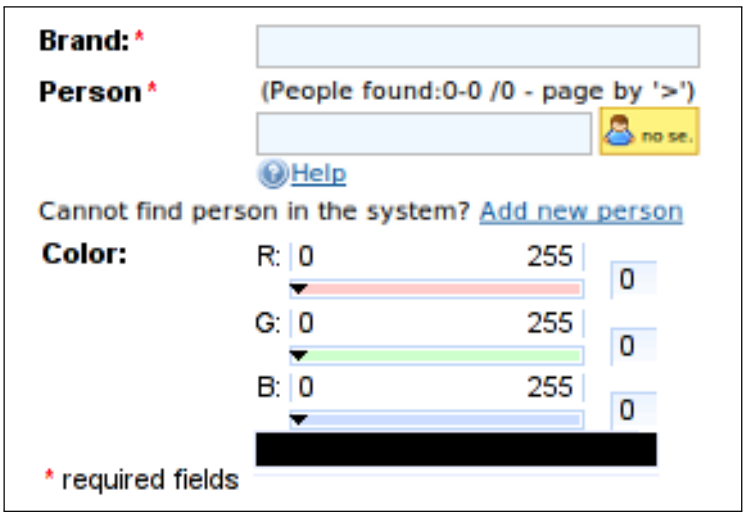

Figure 5.8: Generated car view form

Name ▲	Email ⇕
Burnes Tom	tom@burnes.com
Doneck Bill	bill@doneck.com
Nowak Bob	bob@nowak.com
Smith John	john@smith.com

Figure 5.9: Generated person table

```
<h:form id="formPerson">
  <rich:dataTable id="tablePerson"
     var="element" value="#{collection}">
     <rich:column id="name" label="Name" sortBy="#{element.name}">
         #{element.name}
     </rich:column>
     <rich:column id="email" label="Email" sortBy="#{element.email}">
         #{element.email}
     </rich:column>
  </rich:dataTable>
</h:form>
```

Listing 5.4: Person table code

5.2.3 Form Field Access Control

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

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

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

Runtime form generation. A form is generated upon a request, and user roles are taken into consideration

The first *One Static Form Per Entity* strategy uses only one form that contains all the fields and selectively disables fields restricted for a given role(s) and context. In this case, all the logic determining which fields to use is added to the presentation layer. On the other hand, the usage is simple and can be utilized by systems with a smaller variety in restricted fields per user roles. This strategy can handle role hierarchy and the use of multiple roles per user. The disadvantage of this strategy is that all decisions are repeated per form request, which may impact the performance. Second of all, the maintenance of the security is more difficult than in the alternative options. An example code snippet is available in Listing 5.5.

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

Listing 5.5: Presentation layer logic in a form (Seam)

The second *Multiple Static Forms Per Entity* strategy is related to the UML profiles, which allows us to push the security decisions down to the class model. The MD-Security profile allows specifying *UiRoles*, which are the roles with access to the field. Multiple forms are generated per entity for specified roles. This results in multiple forms in the system for the same entity with different fields available. These forms are then available in a folder specific for each role(s). The proper entity form is selected from a location influenced by active user roles at the request time. For a system where a user has multiple roles and only a single role can be activated in given context, the physical form fragment location is determined by this role (see Listing 5.7). In this case, the system contains $roles \times entity$ forms. The difficulty comes when we consider a system with multiple roles activated at the same time. In this case we apply role *union*. For the multiple role activation scheme, up to 2^{roles} variations of forms per an entity exists.

This second strategy at first pushes the security decisions to the domain model (see Listing 5.6). Second, it generates all the forms before they are used. Third, it provides a form for the combination of an entity and system roles. This, on one hand, speeds up the form load as no field related access decisions are made at runtime. On the other hand, this may result in too many forms per entity. An example code snippet for the entity is available in Listing 5.6 with the use in view shown in Listing 5.7.

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

Listing 5.6: Person entity field access control

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

Listing 5.7: Person static form selection (Seam)

A similar approach is applied to Java EE where an annotation *@Restrict* controls the access to a class, class method, or an attribute. In this case, the restriction is more complex; we may apply restriction based on user role or a context, such as equality of an active class fields, context variables, etc.

The third *On-demand Form Generation in Run-time* strategy is not to pre-generate forms, but create them on demand at runtime. No physical location for such a form exists. Instead, the entity class is inspected, and the form is generated every time it is requested. The advantage over the previous two cases is that we do not need any disk space available for multiple role-forms and the forms do not need to be located and fetched. On the other hand, inspecting an entity and generating a form require us to open multiple files from a UI widget library in order to generate the form. File accesses could be reduced in this case if all widgets are held in memory. In this strategy, we see that the performance bottleneck is the per-demand form generation and a shared cache application is appropriate. In fact if we use a cache, then this strategy does not differ from the previous static forms but provides all above mentioned advantages. The advantage is that only form-role combinations used by the system are taking space in memory. An example coding is in Listing 5.6 with the use in view in Listing 5.8.

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

Listing 5.8: Person run-time form generation (Seam)

Table 5.2: RBAC Performance evaluation – plain page

Role	Approach				Page size	Resources
	(*a*)One static form [ms]	(*b*)Mult. static forms [ms]	(*c*) On-demand [ms]	(*d*) On-demand (cache) [ms]	[B]	[-]
Admin	296	277	313	283	1157	1
User	245	181	196	172	1037	1
Guest	244	94	100	95	947	1

5.2.4 Performance Evaluation

In order to evaluate performance of form field access control strategies, we build a small application in the Seam Application framework Java EE [20]. We design an entity *Test* with 10 *String* fields and apply 3 user roles: Administrator with access to 10 fields, User with access to 5 fields, and a Guest with access to 1 field.

The *Test* form is embedded in a web page that is accessible under a selected role. We evaluate the page load times. In our first experiment we use a plain web page with the form that uses presentation logic (*a*) to restrict field access. Next we apply forms statically-generated (*b*) for all system roles and their combinations. Afterwards we employ forms generated on-demand (*c*). Lastly we use optimization for the on-demand generation with caching (*d*).

For the measurement, we use the Firefox 3.5.9 web browser with disabled cache using a web debugging proxy. Each measurement consists of 10 samples. Measurement results are presented in Table 5.2. In Table 5.2, the columns (*a*) - (*d*) represent strategies discussed earlier. The result shows that (*a*) is sufficient for applications where the full forms are applied and a small variability for field restrictions applies. As the restricted field count grows, other approaches seem to outperform (*a*). One surprising result can be that (*c*) is slightly slower compared to (*b*), although it provides the poorest performance for the full form. Optimization to (*c*) which we denote as (*d*) should provide asymptotically similar results to (*b*).

The above evaluation tests a plain web page with a form; this might provide results that do not reflect a real scenario where a web page contains multiple components and resources, which impact the page size and overall load time. In order to provide broader evaluation, we apply our approach to a custom web page deduced from ACM-ICPC registration site, where we use a web page that contains additional components and measure how our form field restriction approach influences the load time. In this

Table 5.3: RBAC Performance eval. – enterprise env.

Role	Approach				Page size	Resources
	(*a*)One static form [ms]	(*b*)Mult. static forms [ms]	(*c*) On-demand [ms]	(*d*) On-demand (cache) [ms]	[kB]	[-]
Admin	1578	1447	1540	1491	296.0	17
User	1544	1346	1425	1421	295.8	17
Guest	1384	1335	1341	1347	295.7	17

measurement, we again apply the *Test* entity. Table 5.3 provides the measurement, where the page size notably increased requesting 17 resources (file) from the server. The second measurement amortizes the overall load time because of the additional components of the page. The comparison results in this measurement show that (*a*) needs the longest time to load the page, where (*b*) provides the best timing, (*c*) and (*d*) provide performance very close to (*b*).

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

5.2.5 Development and Maintenance Evaluation

To draw the impact of such approach in [A.2], we build a small Java EE application with a domain model that consists of 5 classes in Fig. 5.10. In the study, we consider physical LOC while stripping white spaces and comments, and then we apply Constructive Cost Model (COCOMO) II method to evaluate costs placed on maintenance scenarios [78]. The study compares manually-developed application and one with the model-based approach. The manual application has less code for the domain model, but the UI part becomes bloated, and our approach reduces the UI code by up to 24%. The manual application has 1023/3853 lines of Java/DSL, while the MDD one reduces to 1092/2918 lines.

The maintenance of UI forms/tables becomes easy with our approach; new data field for administrator role would normally require 10 lines in Java and 33 lines in DSL, so the reduction gives only 12 lines in Java. A novel data class with basic UI for management would need 141/659 Java/DSL lines while MDD reduction only needs 143/327. A field constraint change reduces from 1/5 to 1/0 Java/DSL lines and field name change goes from 1/17 to 1/0 lines.

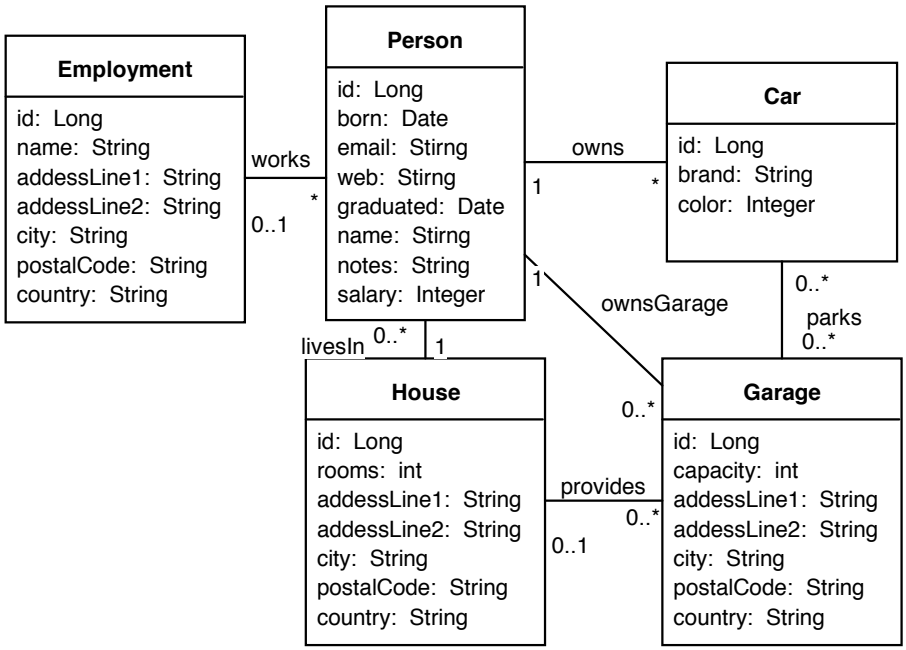

Figure 5.10: Example for case study

Table 5.4: Cost estimation of considered applications and scenarios

Scenario	MDD approach	Common approach
Application development	$2'144	$2'861
New class field for *admin*	$5	$25
Field constraint change	$1	$3
Field name change	$1	$10
New class (Employment)	$240	$357

When we apply COCOMO II to the statistics and consider an expert team with good knowledge with high tool usage and person-month cost $1500, we derive costs of particular maintenance scenarios as shown in Table 5.4. Thus with our approach, we can gain for both development and maintenance.

5.3 Summary

This chapter has shown a possible model extension that allows designers to generate UI forms/tables directly from models. Basic UML class diagram is a good starting point for UI generation, but it is clear that with no extra information, the model can only produce a one-time UI skeleton. The main impact of this section is model extensions; the same extension can be applied to OOP domain model or any alternative model, because these extensions are needed for input validation, data integrity, presentation and security.

This approach brought discussion on reduced maintenance efforts and elimination of the manual UI form development through the use of MDD. The limitations can be seen with integration of alternative presentation elements, support for adaptive UIs, or addressing cross-cutting concerns. Although, this chapter shows that model extensions can be reused for the UI and thus reduce restated information from model in the UI, it did not consider transformation details, genericity of transformation rules or applicability of third party widgets, etc. On the other hand, the provided study makes evident that the aim for runtime generation and its performance expectations. The outcome of this chapter is an initial and partial step towards a more versatile solution that involves AOP, but the data class extension profiles will play a significant role in it for all the code-inspection, transformation rules and also for the widget templates used by the transformation.

6

Rich Entity Aspect/Audit Design (READ)

Simplicity as a result of a creative process is "the ultimate sophistication".

-Leonardo da Vinci

This chapter provides details on an evolution in our approach. It considers advantages brought by MDD approach introduced in Chapter 5 and extends them with the ability to capture model in code. Next, it considers separation of concerns and applies advantages brought by AOP. The AOP-based development in conjunction with a metaprogramming approach, model-based transformations, and ideas of generative programming brings considerable advantages to development of UI data presentations. It not only reduces the development and maintenance efforts, but it open existing systems with support for context-aware UIs. We believe that for the same amount of investments this approach can help to build systems with much wider options and abilities.

This chapter presents work published at [A.4, A.8, A.14, A.15, A.16, A.17]. The chapter shows integration of MDD advantages to code-based applications and thus avoiding the necessity of design and information restatements related to external models. The runtime process considers application context in its reasoning for UI derivations which fits to the area of CaUIs, even though the conventional basic UI design benefits due to increased reuse of concerns and thus reduced development and maintenance efforts. The mechanisms suggested in this chapter are generic and thus particular concerns can be reused across different projects. A case study demonstrating this approach is provided. It involves a production-level enterprise application, with high demands on performance.

6.1 Motivation

Despite broad research in the area of CaUIs [A.4], it is common practice in production applications to design a single UI that serves all types of users and contexts [79]. 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. As demonstrated previously, a large amount of time is devoted to UI development. 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 domain model and again as a reference in the presentation that is often specified through a DSL [27] with weak type safety. In addition, current practices realize multiple UI concerns [5] 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 [4]. 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 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.

6.2 Problem Description and Analysis

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, a process called functional decomposition in OOD [4] or more generally in GPLs. 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, GP [43] and AOP [4] propose another way of thinking about program structure. GP proposes the use of a GPL language together with problem descriptions in the form of DSL [27]. 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 [13].

As described in Section 3.1.1, in order to design a UI fragment representing data, we must consider its fields and for each field select an appropriate presentation component. The selection is based on field type and constraint inspection. Next, we need to bind the data field to the component and provide additional UI settings from field's constraints. We may also set a conditional component rendering. Furthermore, we might need to tangle the result with a layout. The result is represented in a component tree. Having such UI data representation, we want to handle a situation where multiple concerns change individually for given system conditions. For instance, we want to handle situation where the layout changes based on the context, or the field presentation differ for mobile and desktop clients, validation rules apply based on user access rights and context, given fields render based on users access rights, etc. In such cases, we could either use conditionals in the UI or, even worse, copy the UI fragment and modify a given concern. We should assume that each of these concerns can have its own variability dimension, but in conventional approaches, these are mixed together as mentioned in Section 3.1.1.

In [A.17] we present a multidimensional concern space as shown in Fig. 6.1. This demonstrates a form with coupling to five distinct concerns. Fig. 6.2 demonstrates that each concern defines an individual dimension. The way we combine them in conventional code collapses all these concerns to only single dimension, and thus we loose the ability to deal with these concerns separately. Certainly, it is possible to

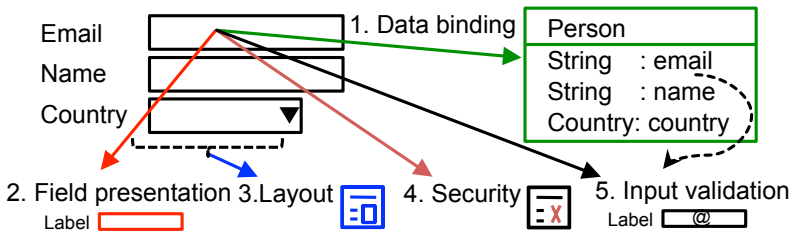

Figure 6.1: UI form decomposition

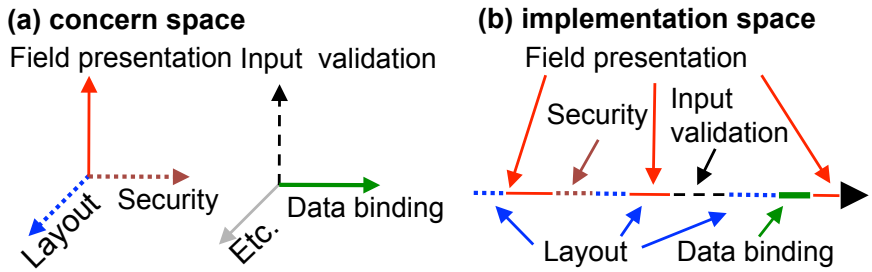

Figure 6.2: (a) Concern / (b) Implementation space

consider that multiple of these concerns vary separately in a given system based on contextual conditions, and since they are captured together, we end up with n copies of the original code fragment representing the data. The worse case scenario for n is derived by Eq. 6.1. In such a case, even a trivial change of data field constraint impacts a large number of UI fragments. Furthermore, the impact is hard to locate in the UI when we consider that the UI is based on weak type safety environment.

$$n = |concern_1| \times |concern_2| \times |concern_3| \ldots \times |concern_m| \tag{6.1}$$

For more details, consider an example that aims to design the Person form given in Fig. 6.1. 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 with its properties are chosen based on the type of a particular field and its constraints. Any time 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

```
<table>
 <tr>
  <td>Email:</td>
  <td><h:input id="email" value="#{person.email}" required="true" maxlength="50"
        render="#{security.hasAccess('email')}" validate="#{v.validate('email')}"/>
  </td>
 </tr><tr>
  <td>Name:</td>
  <td><h:input id="name" value="#{person.name}"  maxlength="50" required="true"/>
  </td>
 </tr><tr>
  <td>Country:</td>
  <td><a:smenu id="country" value="#{person.country}" required="true"/>
  </td>
 </tr>
</table>
```

Listing 6.1: Sample source code for UI form reflecting Fig. 6.2 (b)

them together or presenting them within a given screen size. Designing a non-trivial form layout often results in layout code entangled together with form fields. We provide an example of tangling such concerns in Listing 6.1. 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 the layout in Listing 6.1 for a user with a 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 while other concerns are unchanged, but having all the concerns at the same place limits the reuse, often resulting in 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 input validation. For instance, web applications with client-side validation must restate constraints in a scripting language, such as JavaScript.

Listing 6.1 shows a very simplified implementation of Fig. 6.1; this JSF code shows data binding to the form through a data instance *person* (*value* attribute in widgets), field presentations through UI components (*h:input/a:smenu*), table layout tangled through the fields, security condition (*render* attribute), and validation (*validate, maxlength, required* attributes, etc.). The maintenance of such fragments becomes difficult because all five concerns are captured together. Although Listing 6.1 differentiates concerns by colors, in reality 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 limited since it only allows slight variations of concerns in the code. CaUI design only compounds the problem since it typically increases the number of concerns.

As shown in Fig. 6.2, the AOP approach [13] sees the problem in an n-dimensional *concern space* that is expressed in the *implementation space* using a one-dimensional language, a GPL language. The 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 Fig. 6.2 (a). This concern space is mapped into one-dimensional implementation space in Fig. 6.2 (b). This corresponds to what we see in the one-dimensional implementation in Listing 6.1.

The example above presents only a basic use case; next, let us consider more advanced expectations from an effective UI design. In order to design a CaUI effective from the development perspective, we must consider multiple quality attributes. It should allow designers to capture the expected functionality but also other non-functional attributes. First, it provides multiple presentations, different layouts, easy to address and integrate various concerns, adaptivity to application runtime context, third-party integration (security), etc. Second, it should be easy to develop and maintain individual concerns with low coding effort, while preserving development approaches already known to the developer in GPL or CBD. 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 [5] into readable code fragments to support their reuse and maintenance.

When we consider conventional approaches and look back to Fig. 6.1, we should note that multiple other concerns may exist for CaUI, 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 Fig. 6.2 (a) grows and the complexity represented by Fig. 6.2 (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.

6.3 READ : Rich Entity Aspect/Audit Design Framework

In order to design a CaUI 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), all described in Chapter 3.

First, we specify information that we want to reuse. These are data structural information and their constraints. All these can be found at the application domain model. Assuming that the domain 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 recognize the given data through MP and use its structural model, which 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 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 a 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 uses the same query constructs to question the data field structural model and context. An advice is different; it is a DSL content template that can reference the structural model properties or context variables. All integration rules are considered for given template; if a rule pointcut holds, then its advice content embeds to the template or resolves given reference to the structural model (such as field name, type, etc.). The result of the template interpretation is a 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. 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 to the component tree (in CBD) and UI embedding.

6.3.1 Introduction to READ Conceptual Model

As suggested in Section 2.3.5, we describe the AOP conceptual model with the three main components: the *join point model* that defines available join points, the *pointcut language* that defines the query language to select a subset of join points, and *adaptation mechanism* that allows adding / modifying functionality at selected join points. In comparison to AspectJ and Hyper/J, 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 model and 2) application runtime context (a subset exposed to the READ process). A structural model provides entity and field names, data types, and field annotations with their parameters [9, 10]. An 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 for given data instance. More specifically these join points allow us to support generic/reusable transformation rules.

AOP terminology as described in Section 2.3.5 deals with two types of join points [18], static and dynamic. While the application context corresponds to the dynamic join points, the structural model contains 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 (EL) [80]. 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 [18]. 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 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 add 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.

6.3.2 READ Lifecycle

The READ lifecycle in Fig. 6.3 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, consider the example in Listing 6.2, the context can be an indication that the aimed content is a read-only or editable presentation, fields named *"notes"* and *"password"* 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

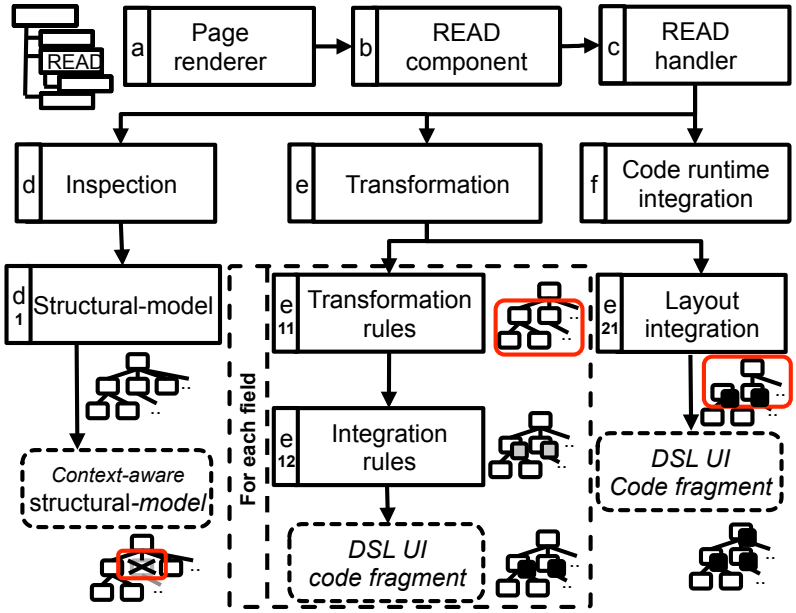

Figure 6.3: READ lifecycle

a given context using the Annotation Driver Participant Pattern (ADPP). This may result in modification of the structural model instance. 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 (e11 and e12). This results in a UI code representation in the target UI language for each field. After all fields process then the layout is integrated (e21) receiving the entire data UI representation as result in the target language. The last stage interprets the resulting UI fragment and builds a component tree, which integrates it to the UI (f).

6.3.3 READ Lifecycle Integration

Next, lets consider how is the *READ UI integrated*. 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 6.1 to describe the data. Instead a custom component is used as shown in Listing 6.2. 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 context). This component is associated with custom handler that pushes the local context to be considered in the READ context and issues the phases described earlier in Section 6.3.2 to receive the CaUI for the given data instance.

Table 6.1: 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
	..		

The *inspection phase* is rather complex; thus more details are provided. It audits classes of the domain model (entities). It specifically looks for class name, class restrictions, its fields and field constraints. While aiming to build on existing industry standards, it complies with the profiles for persistence [9] and input validation [10] introduced in Chapter 5. Both of these standards are usually applied to existing systems already, and the inspection considers them. As shown for the MDD approach [A.2] from Chapter 5, this can be further extended for role-based access control, for presentation, etc. Table 6.1 bases on Table 5.1 and shows the class structure and field elements and a subset of selected extensions applicable to domain model class fields. The table de-

```
<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"/>
```

Listing 6.2: Example use of READ UI component

```
@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 6.3: Example entity with additional markup

scribes 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 extensions.

The result of the data inspection is a structural model, a three-level composite structure, reflecting the class-level, field-level and constraints or extensions. The structural model provides these properties to other READ stages through the variable name (the right most column in Table 6.1). 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.

The *transformation phase* consists of three parts as denoted in Fig. 6.3. The result of the inspections phase, a context-aware structural-model of the data instance, is accessible to the transformation through EL context using variables described in Table 6.1. 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 6.3 (it uses annotations [9] and [10]) and context

```
<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 6.4: Example transformation rules

```
<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$Field$}"/>
```

Listing 6.5: Example template for inputText widget

specified in Listing 6.2. The structural model reflects information about the data, its fields and field properties, while the settings in Listing 6.2 modify the model instance, and in our case fields password and notes are ignored. It 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* of the transformation phase applies *transformation rules* to the model fields. An example of a subset of such rules captured in a DSL is shown in Listing 6.4. Consider the *first name* field from Listing 6.3. 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 *emailTemplate* since the second pointcut applies. The pointcut could use any variable available in EL context (consider context variables in Table 6.1 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 6.5. Note that

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

Listing 6.6: Pointcut strategies for templates: (a)-full/ (b)-brief/ (c)-shorten version of integration rules

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

Listing 6.7: Example layout template

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 6.6. 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, the EL environment is denoted by the $ mark. 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 can express anonymous fields and iterate over repeating layout pattern to populate information. Consider Listing 6.7 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 can possibly reference explicit fields that spans over two columns, etc. If fewer fields exist than specified, only the given amount applies. To hide the target DSL, which may only be applicable with a given field, a conditional (cover) fragment can be denoted. The specific fields take precedence in the layout resolution. The result is a CaUI code fragment that the READ handler compiles and integrates to the UI.

6.3.4 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 domain model by classes and applies ORM. For example, Java EE defines standards [9] for the ORM, which extends the class model with additional markup. Similarly validation [10] can be added. Generalization of such extensions and further enhancements are suggested by [A.2].

READ inspection uses all of this information for the structural domain model composition and for join points. Besides this model, READ can also integrate business rules defined in the above layer. Preliminary work in [A.10, A.23] shows that business rules can be inspected and their definitions reused. This can be integrated into the READ context.

Considering common development approaches, we only expect domain 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 parameters 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 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 domain 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 Asynchronous JavaScript and XML (AJAX) rendered views.

6.4 Evaluation

This section provides an evaluation of READ 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 EA for the worldwide programming competitions, the ACM-ICPC registration system[1]

[1]available at http://icpc.baylor.edu (2015)

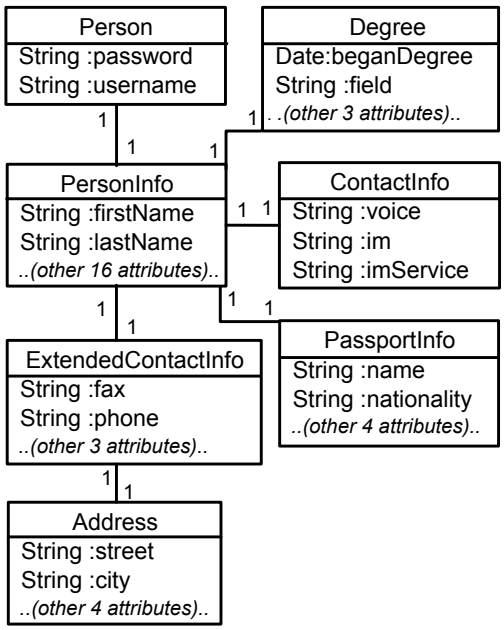

Figure 6.4: Evaluated application domain model

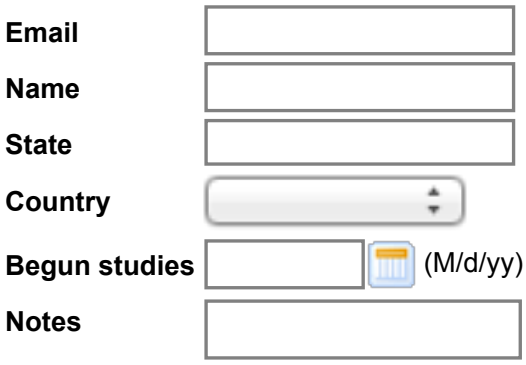

Figure 6.5: Sample simple UI Form

6.4.1 Development and Maintenance Impact

In this section, we consider a subsystem of an existing ACM-ICPC system used for registration of users and user account management with the domain model illustrated in Fig. 6.4. 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-oriented domain model with 7 entities with persistence and validation constraints markup [9, 10]. The business layer contains services with business logic, Create-Read-Update-Delete (CRUD), and search functionality. The presentation layer contains UI implementation using JSF technology.

First, we consider this application with a single UI. The UI part of the application contains search with result listing and a detail and modification page. The presentation covers the entire domain model in Fig. 6.4. Illustration of a simple page fragment, a form, is shown in Fig. 6.5. Form submission of data is validated through enforced business constraints upon 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 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 domain model and its constraints [9, 10]. Next, we implement the same application using READ. The data instance source code is extended with additional presentation marks [A.2]

Table 6.2: Efforts comparison for simple UI

Evaluated concern	**Approach**		
	Manual	READ	READ reusing generic elements
Java LOC	1342	1530	1439
UI XML LOC	2221	1715	1534
Conf. XML LOC	373	442	373
Restated inf.	564	0	0
UI Conditionals	0	0	0

extending the field constraints (see example in Listing 6.3). 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 (layouts do not reference specific fields), which considerably reduces occurrences of restated information in the XML. This results in 1530 LOC of Java, including the additional domain 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 domain model in the UI, whereas 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 Table 6.2. At the same time, it should be noted for READ that if custom layouts apply, they most likely reference field names and increase the restatements. Furthermore, in the results we did not consider as a restatement the reference to the data instance that is passed to the READ components in the UI code. The aspect weaver itself is not included in the evaluation because it is a generic, reusable, and external library (reasoning is given in [4]).

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 a 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 context-awareness. 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. Further-

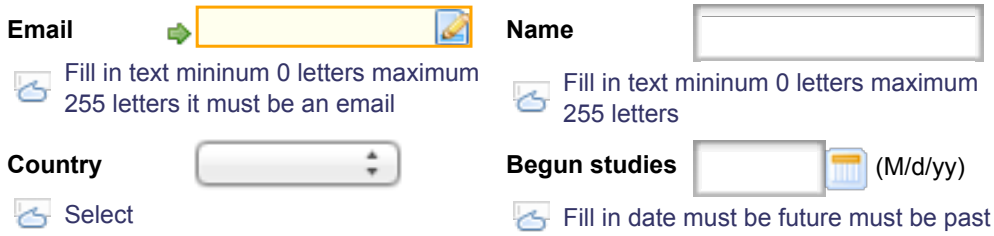

Figure 6.6: Sample form for confused student

Figure 6.7: Sample form for child

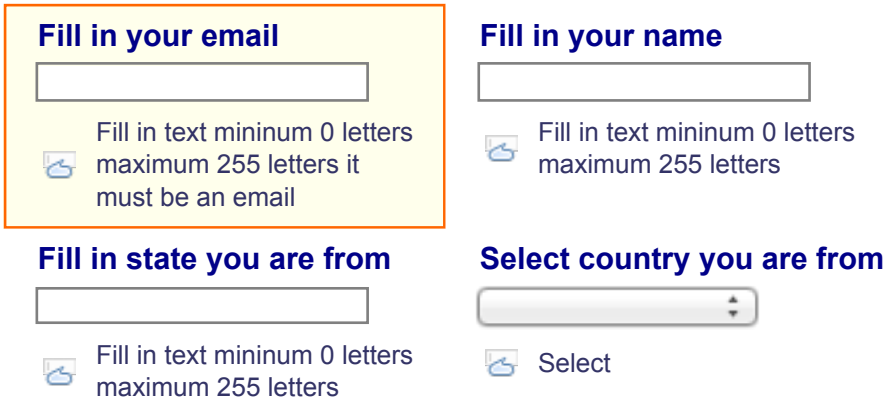

Figure 6.8: Sample form for elderly

more, we provide 4 different presentations for children, elderly, adult, and experienced users, all possibly combining a given layout (see UI examples in Figs. 6.6-6.8).

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. Also presentation cannot be separated from field binding and property settings. The mainstream approach results in 1658 LOC of Java and 13072 LOC of XML presentation, which includes 240 UI conditionals and 6768 restated information from the domain model. Consider that with this approach, developers follow the implementation in Fig. 6.2 (b).

The READ approach allows designers separation of presentation, layout, security, and location-awareness through various stages within the framework. One of main differences in the READ approach is that each concern is implemented separately as demonstrated in Fig. 6.2 (a). The READ weaver then combines these together. In our study,

Table 6.3: Efforts comparison with context-aware UI

Evaluated concern	**Approach**		
	Manual	READ	READ reusing generic elements
Java LOC	1658	1907	1754
UI XML LOC	13072	5036	4508
Conf. XML LOC	373	649	373
Restated inf.	6768	0	0
UI Conditionals	240	20	20

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 domain model, this reduces the original 240 conditionals 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 6.2 to ignoring specified fields. The overall summary of the evaluation is provided in Table 6.3. 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 domain model in the environment with weak type safety. Changes to a data field, its name, or constraints causes inconsistency in all 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 the READ approach, there are fewer UI references to the data elements; thus it does not require many UI corrections. 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 require the 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 the domain 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.

Table 6.4: Performance comparison

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

For the performance evaluation, we consider a page with 5 forms with total of 21 fields. The time needed for page load is evaluated, considering both the conventional design and READ versions. The load times for a page containing the forms, averaged over 250, samples were 545ms (std. dev. 47) for the manual approach and 539ms (std. dev. 41) for the READ approach. Results are highlighted in Table 6.4. The measurement shows that the page load time is similar for both approaches.

6.4.2 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 contest management system described in Section 6.4 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 6.4.1, 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 domain 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 the manual approach was used, each form would combine multiple concerns, and thus the change would impact up to 21451 LOC of Extensible Hypertext Markup Language (XHTML). Instead with READ, we could solely focus on a single concern (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

Table 6.5: Case study summary

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

tables, which did not apply READ approach, took considerably longer time because it entangled multiple concerns that were reused in many locations.

Our code measurement in the production system provides the following results. Out of the 70 entities in the domain 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 21451 LOC of XHTML code. This represents approximately 32% of the entire XHTML 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 6.3). This represents 9-13 references per field in the UI component (see Listing 6.5); 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 a mean value of 7.5 fields per entity (median 6) with a standard deviation of 4.85. When counting that a UI widget has approximately 11 references to the domain 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 6.5; it shows that the use of READ framework reduces the volume of UI code. A 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 6.5 gives our estimate for an average entity of 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 correlates with the estimate given in [19]. Regarding the performance, there is no performance reduction exhibited, and the community has reported no performance issue. The READ is applied in production for the ACM-ICPC contest management system at the time of publishing this thesis.

6.5 Summary

In this chapter, we introduce the READ approach that aims to address two issues related to conventional UI development: information restatement and cross-cutting concerns. The UI design is analyzed to derive involved concerns that should be addressed separately in the design in order to improve reuse, development, and maintenances. Information reuse is addressed through MP and data inspection, which avoids inconsistency among backend and frontend parts. A direct transformation of the inspection result to the UI is avoided, as it would not provide flexibility to consider other concerns. Instead, an indirection is considered. The data structural elements are identified as join points as well as the elements from the application runtime context. Considering join points and their constellation in a given data field allows us to deduce generic transformation rules that can be reused across various fields and domain objects. AOP-terminology works well in describing the transformation. Transformation rules use pointcuts and advices to determine particular DSL templates for the next stage. Such a DSL template, supports custom presentation definition in the target language and give the ability to integrate other concerns applying integration rules that follow the same concept as transformation rules. Based on pointcut, integration rules determine whether or not to integrate a particular advice to the template. On the top

sits the layout integration. The result of the run-time transformation is code fragment in the target UI language representing given data and context in the UI. Finally, the code fragment is interpreted and embedded to the component tree for a particular UI page.

In our evaluation, we implement a tool for Java EE, capable of runtime transformation to JSF. The advantages, which can be also seen from the presented case study, are: reduction of restated information, reduction of restated decision, reduction of code volume, error-avoidance regarding inconsistencies, generic and reusable transformation rules, addressing of cross-cutting concerns, support for third party and extensions, runtime context integration, synergy with OOD and adaptable towards changes. When reusing templates and transformation rules, it produces output quickly with low efforts, while allowing full output customization. It further reduces development and maintenance efforts and supports single focal point perspective. As shown in comparison with JSF, the performance does not need to struggle, although it strongly depends on a given target language. The limitation can be seen for the last stage, which is the interpretation of the resulting target language fragment, integration to the component tree and embedding to the rest of the page. This is very specific to given view framework, and advanced framework knowledge might be required for achieving it.

7

Integration with other Context-aware UI approaches

Writing one-line programs is not usually what we mean by design. Software has improved a lot over the years, and a lot of systems that used to require careful design can now be built by reusing other software. But there are a lot of systems that we can't build that way, and writing 100K lines of new code isn't that much easier now than it was 15 years ago. It will do a lot more, but costs the same.

-Ralph Johnson

In Chapter 3 we introduce existing Context-aware User Interface (CaUI) approaches. These approaches provide great features to support usability although they are prone to information restatements. In order to apply such an approach to existing applications or to even new application that build on code-based approach for data management and follows existing standards (such as Java EE), the approaches require to reference and restate already captured information, which is error-prone and tedious. This chapter shows READ integration to another CaUI approach to extend its capabilities with

reduced restatements as well as the development and maintenance efforts [A.3, A.21]. This chapter considers User Interface Protocol (UIP) approach for CaUI that can stream the UIs to various platforms, while using native platform components and features to improve usability, it also adjusts field layout based on metrics, etc. The integration with READ is demonstrated on a case study. The study demonstrates reduction of restated information, reduced the development and maintenance efforts and also simplifications for integrating UIP to EAs, (e.g. Java EE applications). This chapter bases on work published at [A.3, A.21].

7.1 Motivation

In Chapter 6, we proposed the READ approach to reduce development and maintenance efforts of UIs and even of CaUIs. Many alternative CaUI approaches exist that provide extended features, such as automated element layout composition for given devices, support for native platform elements and native platform features, etc. UIP [14] is an example of such an approach.

The main goal of UIP is to deliver optimized UIs to different platforms. UIP introduces one more level of indirection. It considers an abstract structural description and given UI that is the same for all platforms. The optimization is based on the structural description and a presentation context that influence the presentation and resulting format of the provided UI. The UIP concept is based on the client-server architecture [6, 8]. A set of thin UIP clients connects to a single UIP server. The UIP server streams UI descriptions to various heterogeneous clients that interpret the UI on given platform using native components. In contrast to web-based clients, UIP clients use platform-specific presentation.

The server-side employs a main component for context sensitive UI generation called the Concrete User Interface (CUI) generator, which produces concrete UIs. It takes as the first input an Abstract User Interface (AUI) specification captured in DSL [27]. An AUI specification is a hierarchical composite structure [A.3] that describes UIs in a platform-independent manner. This structure specifies what the UI should consist of (input, output and action triggers), although there is no specification of the current representation of individual elements and the layout. The second input into the UI generation process is a context model. The context model is based on the concepts of ability-based design [81] and is aware of the target platform. The context model together with the AUI specification are composed together based on optimization met-

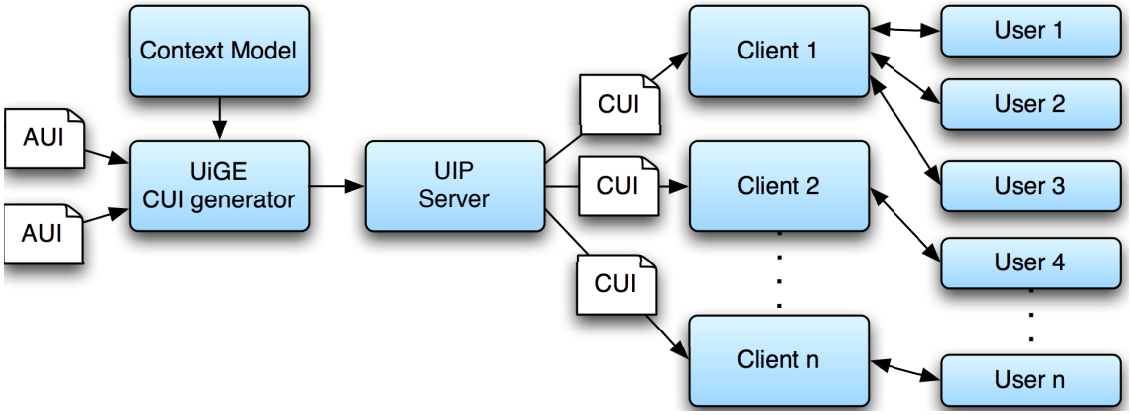

Figure 7.1: UIP platform overview [A.3]

rics (e.g., minimal user effort) and produce CUIs. These resulting and optimized CUI descriptions, together with data, are streamed to various platform-specific UIP clients mentioned above. A sketch of the approach from [A.3] is provided at Fig. 7.1. In the opposite direction, clients submit events of user actions. There are a number of UIP clients based on various platforms, such as desktop version for Personal Computers (PCs), tablet PCs, smartphones, smart TV emulator and web.

7.2 Problem Description

The above description shows that UIP provides reasonable features that are good to have but certainly hard to develop. On the other hand, as described in Chapter 3, UIP struggles with multiple disadvantages. Most likely when we want to integrate UIP to an existing system, we need to restate all the data information in the AUI, which is error-prone and tedious; since it uses a DSL, we have very limited type safety. It struggles from evolution management, since changes in data representation are not reflected in the AUI, and thus not in the UI. Next, the presentation context considered in the CUI generation may consider contextual information, although distinct from the application context of the existing system. Such application context may influence the AUI description at runtime; thus a static design of the AUI may not be efficient and sufficient. UIP suggests designing AUI by hand or through visual editor. Although the editor helps to avoid errors in the DSL description itself, it does not help with restated information.

7.3 READ Integration

When considering an existing application backend, typically an EA, such as Java EE with JPA [9], then multiple, already captured information would need to be restated in the AUI. As demonstrated in Chapter 6, since it is possible to apply code-inspection to the data structures, consider multiple, additional aspects, and transform all these through READ aspect weaver via DSL composition templates to CaUI, then it is possible to automate the AUI descriptions generation as well. The integration templates can use the AUI as the target language and derive the AUI description from the EA backend. Furthermore, when this approach operates at runtime, the AUI description can be influenced by given runtime conditions and thus affect the resulting UI.

Such READ-UIP integration avoids error introduced though manual AUI design, prevents inconsistencies in the UI, and reduces the time needed for AUI integration. Next, it does not bind the developer to base cases, since he or she can define custom composition rules or define custom composition templates. It may further enforce security, a concept missing in UIP. It also supports evolution management since changes to the application backend take an immediate effect in the UI.

[A.3] addresses the integration of UIP and READ, as can be seen from the sketch at Fig. 7.2[1]. The existing system is bound to the UIP approach through AUI descriptions and through domain model mapping. The CUI generator then process the AUI and context model and produces a CUI description that is transmitted to a custom UIP client, together with data[2]. Next, note the use of optional UIP templates in Fig. 7.2. Such a template is a complex mapping variant that provides the relationship between a subset of an AUI and platform-specific structure. A UIP client can send events back to the UIP server for processing. For example, consider a client-side event demanding data modification, which is processed through the server-side event handlers that perform the data changes.

7.4 Evaluation

The evaluation is made based on a data-oriented EA that involves a population census form [A.3, A.21]. The goal of this evaluation is to show the capabilities of the integrated approach to generate context-sensitive, platform-aware electronic equivalents of real

[1] AspectFaces library is the implementation of READ
[2] A more detailed description can be found at [A.3]

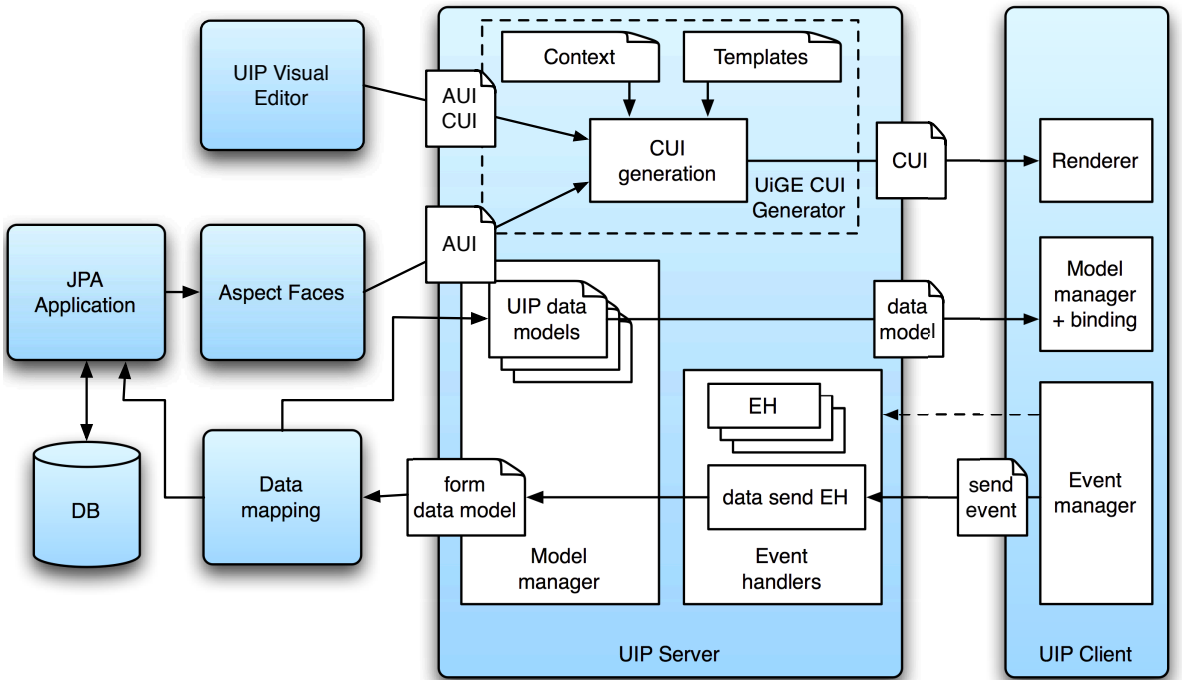

Figure 7.2: UIP and READ integrated platform overview [A.3]

physical forms. One of forms for the population census in the Czech Republic in 2011 was used. A set of forms for three different platforms was generated from a single AUI. The AUI was generated through READ code-inspection of the underlying JPA application backend.

Fig. 7.3 shows visualization at different platforms. Fig. 7.3 a) shows a UI generated for an iPad UIP client. In this case, an UIP template is used for the AUI mapping. It uses a complex platform-specific structure - UITableView. Fig. 7.3 b) shows a visualization of a UI generated for a desktop platform. The advantages brought by UIP are that the font-size, element-size, element spacing, and layout are influenced by the context-model. Using model-wide binding and server-side application logic, the UIP client displays warnings next to elements with content that does not pass the validation criteria. The most suitable mapping to actual CUI elements that visualize individual AUI elements is determined using combinatoric optimization.

Fig. 7.4 shows different UI variants generated for an iPhone UIP client. Fig. 7.4 a) shows a UI generated without UIP templates with the default context model. Fig. 7.4 b) depicts a UI that is generated for a user with slightly reduced vision and dexterity problems. Note that the size of the labels and also of the interactive elements is bigger. Fig. 7.4 c) shows an UI that uses the UIP template with native UITableView.

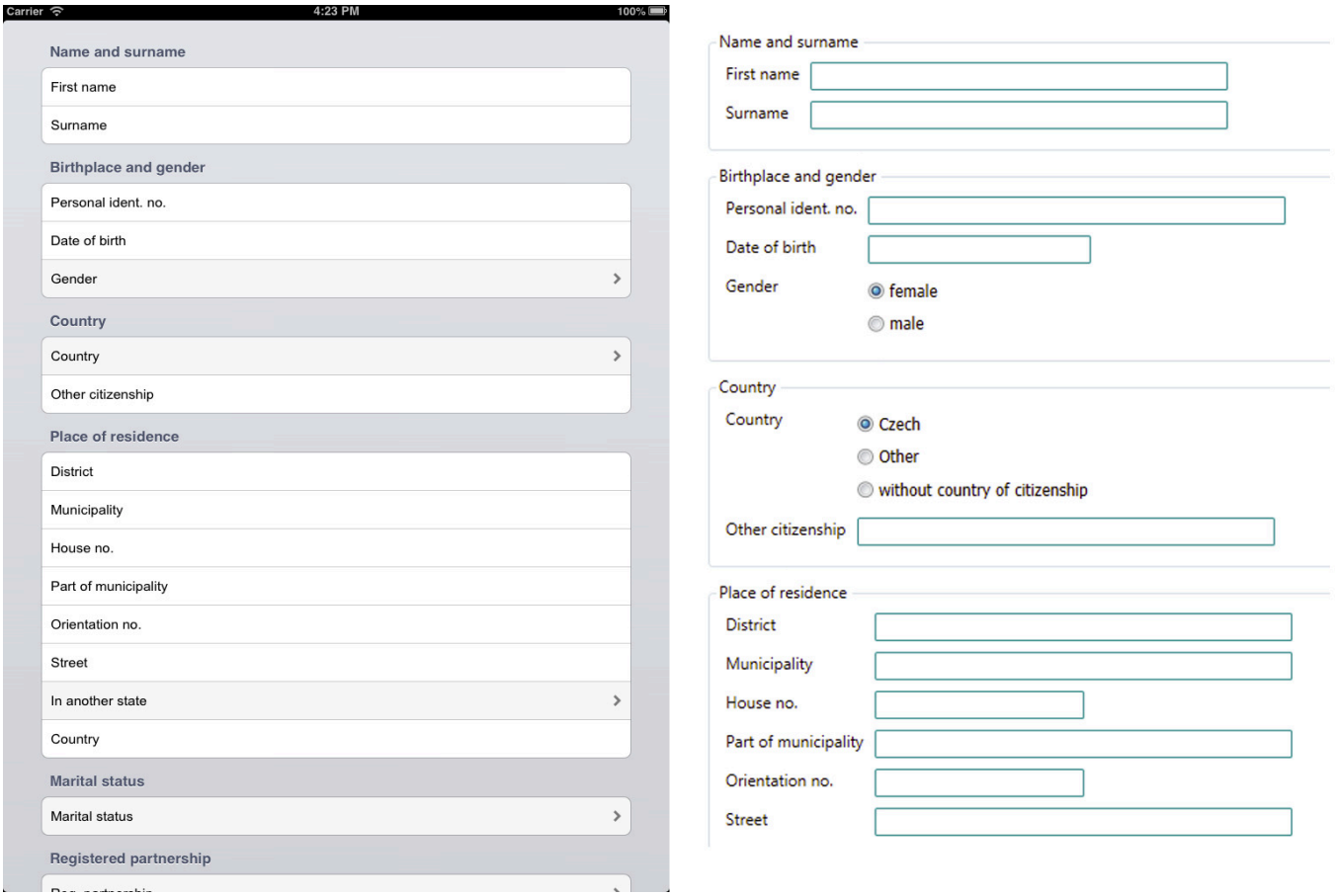

Figure 7.3: UI generated for: a – iPad tablet (left) and b – desktop PC (right) [A.3]

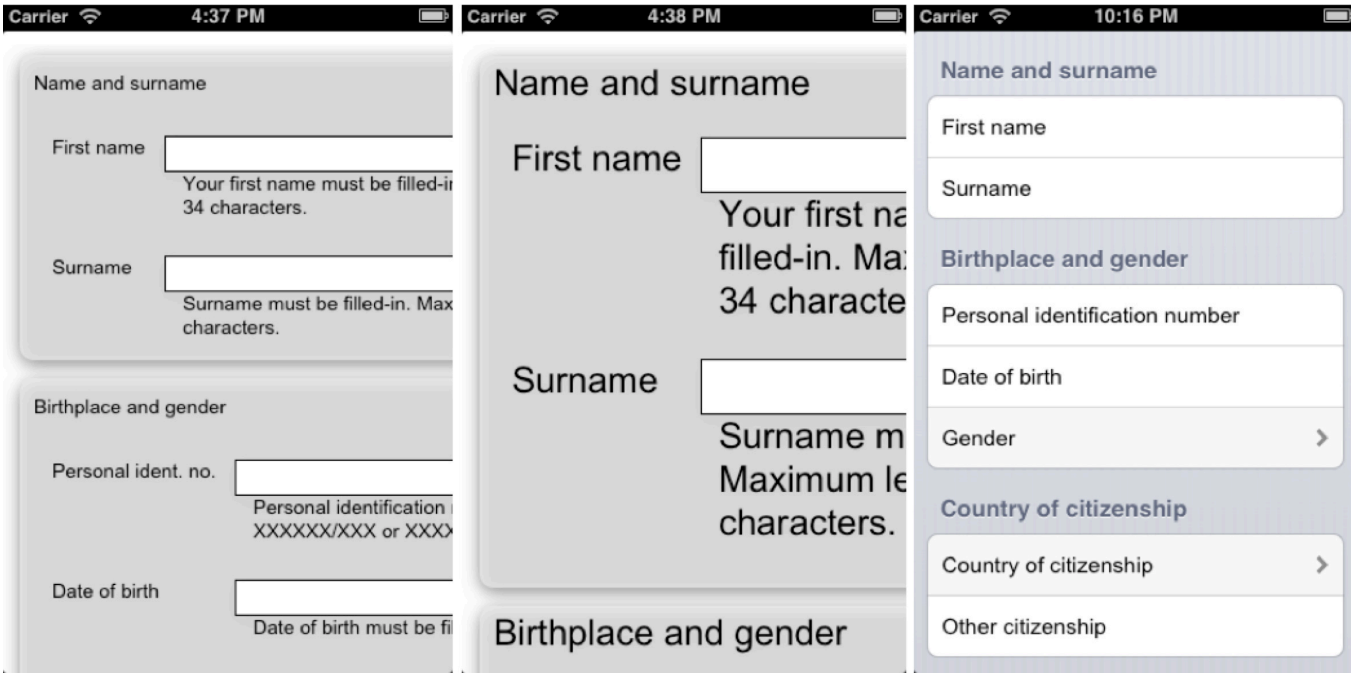

Figure 7.4: UI generated for iPhone: a – default context (left), b – for user with lower vision (middle), c – generated using templates (right) [A.3]

Similarly, [A.21] describes details of a user study that compares generated UIs with manually-implemented, simple, web-based UIs. Three platforms are used during the study - a desktop PC, a tablet PC, and a smartphone. The UIs were assessed subjectively in terms of comfort, efficiency, and aesthetic quality on the Likert scale [82] (1 to 5, where 1 is the best score). The Wilcoxon rank-sum test [83] is used to compare the effect of each UI on comfort, efficiency, and aesthetic quality. The user study showed that the generated UIs could provide a better subjective user experience for the mentioned qualities. The biggest advantage over the web UI is on the iPhone. In this case, the UI generated by the READ-UIP platform is based on easy-to-navigate iOS Table View. By contrast, the web page required a lot of scrolling and zooming, which worsened its usability. The study described in [A.21] is further evaluated from the perspective of development and maintenance. A manually-developed system is compared with a system built using READ-UIP integrated approach. The study revealed that only 874 LOC are required by READ-UIP integrated approach, in contrast with 1819 LOC required by the manually-developed system. Furthermore, with the READ-UIP integrated approach, the changes take place only in a single location, the domain model. Next, consider that there is a reduction not only in the source code, but also in the coupling among different subsystems. The time devoted to both development and maintenance is reduced because the UI part adjusts to the information already captured in the EA backend and no manual restating takes place.

7.5 Summary

The goal of this chapter is to show the synergy effect of READ with other context-aware approaches. This approach may act as universal code to DSL transformer and reduce efforts for developers in third-party integration or even to researchers to apply their prototypes to large application to demonstrate large-scale scenarios while avoiding extensive manual work and errors introduced from human-factor.

Furthermore, code reduction demonstrated in Chapter 6 does not relate solely to a given platform. At the same time, UIP may struggle with addressing runtime application context, security, location awareness, etc. These can be easily handled through READ integration, while extending UIP capabilities.

For possible improvements, the entire READ transformation process could be distributed between the client and the server using web services. For example the code-inspection and composition template selection could be made at the server-side. The

composition template integration may also be partially done at the client-side. This could considerably reduce the data transmitted between client and server, since all templates and data structures could be sent once, for instance in a JavaScript or JSON formats for web systems. They could be then cached and reused over the time. The server would consequently provide only the changing information and its usage and resources involved in the UI delivery would reduce. This extension is introduced in Chapter 8.

8

Distributed, AOP-based UI Design

Software design is always hard. Although most modern development environments do lots to reduce complexity through reusable libraries and toolkits (Eclipse, Apple, Microsoft), designing a software solution to the business problem is still hard.

-Richard Helm

Previous chapters introduced advantages brought by separation of concerns regarding the design of UIs. Although when it comes to UI rendering, the separation drops. This has the effect apparent in web-based application where the server-side put efforts to assemble the UI into a client-readable description in which it provides the UI to clients. Such description although entangles concerns, which extends its volume and disables individual concern caching and reuse at the client-side.

This chapter explores the possibility to apply the advantages of separation of concerns introduced in previous chapters to the UI delivery. It provides a case study applying such an approach to compare its advantages over the conventional form of delivery. The result brings decreased server-side efforts in the UI data presentation assembly, reduced

volume of processed UI description, extended options for concern caching and reuse, improvements to responsiveness, as well as the ability of the client-side to request a particular concern separately of other concerns. This approach pushes more responsibility to the client-side and partially delegates the UI assembly to clients. At the same time, the server-side resources are provided in machine-readable formats, which opens the server-side for further reuse. This is demonstrated by simplified implementation of native client-based applications that take advantages of native widgets, while reusing services and the UI configuration given by the server-side. The content of this chapter based on the research published at [A.1, A.6, A.7, A.12].

8.1 Motivation

Previous chapters introduced existing UI designs and development approaches and suggested various improvements to address restated information and cross-cutting UI concerns, which reduces development and maintenance efforts. Although these improvements bring many advantages, they do not take into account the underlying architecture from the perspective of communication with users and data transmission.

When considering UI delivery to remote clients, in most cases, the UI description uses HTML streamed to clients over the Internet. Although often supplemented with immutable resources such as images, Cascading Style Sheets (CSS), or JS, the description itself is provided as a single piece of information. Such a single block of information has limited caching options; the information volume provided in the block might be extensive. When we consider conventional approaches, they tangle various concerns together. Although the AOP-based UI approach such as READ untangles the UI concern descriptions, the concern weaving takes place at the server-side, and client receives a tangled result, similar to that received from conventional approaches.

Such concern mix is also evident in the streamed HTML. For example, an HTML data form mixes together field presentation, form layout, data binding, field validation, etc. A client web browser interprets the received HTML to present the UI to the user. This mix may consist of repetitive information [A.1, A.6, A.12] and even fragments that do not change over the time, but each time the page reloads, the entire block of information from the HTML must transmit again.

The question to address is whether it is possible to cache a particular concern from the UI description at the client-side. With the above approach, it can be hardly achieved. At the same time, consider that at the server-side we already maintain concern sep-

aration when involving READ or a AOP-based UI design approach; thus in order to provide similar concern separation to the client, the server-side concern weaving must partially delegate to the client. Consider a distribution of the AOP-based UI design approach between sever and client so that the server provides UI concerns to clients. At the same time, the server uses its internal information to enforce UI configuration, and the client becomes responsible for requesting UI concerns plus configuration and for performing the UI assembly. UI concerns can be requested asynchronously in parallel. Since the client requests given concerns separately, caching of individual concerns becomes trivial.

This chapter extends READ design with concern distribution and client-side assembly. The impact of the split concern streaming on the UI page load time, transmission size, content caching, and server resource use is considered. An experiment is performed to compare the proposed approach with the conventional, single-stream delivery with respect to transmission content size, load time, caching and server load.

8.2 Extending the READ Approach

In order to describe the distributed concern delivery extension to the READ UI design, we put into the contrast the conventional design UI approaches and iteratively extend them. The conventional UI designs expect developers to derive presentations of data elements considering a particular situation, context and data definitions that are further fragmented to fields and constraints. Throughout the design developers repeat their decisions to select UI components for given fields. The manually derived presentations struggle with coupling to the data definitions and information restatement as suggested in Fig. 8.1 (left side). The variability of context can cause the need to design multiple presentations for given data definition, which may include repeated decisions. Together this causes significant development and maintenance efforts as well as potential for errors. Limited type safety only deteriorates the efforts. Novel data definitions require to design one or even multiple new UI component.

Code-inspection-based approaches would follow the life cycle at Fig. 8.1 (right side). It shows an extension to the left side of Fig. 8.1 with emphasize on stages A-D, which reference below. Instead of making references to data instances, the UI component assembly deriving data presentation performs data definition lookup and code-inspection (Stage A) to derive the data structure. For each field it determines presentation (Stage B), usually, through hard-coded rules. More flexible mechanisms would use templates

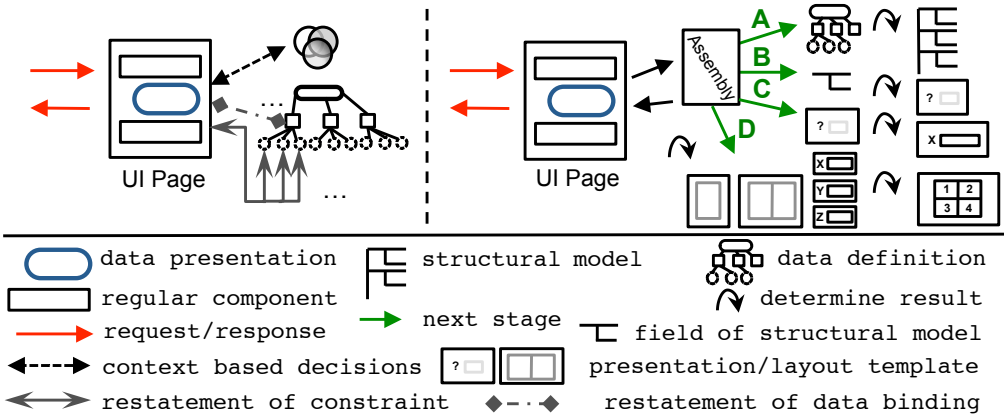

Figure 8.1: Conventional approach binding and restatements (left), Code-inspection-based approach life cycle (right). Assembly stages: A - code-inspection to passed data, B - selection of a field presentation, usually determined by field type, or custom rule, C - field presentation content resolver in case we use template for stage B, D - layout decoration.

(Stage C) allowing developers to adjust the output. Layout selected for particular presentation decorates field templates (Stage D), which produces the data presentation that the assembly embeds to the page. This can happen at runtime, but it is possible to have all presentations generated at compile time.

The AOP-based extension to the assembly in Fig. 8.2 affects the stages A-D. The extension can be seen in consideration of join points, pointcuts and advices (see the legend of Fig. 8.2). Stage A determines the data definition structure and it also considers runtime context to adjust it to a particular user and system conditions. This derives join point model that follow the structure of data and fields. The join points reflect both the data structure and the dynamic context. In practice, the structural model is cached, but the context and join point representation is resolved each time it is used and each time a presentation is requested. This allows UI adaptation to context-awareness.

Stage B aims to find an appropriate presentation for each field, driven by the data structure of join point model. The context can influence the structure. Instead of designating each field with a specific presentation template, we use join points to determine the appropriate presentation to generalize the field selection. A set of aspects is designed for this purpose. Each aspect advice determines an appropriate template to be used; the pointcut is a query to field's join points to the join point model. The pointcuts use an expression language (such as Java Unified Expression Language) that allows determining whether the pointcut applies; all this solely bases on field join points. Since this mechanism does not bind to specific field name or class, it is reusable and allows novel data definition to reuse it with no additional efforts. This is one of the key

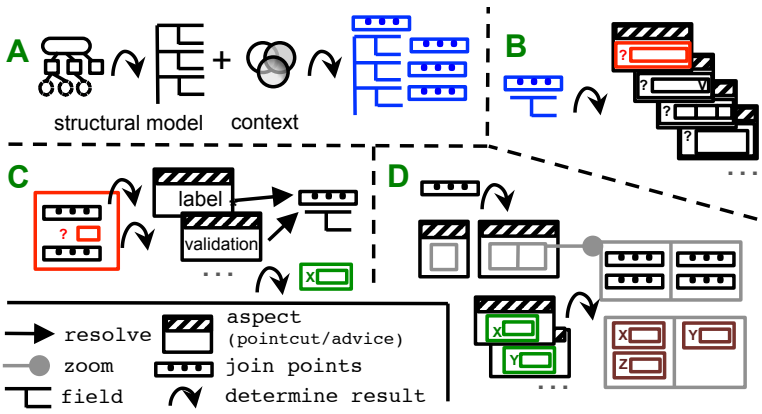

Figure 8.2: AOP-based extension to the code-based inspection assembly

features that allow reusing these aspects among any data class/field across the entire system. The pointcuts are generic, although the expressiveness is not limited to join points, any application context and be used as well, also field-specific selection can be applied.

The selected presentation template is interpreted using similar mechanisms. Stage C in Fig. 8.2 gives an abstract detail of the presentation template and repeats for all presented fields. The template gives a basic presentation for a particular field. It uses the target UI language and join points to integrate other presentation aspects and to incorporate field information. Within the field context, we resolve the template content and supply validation, conditionals, data binding, etc.

The layout integration in stage D uses template selected for particular situation and presented data. It uses the target language extended with join points that either references a specific field by name or an anonymous field. To avoid complex indirection the aspect can be inlined. Besides the specific/anonymous field references, the anonymous fields may use iteration to simplify dealing with repetitive layout fragments. For instance, a generic two-column layout description only captures two anonymous fields wrapped in iteration tag that enforces all data fields to follow the pattern within the tag. Furthermore, this mechanism can combine with specific field references that are stripped from the iteration and position to a designated position.

The illustration in Fig. 8.3 (left side) shows the services provided to clients, when considering the AOP-based UI design. Denoted by colors all stages ⓐ-ⓓ are considered. The aspect weaver does the assembly of data presentation(s) and is used by UI renderer that resolves page content embedding the derived data presentation(s). The rendered content is sent to the client-side. Clients interpret the provided description.

All the above approaches deliver the UI description as a single block of information. Although AOP-based UI design addresses separation of UI concerns for components reflecting data, the separation gets lost upon the UI delivery to clients. The provided description consists of repetition and tangled UI presentation. This naturally extends the volume of provided description, although the disadvantage is addressed by the HTTP compression. On the other hand, the server is responsible to use its resources to process and tangle all the concerns to derive data presentations. Thus it works with the full volume. From the clients perspective there is no mechanism to apply caching for the tangled concerns. To the contrary, consider concerns (such as these from Fig. 6.2a in Chapter 6) streamed separately to clients. Such distribution might appear to be an additional overhead, as we need to handle more requests. On the other hand, this may eliminate repeating patterns in the transmitted content and enable concern caching.

In order to design the distributed, AOP-based UI, the concern weaving process should be partially pushed to client-side. The application data definitions (or data transfer objects [8]) together with the application context are part of the server-side where the inspection takes place. This gives a join point model streamed to the client. Note that certain model elements might not be relevant to presentation or to a particular user. For instance, consider internal fields, primary key, version field, etc. The data definition constraints or the context solves this through the Annotation Driver Participant Pattern [13]. Thus only model elements that are relevant to user UI presentation conforming user rights are provided to clients. The selection of a particular presentation template for particular data field could be executed at the client-side; however, this would increase the complexity of the client since it must be aware of transformation rules and these may need to have access to internal join points or server-side information to resolve the decision. Thus this responsibility remains at the server-side, providing the result to the clients.

The right side of Fig. 8.3 depicts the responsibility assignments between server and client sides through service calls. Each client requests a HTML page that references client-weaver that calls for each data element a service that provides the filtered join point model enriched with the pre-selected template key ① (a,b). Presentation templates are provided ② as JS library. Each template has a corresponding key that matches the set of keys given by ①. The join points of a given data field from ① resolve the template matching the key. There is no enforcement for the client that would prevent it from considering local context in the template selection in fact multi-

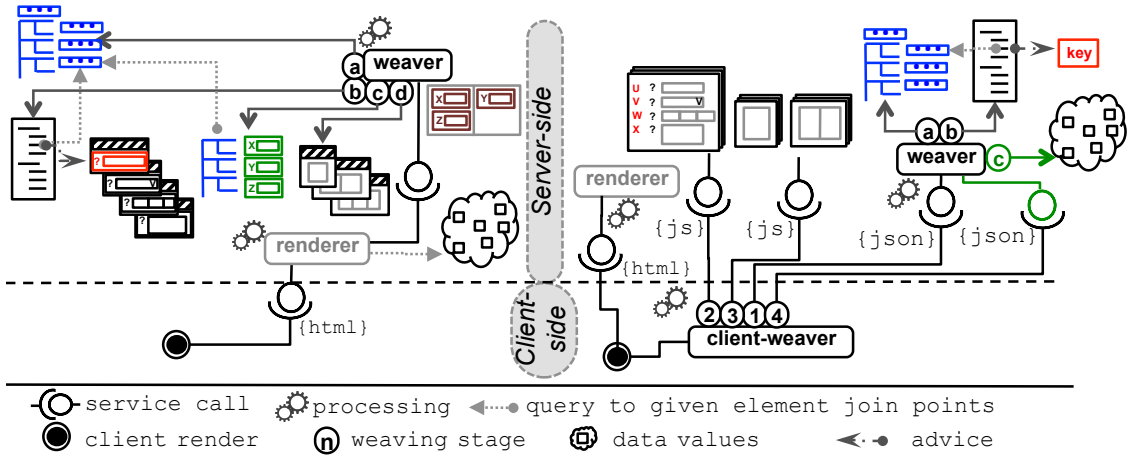

Figure 8.3: Services provided by the AOP-based UI design (left) / the distributed, AOP-based UI design (right)

ple template collections may exist for local context. Layouts ③, similar to presentation templates, are provided to the client-side for integration with the data UI presentation. Other concerns might be provided as separate services, integrated either at the server-side through transformation rules or via client-side presentation templates. Each client composes the UI data component based on received concerns that are influenced by system context. The server also provides the actual data values to the client ④. These values are displayed in the assembled UI component. Weaver can determine the matching data values from data element enforcing context and security. Data submissions use usual HTTP mechanisms or a web service.

The life cycle for the web systems works as follows. The user navigates to a particular page. This page consists of description elements from conventional UI design, which are ordinarily interpreted. The difference is for components representing data. These are replaced by custom tags and interpreted by a client-weaver (for example a JS call). The tag indicates which data to display and what local settings apply for the UI component assembly. The client-weaver requests necessary concerns from the server-side. The provided responses consider user rights, security and application context. As depicted in the right side of Fig. 8.3, the client-weaver generates the UI representation for a data instance given as a parameter of the custom tag. It conforms to its structure, application context and settings provided by the tag. The client-weaver may reuse a particular concern from cache. For example, presentation templates do not change throughout a long period of time; data structure might be immutable in a given context and user session, etc.

The extension to the AOP-based UI design is that concerns are provided to clients separately. The concern assembly is divided between client and server sides. The server-side does automated derivation of the join point model through inspection of data definition and context. This automated derivation provides the model reflecting actual application/user context. Information provided to the client-side avoids repetitions or tangling. The templates as well as the client-weaver are expected to load once by the client-side, the join point model may stay the same for context-unaware UI or may change with particular session or even request. The weaver can derive the data values using the same context and structure used for the join point model. Since the join point model and data values determine the UI presentation, the client assembly is not sensitive to particular data definition and thus the templates reuse over the time. There is no limitation to web systems and it is possible to design native platform weaver with templates reflecting the server provided template keys and the join point model determines the data presentation in native format, which supports further reuse. The abilities brought by the distributed, AOP-based UI are client-side concern reuse, possible caching of UI concerns, reduction of repetitions in the delivery, lessen server side involvement in UI rendering.

8.3 Experiments

This section provides experiments comparing the conventional approach with the distributed, AOP-based UI. The production-level EA ACM-ICPC contest management is considered. In our experiments, the web page for person profile management is used as it contains multiple forms presenting data. First, a subset of the page with 21 form fields is evaluated; next a 42-form field version is considered. The application is built on Java EE 6 (JDK 7) using JSF 2.1.18 and the PrimeFaces 3.4 library. The existing page of the EA is used for the comparison with the distributed, AOP-based UI for which we use AspectFaces library[1] to design the server-side weaver. The same production configuration is used with equivalent logical structure of UI elements for the data presentation at the page, while considering the same static resources for both approach evaluations. The 21 form field page versions are shown in Fig. 8.4 and Fig. 8.5.

In the measurement, the evaluated prototypes are deployed to a server at Baylor University in Texas with the parameters of 8 cores of 2.4 GHz, 16 GB RAM, and network access 645/185 Mbits/s download/upload. The client is situated in Prague with 4 cores

[1] http://www.aspectfaces.com, 2015

of 2.3 GHz, 16 GB RAM, and network access 10/6 Mbits/s download/upload. The CDN is a virtual server in Nuremberg, Germany with 2 cores, 3.4 GHz and 3 GB RAM with guaranteed 200 Mbits/s for download/upload. The round-trip time (RTT) between client and server is 150 ms and client and CDN is 20 ms. 50 measurements are made, and the average results with the standard deviation are provided in Tables 8.1-8.3.

To measure the load times, we use the standardized HTML5 Performance Timeline[2] that provides performance metric data for the given page. Specifically for the distributed, AOP-based UI approach that assembles the UI at the client-side, we consider the finish times of the UI composition. Thus for this reason we add a listener indicating the finish of the assembly (rendered UI presentation). The iterations JS script reloads particular page multiple times and stores the page statistics to Local Storage. This approach gives us minimal skew since the Performance Timeline already applies in existing web browsers.

The following measurements are made. First we measure the page size and load time to download and render the entire page requested from the server at Baylor University using the client in Prague. It involves requesting all page resources and page rendering at the browser. The client-side browser cache impact is considered in the evaluation. Second, to evaluate the network load, traces of resource loading are used. A simulator that interprets these traces is implemented with a thread pool of 10 instances that downloads resources reflecting the web-browser trace. Based on responses, the download time is measured, considering neither resource decompression nor page rendering. The simulator is used to evaluate the impact of the CDN at Nuremberg on the page load. In the third measurement, the server-side is evaluated with a stress test. One hundred simulators run simultaneously, following the trace for a particular page with given approach, while measuring the server-side the CPU usage.

8.3.1 Page Loads with Web Browser

The first measurement with the shorter conventional page (Table 8.1, Row *a*) and *disabled web browser cache* has a main document size of 74.4 KB, which compresses to 9.2 KB through gzip. In total the page calls 10 requests (including static resources), the transmission has 218 KB. Table 8.1 shows the results for conventional approach for Chrome$^{37.0.2062.122}$/Firefox$^{32.0.2}$/Opera$^{24.0.1558.53}$ web browsers (including JS process-

[2] http://www.w3.org/TR/performance-timeline, 2015

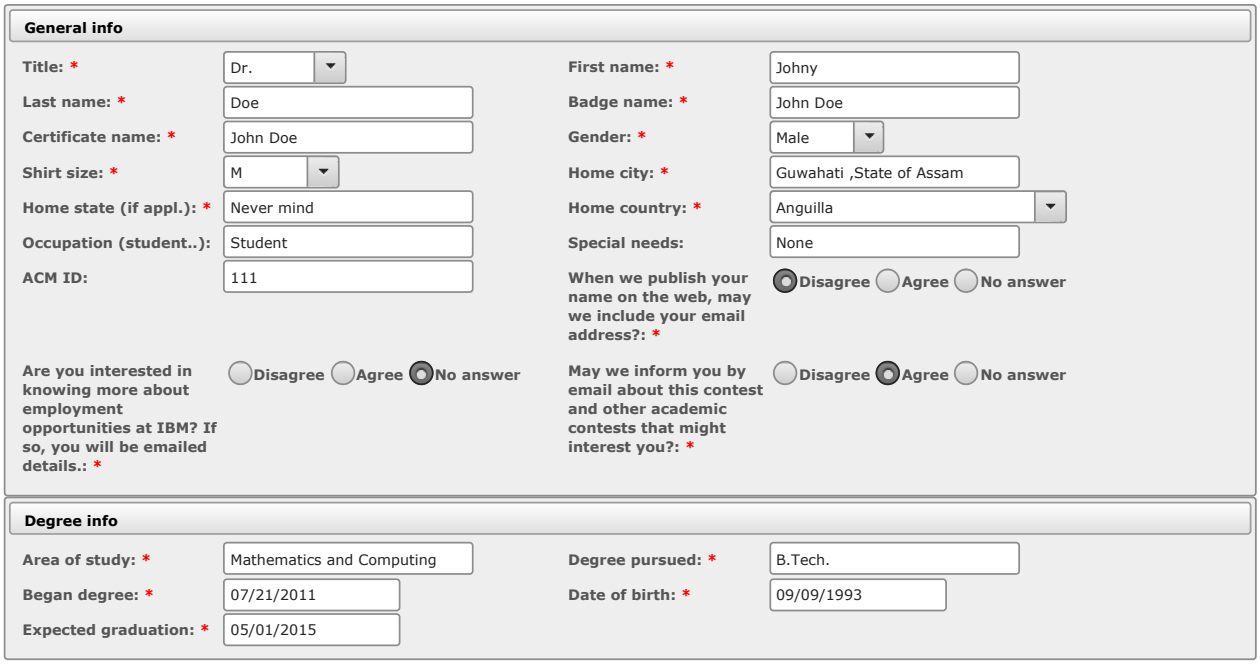

Figure 8.4: Evaluated UI subsystem designed with JSF approach

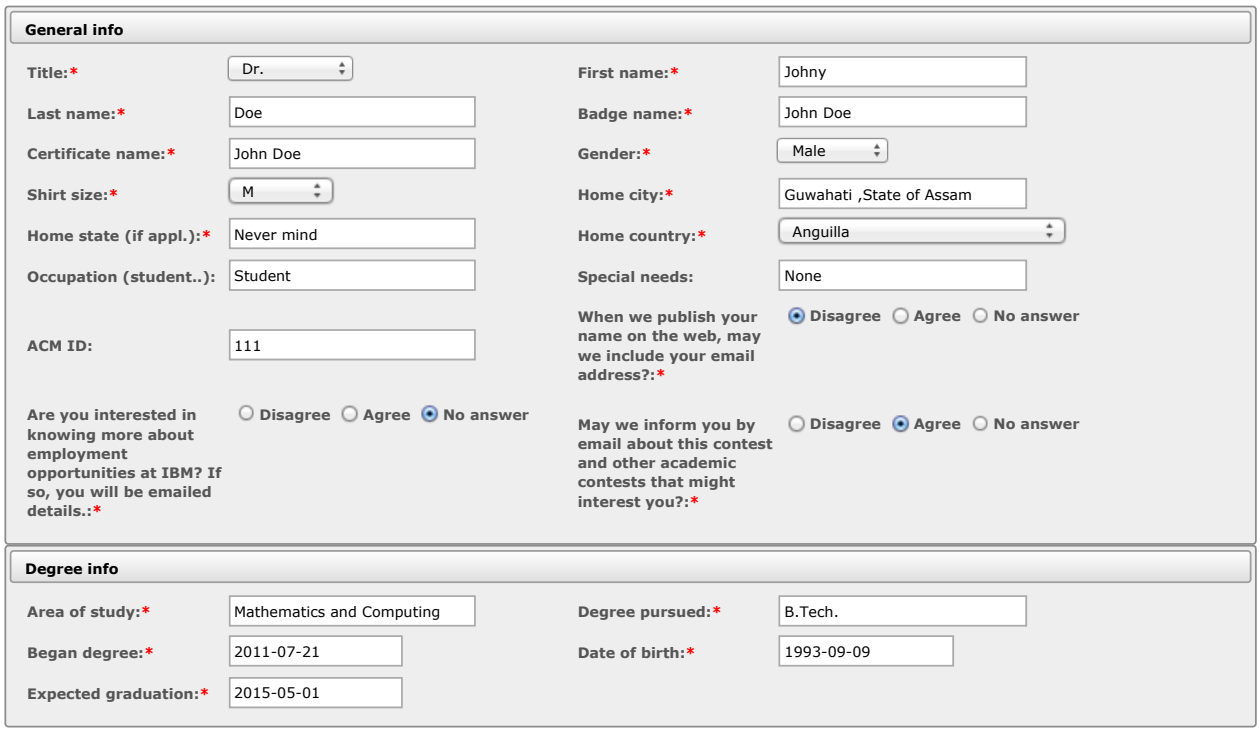

Figure 8.5: Evaluated UI subsystem designed with the distributed, AOP-based UI approach

ing). The distributed, AOP-based UI design approach (Table 8.1, Row *b*) makes 15 requests. The main document size reduces to 3.3 KB (1.3 KB compressed), although it additionally loads a JS assembly library (3.3 KB compressed), data (1 KB compressed), and join point representation (4 KB compressed) from web-resource. The transmission has 218 KB. Notice that additional resources load in parallel, simultaneously. The processed size of UI is considerably smaller (Table 8.1, last column). Row (*c*) gives

Table 8.1: Page load measurements with disabled cache

Row	Requested page	Cache	Chrome$_\sigma$ [ms]	Firefox$_\sigma$ [ms]	Opera$_\sigma$ [ms]	Requests	Transmission compressed	Processed UI size
a	Shorter conventional	No-cache	1637_{199}	1573_{420}	1540_{200}	10	218 KB	74.4 KB
b	Shorter distr. AOP		1419_{114}	1417_{187}	1402_{60}	15	218 KB	29 KB
c	% change		13%	10%	9%	33%	0%	61%
d	Longer conventional	No-cache	1691_{87}	1992_{367}	1669_{195}	10	220 KB	98.9 KB
e	Longer distr. AOP		1560_{92}	1772_{89}	1500_{90}	23	223 KB	37.1 KB
f	% change		8%	11%	10%	57%	1%	62%

the percentage improvement of load times ranging around 10%. The data presentation description volume that is processed by both server and clients, reduces by 61%.

The longer, 42-field conventional page has a main document size of 98.9 KB (compressed 11.1 KB), as shown in Table 8.1 *d*. The distributed, AOP-based UI design approach in Table 8.1 *e*, requires 23 requests since since it requests information from 6 data instances. The main document size 5.3 KB (compressed 1.6 KB), the join point representation 6.5 KB, and data 2.7 KB. Row *f* shows around 10% improvement in page load time. The data presentation information volume improves by 62%. From the results, we see improvement in the page load time. Although the UI description size reduces, the compressed form is the same size as the conventional approach. Notice that the number of requests grows, because data definitions and values are requested per given data instance. To overcome request growth, similar to static resource optimization strategies [A.20], this could be aggregated as evaluated in [A.1].

The Table 8.1 *a*-*f* shows improvement in the page load times and reduction in the processed UI description volume. The compressed transmission is equivalent, although both sides work with the original size. The number of requests in distributed, AOP-based UI design approach grows, because multiple data elements are involved in the requests.

The next measurement considers *enabled web browser cache* and repeats the above measurements (first miss not counted). The load times, shown in Table 8.2 *a*-*f*, drop significantly for both approaches. The conventional approach delivers the “tangled" main document, which is 9.2/11.1 KB for the shorter/longer version (Table 8.2, Rows *a*,*d*). The volume of data presentation description is the same as the uncached version.

The distributed, AOP-based UI design approach needs to transmit the main document and fetch data values, which gives 2.4/4.2 KB in 3/7 requests (Table 8.2, Rows *b*,*e*). There is a 15-20% improvement in page load time. The RTT is the dominant factor. The UI data presentation volume improves by 93% and 62-74% in the transmission.

Table 8.2: Page load measurements with cache enabled

Row	Requested page	Cache	Chrome$_\sigma$ [ms]	Firefox$_\sigma$ [ms]	Opera$_\sigma$ [ms]	Requests	Transmission compressed	Processed UI size
a	Shorter conventional	Cache	573_{21}	659_{49}	517_{12}	1	9.2 KB	74.4 KB
b	Shorter distr. AOP	Cache	456_{29}	552_{92}	446_{28}	3	2.4 KB	4.3 KB
c	% change	Cache	20%	16%	14%	67%	74%	94%
d	Longer conventional	Cache	657_{21}	858_{105}	607_{49}	1	11.1 KB	98.9 KB
e	Longer distr. AOP	Cache	526_{39}	593_{84}	519_{48}	7	4.2 KB	7.2 KB
f	% change	Cache	20%	31%	15%	86%	62%	93%

Table 8.3: Simulation download evaluation

Row	Cache	Page load$_\sigma$ [ms]					
		Shorter conv.	Shorter distr. AOP	% change	Longer conv.	Longer distr. AOP	% change
a	No-cache	1352_{66}	1182_{101}	13%	1379_{135}	$1213ms_{92}$	12%
b	Cache	312_{15}	353_{15}	12%	381_{31}	404_{63}	6%
c	CDN	631_{28}	511_{22}	19%	771_{26}	600_{17}	22%

8.3.2 Page Loads with Traces Involving CDN

Next evaluation considers the simulator that interprets browser request traces. The experiment is conducted to see the network impact, while considering neither the DOM construction nor resource decompression. The resource load results are shown in Table 8.3. At Row (a), the measurement does not consider cache and shows improvements similar to web browsers. The caching trace experiment results at Row (b) shows better results for the conventional approach. The third measurement considers CDN with cacheable resources available at server in Nuremberg. Row (c) shows results demonstrating the distributed, AOP-based UI load times reduced by approximately 20%.

Although Rows (a) and (c) show improvements comparable with web browser results, (b) is worse. The explanation is the dominant RTT, which in the conventional approach occurs once but in the distributed, AOP-based UI multiple times and multiple data element requests are made. Notice this experiment only involves network communication.

8.3.3 Server Impact Evaluation

The server-side impact evaluation involves the simulator from the previous evaluation and implements a stress test with 100 clients loading a particular page simultaneously. We measure the server CPU load during repeated page loads given four pages uncached and cached at the client-side. A Unix tool “sysstat” is used to get the CPU load every second during the stress test. The results for particular pages in Fig. 8.6 and 8.7 show that CPU load is lower and shorter for our proposed approach, due to lower data transmission volume and delegated (offloaded) UI composition to client.

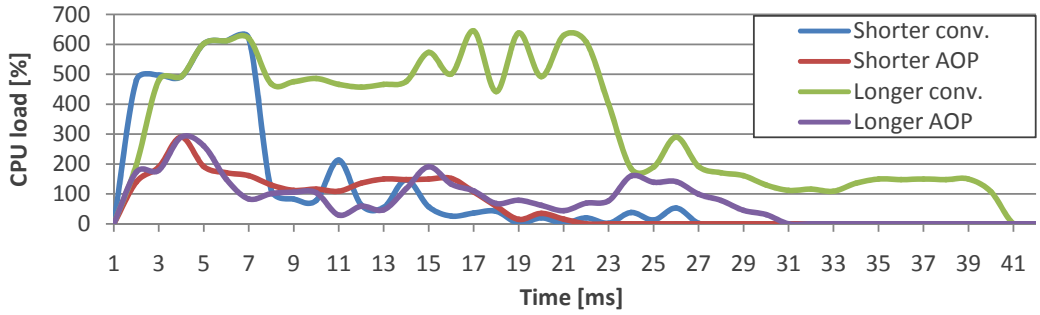

Figure 8.6: Server impact evaluation - CPU load (no cache)

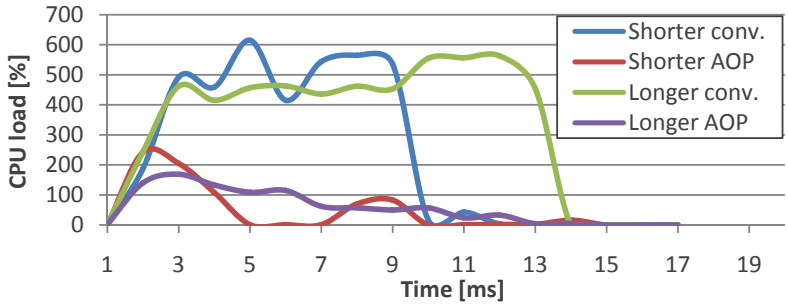

Figure 8.7: Server impact evaluation - CPU load (cache)

8.3.4 Comparison with GWT

GWT, introduced in our related work, targets improvements to UI caching. To compare it with the distributed, AOP-based UI design approach, a 23-field page version of the application prototype is implemented with GWT and distributed, AOP-based UI. GWT does not use the same static resources and JS libraries thus the distributed, AOP-based UI page prototype is modified, although there are still notable differences in both prototypes regarding linked JS libraries. The distributed, AOP-based UI prototype in previous measurements links generic JS libraries related to given JSF component provider, here these libraries are removed. The GWT prototype does not link generic JS libraries. There is a notable difference in the evaluation with disabled web browser cache, relevant to different JS libraries, although once the web browser cache applies, the libraries are cached and not requested; thus does not impact the evaluation.

The distributed, AOP-based UI prototype needs to transmit 41.8 KB to build the UI at the client (161 KB uncompressed). The GWT version needs to transmit 102 KB (379 KB uncompressed). The main document in GWT is partially converted into JS with a cacheable fraction of 141KB (50 KB compressed) and a non-cacheable fraction with 7.2 KB (3.4 KB compressed).

Table 8.4: GWT comparison, transmission of UIs with 23 fields

	No-cache		**Size (KB)**	*Cached*
	Uncompressed	Compressed	Uncompressed	Compressed
GWT	379	102	11.9	5.8
Distrib. AOP-based UI	161	41.8	3.9	2.1

The cache-enabled, page of the distributed, AOP-based UI prototype transmits content of 2.1 KB, which is 3.9 KB uncompressed. The GWT. cache-enabled version needs to download the HTML page, displayed data, and non-cacheable JS fragment, which in total is 5.8 KB (11.9 KB uncompressed). The results are summarized in Table 8.4. The GWT version still presents tangled code through mixed concerns, which extends its size. The approach with separately streamed UI concerns can reduce the UI description and improve caching beyond what is brought by GWT approach.

8.3.5 Threats to Validity

To mitigate the impact of network fluctuation to the measured results of page load times we averaged 50 samples of the same scenario, where we re-run each measurement right after each other. The measurement involves HTML5 Timeline that mitigates skew results. To avoid false times related to client-side execution, for the distributed, AOP-based UI approach we explicitly consider the finish times of client-side UI assembly, even though this might give better timing to the conventional version. Next, we provide criteria of evaluation perspective that are not sensitive to network changes and fluctuation. Specifically we measure the total volume of transferred data, and the volume of UI assembled by server. In addition, we consider the impact of caching abilities that are reflected by the transmitted volume.

Our application is one representative of a real-world application. The selected page reflects part of the application; we aimed to mitigate the specificity of the particular page size by considering page content extension. At the same time, the representative does reflect neither all aspects of the data presentations, nor all conventional approaches. The constellation of client-server represents one particular scenario to provide real-world results rather than a wide spectrum of constellations. We could use laboratory environment for the study, although the goal was a demonstration on a real environment to provide practical impact. This case study serves as a demonstration of the proposed approach impact on performance and transmitted volumes when compared

to conventional concern-tangling applications. The study considered the ability of a conventional web-browsers Chrome, Firefox and Opera.

8.3.6 Summary

Streaming various concerns separately from server to clients improves page load times for UI fragments presenting data (considering complete UI rendering) and improves caching options. In our evaluation, we streamed presentation, layout templates, data structure with applied security, as well as the actual data values. The distributed, AOP-based UI design approach extends caching options for concerns that are usually tangled together in conventional approaches.

The study on a production system shows page load time reduction in the range of 10 % and further to around 15 % when considering cache. The extended caching capabilities shown in our experiments reduce the transmission to 62-74 % compare to the JSF approach. The volume of information process by the server-side that corresponds to the data UI presentation reduces by more that 90 % using the proposed approach.

With trace simulation that considers solely network transmission and does not consider UI rendering and resource decompression, we see that requesting UI concerns separately has limitations in the transmission. Uncached trace transmission is similar to browser results and shows the distributed, AOP-based UI design approach to be faster, although we show cache-enabled evaluation to be slower, since most UI concerns are loaded after the main document transmission finishes in the distributed, AOP-based UI design approach. With trace simulation, we also evaluate the use of a CDN, which shows that the distributed, AOP-based UI design approach outperforms the conventional approach and reduces page load to 20 %.

By stress testing the server side, we demonstrate considerable reduction to the CPU load for the distributed, AOP-based UI design approach, which is demonstrated by graphs in Fig. 8.6 and 8.7. This corresponds to the reduce volume of server processed information.

The approach was also compared with GWT. The transmitted content volume with cached resources reduced by 63%, although a broaden study should be considered mostly regarding to the specific aspects involving multiple page states.

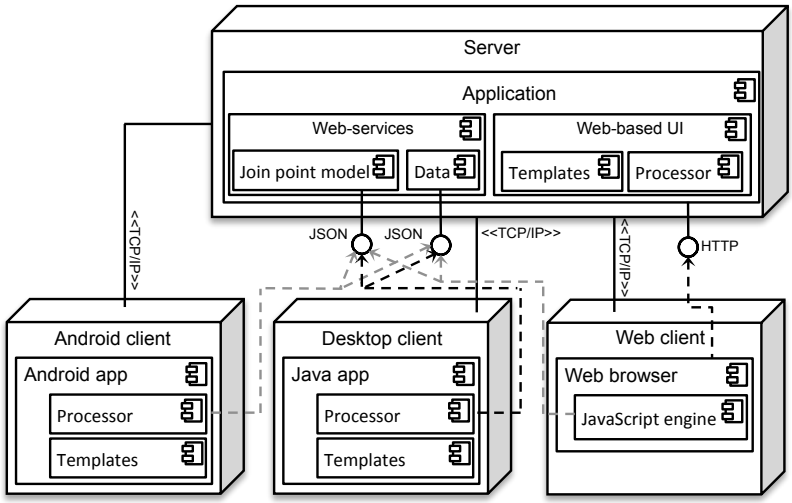

Figure 8.8: Sample of deployment diagram considering three heterogeneous clients

8.4 Native Platform-specific UI Clients

In addition to the above benefits, the distributed, AOP-based UI design approach presents parts of the UI description in a platform-independent way that is easy to process by other programs [A.7, A.22]. For example, it is easy to build mobile or standalone clients using the same web-services that are used for the web-browser-based clients. Consider Fig. 8.8 that shows the above situation.

Client applications provide a set of platform-specific templates matching the expected UI widgets and implement the client-side weaver that interprets the join point model and integrates data values. Next, it should consider native layouts.

In addition to the distributed, AOP-based web UI in Fig. 8.5, we implement an Android mobile app in Fig. 8.9 and 8.10 and a standalone Java Swing version in Fig. 8.11, which all use the same application server, business rules, data, services and context.

When server-side data change, all the client applications reflect the change, since the data UI presentations adjusts to the updated join point model. Platform-specific clients take the advantage of native widgets for the UI presentation to improve usability, although, the application page-flow is left for custom implementation.

Compared to UIP [A.3] discussed in Chapter 7 the server-side does not need to change with a novel supported client-side platform. Instead, the same information are provided across various platforms.

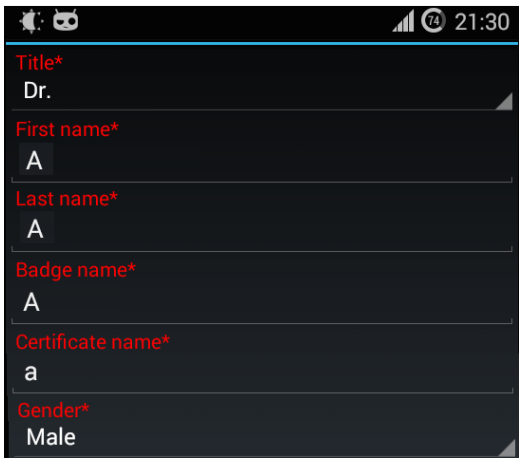

Figure 8.9: Android-based UI

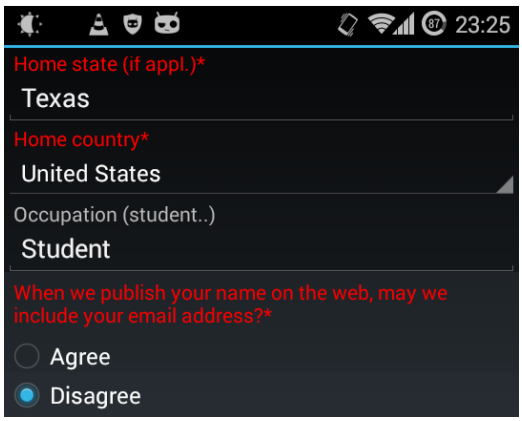

Figure 8.10: Android-based UI

Figure 8.11: Java Swing-based UI

8.5 Conclusion

This chapter considers alternative UI delivery for data presentations that involves separation of concerns. Conventional designs, as well as AOP-based UI design, mix various concerns together in the UI delivery, which may produce extended volumes and limit the client abilities regarding to concern reuse and caching.

Proposed distributed, AOP-based UI design may reduce the volume of transmitted information, positively impact responsiveness, extend client-side caching options as well as reuse of specific UI concerns. Reduced information volumes processed at the server-side positively impact resource use, such as CPU. On the other hand, such approach increases the amount of requests, although supporting concurrency. Provided case study demonstrates the above mentioned advantages of the approach.

The approach provides information in a machine readable format, which opens the application for further reuse across different platforms.

The results are promising, although limitations must be considered. The design approach fits well for data presentations; it builds on the top of other approaches that deal with interaction, page-flow, etc. Next, a small amount of field components exist, such

as suggestion boxes, and data manipulations, these must be defined, although it is easy to consider existing library integration, HTML5 components or custom presentations for fat-clients.

9

Future Work and Synergy

Time abides long enough for those who make use of it.

-Leonardo da Vinci

The presented research spans through multiple disciplines and areas from Software Engineering, Model-driven development, Networking and touches also HCI.

The extension to the accomplished research is given by [A.5, A.10, A.23]. It considers generalization of READ and AOP-based mechanisms to the EA development, it also presents the idea of capturing business constraints and rules in a DSL and weaving them to the entire application to conform the constraints at all application levels. The AOP provides efficient instruments to integrate various system concerns. In result such extension can be applied to the UI generation as well.

Next potential area of its application is its use for application middleware composition and for system integration. The predictiveness of UI element naming and identification given by rules in the generation process open possibilities for UI testing automation. With the ability to predict identification of components representing data, their binding to particular data and mostly with the knowledge of the meaning of each particular

UI field makes it suitable to automated large sets of UI tests. Development of such test is usually complex and delicate to application changes, which requires considerable maintenance efforts. The test generation automation for UI testing would reflect future changes to data and thus reduce maintenance efforts. Although these topics are promising, they are left for future work.

Similar to what was achieved in conjunction with UIP [A.3, A.21], in a defended diploma thesis in which we applied READ to other CaUI libraries XML User Interface Language (XUL) and User Interface Markup Language (UIML) and compared the efforts with JSF and UIP approaches. All the descriptions we generated by READ to reduce development and maintenance efforts. The result of this thesis is subject of a research publication.

The distributed, AOP-based UI design brings further potential for research mostly from the perspective of platform-independence. The use of the distributed, AOP-based UI design approach can consider, besides the client-side caching and CDNs, also involvement of cooperative web caching [A.9, A.19, A.27] in P2P overlay networks. With this approach it would be possible to further improve caching through distribution the UI fragments that are reusable.

In submitted journal paper that is under review we elaborate the idea of delivery strategy selection based on context. For instance, users with desktop have the ability to render their view using the all-state GWT approach. Although, when client poses restricted abilities, such as phone with discharged battery of smart watch, the GWT would be an overkill. Using the the distributed, AOP-based UI approach could fit better, although what is the impact on the battery use? Should it rather consider server-side rendered UI where the client only interprets the UI. Selection of an appropriate delivery/rendering strategy based on context is subject of next research. We believe that AOP-based approach has the prerequisites to effectively deal with strategy selection, while mitigating the development and maintenance efforts.

10

Conclusion

A conclusion is the place where you get tired of thinking.

-Arthur Bloch

This work addresses multiple problems in the area of UI design for data presentations. In order to reduce restatements and repeated decisions in the UI design the MDD approach was considered to derive data presentations from UML class models. Although, it is possible to derive UIs from UML class models, often the result is poor and requires manual changes to generated descriptions, which disables future derivations from the model, since manual changes would erase. Suggested solution is an extension to UML class models that captures additional information through UML profiles and stereotypes that allows deriving UIs that conform data constraints, input validation or security. Such an extension provides the ability to describe data presentations at the model-level, which reduces UI design efforts. This idea applied to models is platform-independent. On the other hand there are limits to the approach. It operates at compile time, and it does not provide generally accepted mechanism to address cross-cutting concerns. The provided benefits are not directly applicable to code-based development.

Throughout the research of this thesis is discovered that a model similar to the initial MDD approach can be received by applying metaprogramming to code-based systems. To preserve the above benefits and apply them to code-base development our approach must go beyond MDD. The GPL code can act as the source of information, and the model can be derived through metaprogramming. Next, we consider operability at runtime to avoid possible issues with exponential state growth apparent with the compile-time approaches. To effectively address tangled code, cross-cutting concerns must be addressed. Furthermore, to avoid reinventing the wheel, existing development standards must be taken into account.

From the existing approaches that address concern-separation we selected AOP as it has well accepted terminology and is applicable at runtime. The metaprogramming provides mechanisms to receive information from the code, specifically from data structures and derive join point representation, that represent the model. It consists of data structure, field information, their constraints and the result can be further extended with the runtime context. The UI derivation process is then similar to MDD, but the transformation rules and templates are extend with AOP mechanisms, pointcuts and advices. This provides versatile and generic mechanism to integrate UI concerns as well as third party mechanisms (e.g. security). The benefits brought by such transformation that at runtime derives the UI, while considering the context, is its ability to provide personalized UI experience. Even applications with simplistic UIs receive benefits, since concerns reuse throughout the time, across different data. Thus the amount of application data no longer indicate efforts for UI development and maintenance. The concern specification has increased cohesion, which reduces development and maintenance efforts and prevent inconsistency error, since information are maintained at a single place rather than distributed throughout the application.

Integration of a novel library with UI widgets to an existing application no longer requires large, tedious and error-prone maintenance efforts, since only limited amount of templates must update to reflect the changes in the entire system. Similar experience we present a large productions-level application where only 25 updates would be necessary. From another perspective the discipline of HCI provides large amount of approaches to describe CaUIs, although they mainly consider the usability aspects and not the integration in to a complete application; thus the global perspective is missing. We showed that with READ approach it is possible to integrate such third-party libraries, while providing benefits of reduced development and maintenance efforts, etc. As a conse-

quence, if an expert with advanced knowledge of a particular CaUI approach defines a set of presentation templates for READ then it is fairly easy for other developers to apply the approach to conventional application not even having a deep knowledge to the particular DSL or approach because the required approach description is generated by READ. This simplifies the transition of various CaUI approaches across different systems.

The last part of presented research considered the UI delivery from the perspective of separation of concerns. We identified that client-readable UI descriptions tangle concerns together and that their separation may provide multiple benefits. It may reduce server-side efforts related to UI processing and assembly, since the separated-concern delivery partially delegates the UI assembly to the client-side. Next, the volume of information provided by the server to clients reduces, and makes it possible for clients to reuse concerns as well as to apply cache. This reduces the UI page load times and improves UI responsiveness. Furthermore, it opens the server-side for further reuse since concerns are provided in a machine-readable format. In a case study we presented not only the UI page load time improvements but also show platform-specific client-based application prototypes that reuse platform-independent information provided by the server. This may, for instance, simplify the development and maintenance efforts of mobile applications, that supplement web-system, while providing improved usability to users.

The presented approaches were demonstrated on multiple case studies that considered the impact on development and maintenance efforts, reduced dual decisions, information restatements, code volume, as well as providing the performance evaluation. The outcome of this work is not only theoretical, a library implementing READ is provided and deployed in a production-level application for the ACM-ICPC contest management. The application serves to users from more than 2,534 universities and 101 countries worldwide.

References A

Bibliography

[1] Hans Bergsten. *JavaServer Faces*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[2] Ed Burns and Neil Griffin. *JavaServer Faces 2.0, The Complete Reference*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming. In *In ECOOP'97-Object-Oriented Programming, 11th European Conference*, volume 1241, pages 220–242. Springer, June 1997.

[5] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., October 1976.

[6] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[7] Max Schlee and Jean 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.

[8] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[9] Linda DeMichiel. JSR 317: JavaTM persistence API, version 2.0, November 2009.

[10] Emmanuel Bernard. JSR 303: Bean validation, November 2009.

[11] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[12] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and generative programming (invited paper). In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-7, pages 2–19, London, UK, UK, 1999. Springer-Verlag.

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

[14] Miroslav Macik, Martin Klima, and Pavel 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.

[15] Anthony Finkelstein and Jeff Kramer. Software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 3–22, New York, NY, USA, 2000. ACM.

[16] Linda DeMichiel and Michael Keith. JSR 220: Enterprise javabeans version 3.0. java persistence API, May 2006.

[17] Jean-Sébastien Sottet, Gaëlle Calvary, Joëlle Coutaz, and Jean-Marie Favre. A model-driven engineering approach for the usability of plastic user interfaces. In *Engineering Interactive Systems*, pages 140–157. Springer, 2008.

[18] Maximilian Stoerzer and Stefan 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.

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

[20] Dan Allen. *Seam in Action*. Manning Publications Co., Greenwich, CT, USA, 2008.

[21] Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 195–202, New York, NY, USA, 1992. ACM.

[22] Anthony Finkelstein and Jeff Kramer. Software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 3–22, New York, NY, USA, 2000. ACM.

[23] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[24] Edsger W. Dijkstra. The structure of the “the”-multiprogramming system. *Commun. ACM*, 11(5):341–346, May 1968.

[25] David Garlan and Mary Shaw. An introduction to software architecture. *Advances in software engineering and knowledge engineering*, 1:1–40, 1993.

[26] J.-S. Lee and H.S. Chae. Domain-specific language approach to modelling ui architecture of mobile telephony systems. *Software, IEE Proceedings* -, 153(6):231–240, 2006.

[27] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[28] Richard Kennard, Ernest Edmonds, and John Leaney. Separation anxiety: stresses of developing a modern day separable user interface. In *Proceedings of the 2nd conference on Human System Interactions*, HSI'09, pages 225–232, Piscataway, NJ, USA, 2009. IEEE Press.

[29] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[30] Debu Panda, Reza Rahman, and Derek Lane. *EJB 3 in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

[31] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[32] Hibernate validator, open-source validation library for hibernate framework, 2012.

[33] Michal Bali. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing, 2009.

[34] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligences*, 19(1):17–37, 1982.

[35] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[36] OMG, Unifiend Modeling Language (UML) version 2.1.2, Model Driven Architecture (MDA), Object Constraint Language (OCL) version 2.0, 2010.

http://www.omg.org.

[37] Nathalie Moreno, José R. Romero, and Antonio Vallecillo. An overview of model-driven web engineering and the mda. In *Human-Computer Interaction Series*, pages 353–382. Springer London, 2008.

[38] Sven Kloppenburg Vasian Cepa. Representing explicit attributes in UML. In *7th International Workshop on Aspect-Oriented Modeling*, 2005.

[39] OMG, Unifiend Modeling Language (OMG UML), infrastructure (version 2.2), February 28 2009.

http://www.omg.org/cgi-bin/doc?formal/09-02-04.

[40] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, October 2009.

[41] Jean-Sébastien Sottet, Gaëlle Calvary, and Jean-Marie Favre. Models at runtime for sustaining user interface plasticity. In *Models@ run. time workshop (in conjunction with MoDELS/UML 2006 conference)*, 2006.

[42] David Lorge Parnas. Software aspects of strategic defense systems. *Commun. ACM*, 28(12):1326–1335, December 1985.

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

[44] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[45] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[46] Shigeru Chiba. Javassist – a reflection-based programming wizard for Java. In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[47] Swing GUI Builder, November 2013.

[48] Windowbuilder, November 2013.

[49] RedView: Riena EMF dynamic views for business applications, 2010. http://redview.wordpress.com.

[50] S Salah and Hyontai Sug. The effectiveness of rapid business application development using oracle forms. In *Advanced Information Management and Service (ICIPM), 2011 7th International Conference on*, pages 33–37. IEEE, 2011.

[51] Jorge-luis Perez-medina, Sophie Dupuy-chessa, and Agnes Front. A survey of model driven engineering tools for user interface design. In *In Proc. of 6th Int. workshop on Task Models and Diagrams (TAMODIA'2007)*, pages 84–97, Berlin, 7-9 Nov. 2007. Springer.

[52] Jos Warmer and Anneke Kleppe. *The object constraint language: precise modeling with UML.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[53] Jacob W Jespersen and Jesper Linvald. Investigating user interface engineering in the model driven architecture. In *In Proceedings of the Interact 2003 Workshop on Software Engineering and HCI. IFIP.* Press, 2003.

[54] Jeffrey Nichols and Andrew Faulring. Automatic interface generation and future user interface tools. In *Tools ACM CHI 2005 Workshop on The Future of User Interface Design Tools*, 5000 Forbes Ave, Pittsburgh, PA 15213 USA, 2005.

[55] Alexandre Torres, Renata Galante, and Marcelo Soares Pimenta. Towards a uml profile for model-driven object-relational mapping. In *SBES '09: Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering*, pages 94–103, Washington, DC, USA, 2009. IEEE Computer Society.

[56] Kris Luyten, Chris Vandervelpen, Jan Van den Bergh, and Karin Coninx. Context-sensitive user interfaces for ambient intelligent environments: Design, development and deployment.

[57] Víctor López-Jaquero, Francisco Montero, and Fernando 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.

[58] Mart Karu. A textual domain specific language for user interface modelling. In Tarek Sobh and Khaled 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.

[59] Richard Kennard and John Leaney. Is there convergence in the field of ui generation? *J. Syst. Softw.*, 84(12):2079–2087, December 2011.

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

[61] Arnaud Blouin, Brice Morin, Olivier Beaudoux, Grégory Nain, Patrick Albers, and Jean-Marc 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.

[62] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. USIXML: A Language Supporting Multi-path Development of User Interfaces Engineering Human Computer Interaction and Interactive Systems. In Rémi Bastide, Philippe Palanque, and Jörg Roth, editors, *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.

[63] 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.

[64] Veit Schwartze, Sebastian Feuerstack, and Sahin 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.

[65] Giulio Mori, Fabio Paterno, and Carmen Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Trans. Softw. Eng.*, 30(8):507–520, August 2004.

[66] Marco Blumendorf, Grzegorz Lehmann, and Sahin 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.

[67] Tim Clerckx, Kris Luyten, and Karin Coninx. The mapping problem back and forth: customizing dynamic models while preserving consistency. In *Proceedings of the 3rd annual conference on Task models and diagrams*, pages 33–42. ACM, 2004.

[68] S. Berti, F. Correani, G. Mori, F. Paternò, and C. Santoro. Teresa: a transformation-based environment for designing and developing multi-device interfaces. In *CHI'04 extended abstracts on Human factors in computing systems*, pages 793–794. ACM, 2004.

[69] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

[70] KZ Gajos, DS Weld, and JO Wobbrock. Automatically generating personalized user interfaces with supple. *Artificial Intelligence*, 174(12-13):910–950, August 2010.

[71] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The akamai network: A platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, August 2010. http://dx.doi.org/10.1145/1842733.1842736.

[72] Robert Hanson and Adam Tacy. *GWT in Action: Easy Ajax with the Google Web Toolkit*. Manning Publications Co., Greenwich, CT, USA, 2007.

[73] AngularJS documentation, April 2015.

[74] Bin Swen. Outline of initial design of the structured hypertext transfer protocol. *J. Comput. Sci. Technol.*, 18(3):287–298, 2003. http://dx.doi.org/10.1007/BF02948898.

[75] Robert L. R. Mattson and Somnath Ghosh. HTTP-MPLEX: An enhanced hypertext transfer protocol and its performance evaluation. *J. Netw. Comput. Appl.*, 32(4):925–939, 2009. http://dx.doi.org/10.1016/j.jnca.2008.10.001.

[76] David F. Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-Based Access Control*. Artech House, Inc., Norwood, MA, USA, 2003.

[77] Alexandre Torres, Renata Galante, and Marcelo Soares Pimenta. Towards a uml profile for model-driven object-relational mapping. In *Proceedings of the 2009 XXIII Brazilian Symposium on Software Engineering*, SBES '09, pages 94–103, Washington, DC, USA, 2009. IEEE Computer Society.

[78] Barry W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, Ray Madachy, and Bert Steece. *Software Cost Estimation with Cocomo II with Cdrom*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[79] Jung-Min Oh, Yong Sub Lee, and Nammee 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.

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

[81] J.O. Wobbrock, S.K. Kane, K.Z. Gajos, S. Harada, and J. Froehlich. Ability-based design: Concept, principles and examples. *ACM Transactions on Accessible Computing (TACCESS)*, 3(3):9, 2011.

[82] Jakob Nielsen. Usability engineering. *Boston: AP Professional,/ c1993*, 1, 1993.

[83] Anthony M Graziano and Michael L Raulin. *Research methods: A process of inquiry*. HarperCollins College Publishers, 1993.

References B

Refereed publications

Journals with Impact Factor:

[A.1] Tomas Cerny, Miroslav Macik, Jeff Donahoo and Jan Janousek. On Distributed Concern Delivery in User Interface Design. *Computer Science and Information Systems (ComSIS) Journal*, accepted for publication 2015. (WOS) [50/15/25/10]

[A.2] Tomas Cerny and Eunjee Song. Model-driven rich form generation. *INFORMATION-An International Interdisciplinary Journal*, 15(7, SI):2695–2714, JUL 2012. (WOS)

[A.3] Miroslav Macik, Tomas Cerny, and Pavel Slavik. Context-sensitive, cross-platform user interface generation. *Journal on Multimodal User Interfaces*, pages 1–13. Springer Berlin Heidelberg, 2014. (WOS) [50/40/10]

Journals without Impact Factor:

[A.4] Tomas Cerny, Karel Cemus, Michael J. Donahoo, and Eunjee Song. Aspect-driven, data-reflective and context-aware user interfaces design. *Applied Computing Review*, 13(4):53–65, 2013. (ACM DL)

[A.5] Karel Cemus, Tomas Cerny and Michael J. Donahoo. Automated Business Rules Transformation into a Persistence Layer. *(To appear) Procedia Computer Science Journal*, Elsevier, 2015. (Scopus) [90/5/5]

Conference papers in Institute for Scientific Information (ISI):

[A.6] Tomas Cerny, Lubos Matl, Karel Cemus and Michael J.Donahoo. Evaluation of Separated Concerns in Web-based Delivery of User Interfaces. *Information Science and Applications, LNEE*, Springer, 2015. (WOS)

[A.7] Tomas Cerny and Michael J.Donahoo. Separating out Platform-independent Particles of User Interfaces. *Information Science and Applications, LNEE*, Springer, 2015. (WOS)

[A.8] Tomas Cerny, Vaclav Chalupa, Lukas Rychtecky, and Tomas Linhart. Machine-driven code inspection to reduce restated information. In *Lecture Notes in Information Technology*, 2012. (WOS)

[A.9] Tomas Cerny, Petr Praus, Slavka Jaromeska, Lubos Matl and Michael J. Donahoo. Towards a Smart, Self-scaling Cooperative Web Cache. In *SOFSEM 2012: Theory and Practice of Computer Science, LNCS 8327*, pages 443–455. Springer International Publishing 2012, 2012. (WOS)

[A.10] Karel Cemus and Tomas Cerny. Aspect-driven design of information systems. In *SOFSEM 2014: Theory and Practice of Computer Science, LNCS 8327*, volume 8327, pages 174–186. Springer International Publishing Switzerland 2014, 2014. (WOS) [75/25]

Other conference papers:

[A.11] Tomas Cerny and Eunjee Song. A profile approach to using uml models for rich form generation. In *Information Science and Applications (ICISA), 2010 International Conference on*, pages 1–8, 2010. (Scopus)

[A.12] Tomas Cerny Miroslav Macik, Michael J. Donahoo. and Jan Janousek. Efficient Description and Cache Performance in Aspect-Oriented User Interface Design. In *Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, ACSIS*, volume 2, pages 1667–1676. IEEE Computer Society Press and Polish Information Processing Society, 2014. (Scopus)

[A.13] Tomas Cerny and Eunjee Song. Uml-based enhanced rich form generation. In *Proceedings of the 2011 Research in Applied Computation Symposium (RACS 2011)*, pages 192–199, November 2011. (Scopus)

[A.14] Tomas Cerny and Michael J. Donahoo. Formbuilder: A novel approach to deal with view development and maintenance. In *In SofSem 2011 Proceedings of Student Research Forum*, pages 16–34. OKAT, January 2011. (Google Scholar)

[A.15] Tomas Cerny, Vaclav Chalupa, and Michael J. Donahoo. Towards smart user interface design. In *Information Science and Applications (ICISA), 2012 International Conference on*, pages 1–6, may 2012. (Scopus)

[A.16] Tomas Cerny, Vaclav Chalupa, and Michael J. Donahoo. Impact of user interface generation on maintenance. In *Computer Science and Automation Engineering (CSAE)*, volume 2, pages 621–625. IEEE, 2012. (Scopus)

[A.17] Tomas Cerny, Michael J. Donahoo, and Eunjee Song. Towards effective adaptive user interfaces design. In *Proceedings of the 2013 Research in Applied Computation Symposium (RACS 2013)*, October 2013. (Scopus)

[A.18] Tomas Cerny and Michael J. Donahoo. How to reduce costs of business logic maintenance. In *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, volume 1, pages 77–82, june 2011. (Scopus)

[A.19] Tomas Cerny, Petr Praus, Slavek Jaromeska, Lubos Matl and Michael J. Donahoo. Cooperative web cache. In *Systems, Signals and Image Processing (IWSSIP), 2011 18th International Conference on*, pages 1–4, IEEE, 2011. (Scopus)

[A.20] Tomas Cerny and Micheal J. Donahoo. Performance Optimization for Enterprise Web Applications Through Remote Client Simulation. In *Proc. of the 7th EUROSIM Congress on Modelling and Simulation, Prague, CZ*, volume 2, CTU, Prague, 2010. (Google Scholar)

[A.21] Miroslav Macik, Tomas Cerny, Jindrich Basek, and Pavel 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. [55/25/10/10] (Scopus)

[A.22] Martin Tomasek and Tomas Cerny. Automated User Interface Derivation for Remote Data in Standalone Apps. In *Proceedings of the 19th International Scientific Student Conferenece POSTER 2015*, Prague, 14, May 2015, Czech Technical University in Prague. (Google Scholar)

[A.23] Karel Cemus and Tomas 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. [95/5] (Google Scholar)

() - Citation database/indexing : Web of Science | Scopus | ACM Digital lib. | Google Scholar
[] - Explicit authorship ratio by author : even by default

References C

Unrefereed publications

Journals without Impact Factor:

[A.24] Tomas Cerny and Bozena Mannova. Competitive and Collaborative Approach Towards a More Effective Education in Computer Science. *CONTEMPORARY EDUCATIONAL TECHNOLOGY.*, 2(2):163–173, 2011. (Google Scholar)

Conference papers in Institute for Scientific Information (ISI):

[A.25] Petr Praus, Slavka Jaromerska and Tomas Cerny. SScAC: towards a framework for small-scale software architectures comparison. *SOFSEM 2011: Theory and Practice of Computer Science.*, pages 482-493, Springer, 2011. (WOS)

Other conference papers:

[A.26] Tomas Cerny and Michael J. Donahoo. MetaMorPic: Self-contained photo archival and presentation. *Information Systems Development.*, 149–158, Springer New York, 2011. (Scopus)

[A.27] Lubos Matl, Vladimir Kloucek, Viktor Bohdal, Jan Kubr and Tomas Cerny. ELISA: Extensible Layer for Internet Services and Applications. *Building Sustainable Information Systems.*, 309–321, Springer, 2013. (Scopus)

[A.28] Tomas Cerny and Bozena Mannova. Debt Environment in Computer Science Education. *In he 3rd International Multi-Conference on Complexity, Informatics and Cybernetics: IMCIC 2012.*, 1:396–401, 2011. (Google Scholar)

[A.29] Tomas Cerny and Bozena Mannova. Competitive and Collaborative Approach Towards a More Effective Education in Computer Science. *In: The 9th Annual Hawaii International Conference on Education.*, pages 2886–2895, 2011. (Google Scholar)

[A.30] Tomas Cerny and Michael J. Donahoo. A Tool for Evaluation and Optimization of Web Application Performance. *In Proceedings of 44th Spring International Conference MOSIS'X.*, pages 49–54, 2010. (Google Scholar)

[A.31] Tomas Cerny and Michael J. Donahoo. Evaluation and Optimization of Web Application Performance Under Varying Network Conditions. *In Proceedings*

of 44th Spring International Conference MOSIS'X., pages 41–48, 2010. (Google Scholar)

[A.32] Lubos Matl and Tomas Cerny. ELISA: Extensible Layer for Internet Services and Applications. *Proceedings of the 17th International Scientific Student Conferenece POSTER 2013.* , 2013. (Google Scholar)

[A.33] Lubos Matl, Tomas Cerny and Michael J.Donahoo. Effective manycast messaging for Kademlia network *(To appear) In Proceedings of 30th ACM/SIGAPP Symposium On Applied Computing*, 2015, (Google Scholar)

References D

Citations

Refereed publications:

[A.34] Saad Masood Butt, Mazlina Abdul Majid, Suziyanti Marjudi, Shahid Masood Butt, Azura Onn, Moaz Masood Butt. CASI METHOD FOR IMPROVING THE USABILITY OF IDS. *Sci.Int.(Lahore)*. SCIENCE INTERNATIONAL-(Lahore), pages 275-286, 2015. [A.3]

[A.35] P. Biswas, PM Langdon, J Umadikar, S Kittusami and S Prashant. How interface adaptation for physical impairment can help able bodied users in situational impairment. *Inclusive Designing.* Springer International Publishing, pages 49-58, 2014. [A.21]

[A.36] Joaquın Canadas, José Palma, and Samuel Túnez. Model-Driven Rich User Interface Generation from Ontologies for Data-Intensive Web Applications. *In Proceedings of 7th Workshop on Knowledge Engineering and Software Engineering (KESE7).*, 2011. [A.11]

[A.37] K. Santhi, G. Zayaraz and V. Vijayalakshmi. Resolving Aspect Dependencies for Composition of Aspects. *Arabian Journal for Science and Engineering.* Springer Berlin Heidelberg, pages 1-12, 2014. [A.10]

[A.38] Pradip Peter Dey, Bhaskar Raj Sinha, Gordon W. Romney, Mohammad Amin, and Hassan Badkoobehi. Innovative User Interface Engineering. *In International conference on Innovative Engineering Technologies (ICIET'2014).*, pages 1-11, 2014. [A.4]

[A.39] Milorad Filipovid, Sebastijan Kaplar, Renata Vaderna, Željko Ivkovid, Gordana Milosavljevic and Igor Dejanovid. Aspect-Oriented Engines for Kroki Models Execution. *5th International Conference on Information Society and Technology (ICIST 2015).*, pages 1-6, 2015. [A.12] [A.4]

Unrefereed publications:

[A.40] Gulley, O. David, and Aaron L. Jackson. Teaching a Class Dedicated to the College Fed Challenge Competition. *Eastern Economic Journal.*, Nature Publishing Group, 2015. [A.24]

[A.41] Wicaksono, Soetam Rizky. Implementation of Collaborative Learning in Higher Education Environment. *Journal of Education and Learning.*, 7(4):219–222, 2013. [A.24]

[A.42] Armas, Audrius and Šniras, Šarūnas. Interdependence-based model consistency among competition, cooperation and collaboration. *Žurnalas „Ugdymas. Kūno kultūra. Sportas "leidžiamas nuo 1968 m.(ankstesnis pavadinimas–mokslo darbai „Kūno kultūra ").*, 2013. [A.24]

[A.43] Armas, Audrius. Tarpusavio priklausomybes prielaida pagrįsto konkuravimo, kooperavimo ir bendradarbiavimo derinimo modelio taikymas sporte. *Sportinį darbingumą lemiantys veiksniai (V).*, 2012. [A.24]

Appendix A

List of abbreviations

ADT Abstract Data Type 24
ADPP Annotation Driver Participant Pattern 88
AJAX Asynchronous JavaScript and XML . . . 94
AOP Aspect-Oriented Programming 11
AUI Abstract User Interface 104
AWT Abstract Window Toolkit 33
CaUI Context-aware User Interface 12
CCT Concur Task Tree 51
CBD Component-Based Design 25
CDN Content-delivery Network 53
CDN Content-delivery Networks 53
COCOMO Constructive Cost Model 75
CPU Central processing unit 54
CRUD Create-Read-Update-Delete 95
CSS Cascading Style Sheets 112
CUI Concrete User Interface 104
CWC Cooperative-Web cache 54
DSL Domain Specific Languages 3
EA Enterprise Application 9
EAA Enterprise Application Architecture . . . 25
EL Expression Language 86
GP Generative Programming 11
GPL General-Purpose Languages 3
GWT Google Web Toolkit 34
HCI Human-Computer Interaction 11
HTML Hypertext Markup Language 13
HTTP Hypertext Transfer Protocol 25
ACM-ICPC ACM International Collegiate Programming Contest iv
IDE Integrated Development Environment . 46
IP Internet Protocol 25
Java SE Java Standard Edition 2
Java EE Java Enterprise Edition 2
JPA Java Persistence API 28
JS JavaScript 34
JSF JavaServer Faces 2
JSON JavaScript Object Notation 53
JSP JavaServer Pages 48
LOC Lines Of Code 41
MD Model-Driven 47
MDA Model-Driven Architecture 35
MDD Model-Driven Development iii
MP Meta-Programming 35
MVC Model-view-controller 32
OCL Object-Constraint Language 46
OO Object-Oriented 20
OOD Object-Oriented Design 23
OOP Object-Oriented Programming 24
ORM Object-Relational Mapping 28
P2P Peer-To-Peer 54
PC Personal Computer 105
RBAC Role-Based Access Control 67
READ Rich Entity Aspect/Audit Design 60
STTP Structured Hypertext Transfer Protocol 54
TCP Transmission-control protocol 25
UI User Interface iii
UIV User Input Validation 47
UIP User Interface Protocol 60
UML Unified Modeling Language 14
UIML User Interface Markup Language 130
XHTML Extensible Hypertext Markup Language 99
XML Extensible Markup Language 20
XSL Extensible Stylesheet Language 45
XSLT Extensible Stylesheet Language Transformations 45
XUL XML User Interface Language 130

Appendix B

Český abstrakt

Uživatelská rozhraní (anglicky UI) softwarových systémů hrají zásadní roli při jejich použitelnosti. Uživatelé stále rozšiřují nároky a osobní očekávání pro to co je "dobré" UI, jako například, dynamické přizpůsobení se různým vstupům a podnětům od uživatele. Bohužel, vývojové frameworky typicky adresují vývoj i návrh UI jako zcela nezávislý útvar, ignorující globální perspektivu nad celou aplikací. Toto má za následek replikaci informací pocházejících z definic aplikačních dat, které jsou znovu zachyceny v popisu uživatelských rozhraní (tj. vertikální opakování). Specifikace očekávaného typu, ověřování vstupů na straně klienta, a dokonce i výběr vstupního widgetu představují opakující se zachycení informace, které musí být udržováno během evoluce dané aplikace. Toto je navíc často ztíženo nižší jazykovou podporou detekce možných chyb, a to především díky častému použití jazyků pro popis UI s nízkou typovou bezpečností. Kromě toho, vlastní popis UI vede ke křížení velkého množství aplikačních zájmů, jako je rozložení prvků na obrazovce, svázání elementů UI a dat, validace vstupů, bezpečnost, atd. Toto pak ve výsledků zachycuje jednotlivé zájmy prolínajíce jiné zájmy a znemožňuje tak jejich znovupoužití v jiných komponentách UI či pro jiné kontextové situace téhož UI. Takovéto křížení zájmů v popisu UI znamená, že konkrétní definice zájmu není definována na jednom místě, nýbrž je distribuována a opakuje se v mnoha různých fragmentech UI. Změna takového zájmu potom vyžaduje značné úsilí. Celá situace se dále zhorší pokud se od aplikací očekává, že se dokáží přizpůsobit pro jednotlivé uživatele (např expertní uživatel, začínající, atd.), kontexty (např pro jednotlivé země původu, poštovní informace, apod.), a dokonce i fyzické prezentace (např., na desktopu vs. mobilním zažízení). Současná řešení vývoje a návrhu UI neposkytují takovou flexibilitu popisu UI, aby bylo možné snadno manipulovat se zájmy a poskytnout tak variace UI pro podporu personalizace. Stejně tak současná řešení míchající různé zájmy mají za následek vznik mnoha podobných popisů UI, které opakují informace o typech vstupů, omezení, apod., a liší se pouze v detailech (tj., horizontální opakování). Je zřejmé, že v takovéto situaci je úprava daného zájmu UI velmi vyčerpávající a náročná. Vysoká úroveň replikace, znovuzachycení informací i rozhodnutí zvyšuje náchylnost k chybám i nestabilitu vůčí změnám v aplikaci. Negativní dopad

takovéhoto opakování a křížících se zájmů v popisu UI směřuje k vysokým nákladům na vývoj a údržbu. Popisy UI obsahující opakující se informace vyžadují více zpracování na straně serveru, stejně tak i větší šířku přenosového pásma pro doručení klientům, což negativně ovlivňuje odezvu UI i škálovatelnost serveru. Informace popisující dané UI, které obdrží klient, opět obsahují křížící se zájmy, což znemožní ukládat informace o jednotlivých zájmech do vyrovnávací paměti a opětovně je použit na straně klienta.

Tato práce zkoumá existující přístupy návrhu UI z perspektivy vývoje, údržby, rychlosti odezvy na akci uživatele, integrace se stávajícími podnikovými frameworky, schopnost podporovat personifikaci odrážející kontext uživate i systému. Práce identifikuje, že konvenční přístupy řeší výše uvedené problémy pro prezentace dat v UI jen částečně nebo neefektivně a navrhuje nový přístup oddělující křížící se zájmy. Práce dále zkoumá existující alternativní přístupy návrhu, jako je modelem-řízený vývoj, a jejich mechanismy snížující replikace a opakující se informace v popisu UI z pohledu samotného vývoje i přenosu dat klientům. Navržený přístup snižuje komplexitu týkající se vývoje a údržby UI a poskytuje snadné rozšíření schopností UI i sledovaných zájmů, a podporuje tak kontextovou-uvědomělost UI a personifikace. Vlastní oddělení zájmů lze aplikovat na přenos popisu UI klientům, čímž dojde například ke zlepšení rychlosti reakce na uživatelské akce, zvýšení znovupoužití informací či zájmů na klientské straně, či zlepšení výkonu při zpracování UI na straně serveru. Práce demonstruje efektivitu navrhovaného přístupu prostřednictvím případových studií. Mimo jiné, práce vyhodnocuje nasazení navrženého přístupu v produkčním prostředí podnikové aplikace ACM-ICPC, kterou používají desítky tisíc účastníků z více než 2534 univerzit, ze 101 zemí světa, pro registrace na soutěže v programování.

Klíčová slova:

Uživatelská rozhraní, Metaprogramování, Aspektově orientovaný vývoj, Modelem řízený vývoj, Transformace modelu do zdrojového kódu, Údržba, Oddělení zájmů