Computer Science and Information Systems 12(2):655–681 DOI: 10.2298/CSIS141202021C

On Distributed Concern Delivery in User Interface Design

Tomas Cerny[1], Miroslav Macik[2], Michael J. Donahoo[3], and Jan Janousek[4]

[1] Computer Science, FEE, at Czech Technical University,
Charles square 13, Prague 2, 121 35, Czech Republic
tomas.cerny@fel.cvut.cz
[2] Graphics and Interaction, FEE, at Czech Technical University,
Charles square 13, Prague 2, 121 35, Czech Republic
macikmir@fel.cvut.cz
[3] Computer Science at Baylor University, Waco, TX
One Bear Place #97356, 76798-7356, USA
jeff_donahoo@baylor.edu
[4] Theoretical Computer Science, FIT, at Czech Technical University
Thakurova 9, Prague 6, 160 00, Czech Republic
jan.janousek@fit.cvut.cz

Abstract. Increasing demands on user interface (UI) usability, adaptability, and dynamic behavior drives ever-growing development and maintenance complexity. Traditional UI design techniques result in complex descriptions for data presentations with significant information restatement. In addition, multiple concerns in UI development leads to descriptions that exhibit concern tangling, which results in high fragment replication. Concern-separating approaches address these issues; however, they fail to maintain the separation of concerns for execution tasks like rendering or UI delivery to clients. During the rendering process at the server side, the separation collapses into entangled concerns that are provided to clients. Such client-side entanglement may seem inconsequential since the clients are simply displaying what is sent to them; however, such entanglement compromises client performance as it results in problems such as replication, fragment granularity ill-suited for effective caching, etc.

This paper considers advantages brought by concern-separation from both perspectives. It proposes extension to the aspect-oriented UI design with distributed concern delivery (DCD) for client-server applications. Such an extension lessens the server-side involvement in UI assembly and reduces the fragment replication in provided UI descriptions. The server provides clients with individual UI concerns, and they become partially responsible for the UI assembly. This change increases client-side concern reuse and extends caching opportunities, reducing the volume of transmitted information between client and server to improve UI responsiveness and performance. The underlying aspect-oriented UI design automates the server-side derivation of concerns related to data presentations adapted to runtime context, security, conditions, etc. Evaluation of the approach is considered in a case study applying DCD to an existing, production web application. Our results demonstrate decreased volumes of UI descriptions assembled by the server-side and extended client-side caching abilities, reducing required data/fragment transmission, which improves UI responsiveness. Furthermore, we evaluate the potential benefits of DCD integration implications in selected UI frameworks.

Keywords: user interface, evaluation study, separation of concerns, responsiveness.

1. Introduction

Conventional UI design approaches usually provide a wide range of mechanisms to describe a particular UI: a large collection of composable components, widgets, component control mechanisms, etc. Such approaches typically focus solely on the UI portion of the application. Unfortunately, this isolation of UI description from the application often results in application information restatement especially in the case of data definitions. [4, 18]. Thus existing data definitions along with their field constraints, input validation rules, and even security are replicated by the UI description. This correlation between the UI and other elements must be manually maintained. Besides that, various context-related situations may demand adjustments to the UI descriptions, which due to limited description mechanisms lead into duplicated descriptions [4]. Conventional approaches become inefficient when it comes to support of UI variations related to context-awareness [21], usually resulting in multiple, highly similar UI descriptions.

Recent work applying Aspect-Oriented Programming (AOP) to UI design [4] identifies multiple deficiencies in conventional UI design approaches and provides a mechanism to address them. First, it provides mechanisms to reduce information restatement by utilizing metaprogramming and code-inspection [18] that inspect existing data elements and incorporates the result into UI descriptions. Next, it identifies that conventional approaches mix different sorts of information (concerns) together in particular UI descriptions. For instance, field presentation, layout, and security are all tangled together in the description. Consequently, this limits reuse of particular concerns, which particularly hinders support of variations found in context-aware UIs. To address this, AOP UI alters UI description. Instead of describing it all at once, individual concerns are described separately and integrated upon request at runtime. This allows considering particular runtime context, which influences concern selection and thus variations. Such mechanisms reduce human-errors related to typological errors resulting from restatement by using automated enforcement of the correlation between separately defined the data elements and their UI presentation. Next, the supported variability and reuse of concerns allows design of context-aware UIs with low development and maintenance efforts [21]. For instance, [4] reports 30% reduction of UI code volume in a case study applying AOP UI when compared to conventional UI.

Although AOP UI design brings multiple benefits to the development and maintenance, it does not speak to the UI delivery to clients. In particular, the concerns separation gets lost at the server-side upon the delivery to clients, who then cannot take any benefits that would result from the separation. We consider the maintenance of concern separation from both client and server perspectives. We extend the AOP UI design with the ability to preserve the separation on both sides and consider the impact of such distributed concerns delivery (DCD). Specifically, we consider the volume of transmitted information, the volume of information processed by server, opportunities for concurrent delivery, concern reuse and caching capabilities, as well as the impact on UI responsiveness and performance. Additionally, we consider DCD integration to selected UI frameworks and evaluate the potential benefits. We evaluate the impact of DCD on a case study on an existing production web system and compare it with a conventional UI approach. Our results show extended caching capabilities, reduced transmission size, improved UI load times and decreased amount of information processed by server.

The remainder of this paper is organized as follows: Section 2 describes the background of the UI designs and gives basic notions. Section 3 provides an overview of existing work. Our proposed approach is presented in Section 4. A case study is discussed in Section 5. DCD integration to selected UI frameworks is considered in Section 6. Conclusion and future work close the paper.

2. Background

The UI is one of the most critical parts of an application; it plays the role of the ambassador of an application, as it is the portion of the application experienced by users. It provides mechanisms to control the application, submit and retrieve data as well as influence the presentation of information. The UI graphical design aims to make user interaction simple and efficient for given tasks. On the other side, designers must balance the demand for usability with the cost of maintainability and the impact on speed. Typically, improving usability increases application complexity, driving up maintenance costs, and reducing performance. The scale of UI development efforts are apparent from [4, 17] showing that approximately 48% of application code and 50% of development time is devoted to implementing UIs, which only increases with expanding the UI abilities and context-awareness.

Existing enterprise applications involve large volumes of data [13], tend to be web-based and provide many different UIs for particular users with different levels of technical expertise, goals and various purposes. A single piece of data might be presented in many different ways, and although the data definition is the same, its particular presentation tends to have a specific UI description. For instance, consider one particular presentation given by the UI description of person data in Listing 1.1 using JavaServer Faces (JSF) framework [3] with HTML. Such a description references a person data object from a co-existing definition in Listing 1.2, defines a particular layout, and uses specific widgets for each data field. In addition, it can use conditional rendering as well as enforce validation. Such a description is usually resolved to HTML and provided to clients. In order to provide slight variations of the presentation, such as changes in the layout, reordered fields, less restrictive input validation enforcement, etc., it might be necessary to design multiple such UI descriptions. More variations in the UI presentation mean, more UI descriptions, which leads to extended development and maintenance efforts.

The conventional UI design approaches possess multiple deficiencies. For example, UI descriptions for data presentations typically restate information [17] from lower layers, risking mistype errors that lead to inconsistencies. The complexity is most evident when the UI description uses Domain Specific Languages (DSLs) [24] with limited type safety [4]. Our example in Listing 1.1 uses a DSL provided by JSF. For instance, consider the data property bindings indicated by the green color. Clearly, there is limited type safety when mistyping maximum length validation restriction (*maxlength*), not empty enforcement (*required*), and even the data field value binding. Modification to the co-existing person object data definition in Listing 1.2 requires manual changes to the UI description in Listing 1.1. Such maintenance is error prone, and with such weak type safety, there is no mechanism for warning about failures to maintain perfect synchronization. Moreover, if multiple variants of the person UI description exist, we must apply the changes to multiple locations.

Listing 1.1. Sample conventional UI description of a 3-field form based on JSF.

```
<table><tr>
 <td><h:outputLabel value="Email:"/>
     <h:input id="email" value="#{person.email}" required="true" maxlength="50"
     render="#{security.hasAccess('email')}" validate="#{v.validate('email')}"/></td>
</tr><tr>
 <td><h:outputLabel value="Name:"/>
     <h:input id="name" value="#{person.name}" maxlength="100"
     required="true"/></td> </tr><tr>
 <td><h:outputLabel value="Country:"/>
 <a:smenu id="country" value="#{person.country}" required="true"/></td>
</tr></table>
```

Listing 1.2. Sample Person data object (referenced from Listing 1.1).

```
@Entity @Table(name = "person")
public class Person {
   ...
  @UiUserRoles({"Admin","Owner"}) @UiOrder(1) @NotEmpty @Email
  @Length(max=100) @Column(nullable=false, length=100)
  public String getEmail() { return email; }

  @UiOrder(3.1) @NotEmpty @Pattern(regex="^[^\\s].*")
  @Length(max=100) @Column(nullable=false, length=100)
  public String getName() { return name; }

  @UiOrder(81) @UiProfiles({"US"}) @NotEmpty @Column(nullable = false)
  public Country getCountry() { return country; }
}
```

Listing 1.3. Layout concern separated from Listing 1.1 single/double column.

```
<table>                                  |  <table>
 <tr><td>$af:email$</td></tr>            |   <tr><td>$af:email$</td><td>$af:name$</td></tr>
 <tr><td>$af:name$</td></tr>             |   <tr><td colspan="2">$af:country$</td></tr>
 <tr><td>$af:country$</td></tr>          |  </table>
</table>                                 |
```

Listing 1.4. Field presentation concern separated from Listing 1.1 representing a text field.

```
<h:outputLabel value="$Field$:"/>
<h:input id="$field$" value="#{$entity$.$field$}" required="$required$"
  minlength="$minLength$"  maxlength="$empty maxLength ? 255 : maxLength$"
  readOnly="#{empty edit$Field$ ? edit : edit$Field$}"
  pattern="$pattern$" $validationResolver(field)$/>
```

Listing 1.5. Sample transformation rules for String fields to derive UI presentation.

```
<mapping>
 <type>String</type>
 <default tag="textTemplate" size="20" 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>
```

The conventional UI design aims to describe a particular UI presentation as a self-contained fragment. In fact, this is the reason for the inefficiency when addressing UI variations or context-aware UIs. AOP emphasizes the notion of concerns. A concern is understood [19] as a set of information that influences the source code, description, or a particular component. An example of a concern [19, 20] is an optimization to perfor-

mance of a particular component, screen-size element allocation, security enforcement, etc. Good design should target separation of concerns [19, 20], which is an approach for modularization that brings a reduction of design complexity. When we have multiple concerns that tangle in the same section (consider code fragments captured by different colors in Listing 1.1), we call it highly coupled or tangled. Thus we separate concerns to support readability and maintenance.

The situation is not always straightforward as concerns that cannot be easily separated and tangle together in a given source code section. These concerns are called cross-cutting [19, 20]. These concerns cannot be cleanly separated from each other, which causes code duplication, hard reuse, and has the effect of "spaghetti code" with difficult maintenance. The reason why we cannot separate cross-cutting concerns lies behind the limited constructs of the underlying programming language. Now reconsider Listing 1.1 in the context of separation of concerns. Note that multiple UI concerns tangle together into a single component [4]. The colored fragments indicate concerns related to field presentation, layout, security, input validation and data binding. In order to provide variation of a particular concern, such as layout, or varying field order, we have only limited constructs to decompose the layout. Most likely the situation results with producing a novel highly similar variant of code in Listing 1.1, which extends the maintenance efforts.

At the same time, we should note a particular concern perspective. The cross-cutting concerns are repeatedly defined and tangled to particular UI description. This means that a particular concern is captured and defined multiple times and distributed on multiple places in the application. Global change to a particular concern leads to complex and tedious work. This "multi-concern component solution" is the result of the inability of conventional approaches to capture concerns separately [19, 33].

The AOP UI design [4] suggests describing concerns separately, applying the concern assembly at runtime for each individual user request. It also aims to reduce information restatement, thus it involves data inspection utilizing metaprogramming. For instance, the data definition in Listing 1.2 is the subject of inspection in order to produce its presentation. The collected information together with the application context determine the particular data presentation, thus the context itself may change the resulting presentation. For illustration, consider separating the layout from Listing 1.1, which is provided by Listing 1.3; next consider a particular field template in Listing 1.4 that describes a text field. The field template does not reference any data specific elements, which would introduce restatement, instead it uses expressions that either reference the result received from the data inspection or information from the application context. Note the corresponding elements in Listing 1.2 and field conditional restriction to given user roles. Each expression in Listing 1.4 is interpreted upon assembly. The expression elements are not limited to the inspection or context; it can integrate any third party service, such as the validation resolver in Listing 1.4 and involve logical and arithmetical operations. In a case study [4] that involved 63 data definitions with 473 fields, only 28 templates similar to Listing 1.4 were needed for an entire production-level application, which shows that such templates are reusable.

The last piece to the puzzle is the *data field* to *field template* transformation. Various code-based approaches use one-to-one mapping that suits to a single UI situation, although lack the ability to support UI variations. The AOP UI provides a flexibility demanded for context-awareness. The field template selection is determined at runtime by

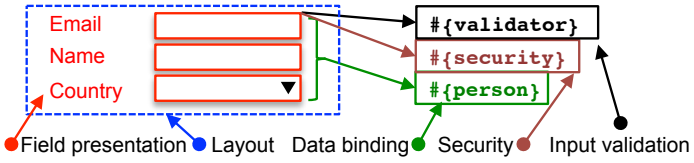

Fig. 1. Graphical sketch of Listing 1.1 denoting individual concerns Figure 2

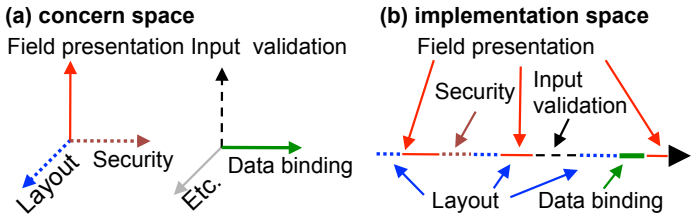

Fig. 2. (a) Concerns as orthogonal dimensions / (b) Implementation space in a single dimension with tangled concerns

transformation rules that query field properties received from the inspection and combine it with contextual information. The collection of such rules allow adjustments to cover various situations determined by context changes. Consider the example rules in Listings 1.5 that consider String-typed fields with default *textTemplate* (Listing 1.4) as well as alternatives for specific field *username*, and for fields enforcing *email* validation, *link* validation or *length* exceeding 255 characters.

Next, we consider AOP formalization and terminology for the above description. First, note the graphical sketch of the description in Listing 1.1 in Figure 1. It shows the form represented as a composition of multiple concerns denoted by different colors. The AOP suggests that each of these concerns can be thought of independently, and Figure 2a shows them in separate dimensions. For instance, when we consider two-column layout where the required input fields come first, we do not need to consider which presentation is used or what kind of validation is involved, etc. When we start to implement it for a particular view, we find out that constructs of conventional programming language are limited. Thus to describe a particular UI presentation, we must capture all concerns tangled together, as depicted in Figure 2b collapsing all concerns into a single source code [20].

AOP can be seen as an extension to General Purpose Languages (GPLs). It differentiates GPL components, and an additional concept called an aspect that is intended to capture a particular concern. Similar to functional or object-oriented decomposition, AOP decomposes a program on components and aspects. An aspect captures cross-cutting concerns separately from the components. In our example the Listing 1.2 represents a component and Listings 1.3-1.5 represent aspects.

The most remarkable construct brought by AOP is the way components and aspects connect together. Each aspect is further divided onto a pointcut and an advice. A *pointcut* specifies a situation, location or context under which an aspect is woven into a component. An *advice* gives concern definition specified either in a GPL or a DSL. Going back to our

examples at Listings 1.4-1.5, the expression language represents a pointcut. The advice of Listing 1.5 is the field template selection, the advices for Listing 1.4 give resolved content.

The next question to answer is to determine what should be the terms used in the expression language. In AOP, it should be something that can effectively address location in GPL components. By the terminology it is called join point. A join point can range from code location specified by a name or a wildcard, method invocation based on method name, annotation, or even information from a particular application context [33]. In our example, a set of join point is associated with each data object, further divided by its fields. The code-inspection gives such a set of join points for each class that is the subject of UI presentation. In [19], such join point structure is referenced as a join point representation.

For instance, the data object in Listing 1.2 provides information such as class name, field names, field types, field properties given by annotation and their specific properties. Furthermore, each annotation property can be further resolved. Moreover, the application context can shadow and even extend the volume of information. Referring back to our examples in 1.4-1.5, each expression uses as terms these join points that in given context match a particular data field and current application context (user location, time, user error rate, server load, etc.).

The considered join point mechanism is not limited to data object inspection. It can involve inspection of Data Transfer Objects (DTO) [13] or it even can be a result of a Builder [14] from XML (or other format) parsing. Furthermore, [8] shows that existing enterprise applications usually involve object-oriented design and classes representing data, such as Listing 1.2 extend the data fields with constraints, input validation rules, security, etc. Java Enterprise Edition platform (Java EE) [9] provides a standard for such extensions, specifically for persistence, constrains and input validation, and this information usually reoccurs in the UI description. Also demonstrated by initial UI description in Listing 1.1. Automated propagation of such information from data elements to UI through code-inspection naturally improves UI development and maintenance efforts.

Understating the AOP UI design, we can consider the UI delivery to clients, in particular client-server communication. The concern assembly takes place at the server-side, which seems a considerable limitation because the client-side cannot take any advantage of the separation. The concern assembly produces tangled concerns with high amount of repetitions, even though the HTTP transmission compression may face the extended size, the delivered information cannot be logically divided on the client-side, which limits its reuse and caching capabilities. As indicated in [4], the approach has the ability to achieve more than 30% reduction of UI code volume, the question to answer is whether it is possible to apply similar concern separations approach to the perspective of UI delivery and receive benefits in form of reduced content transmission, reduced amount of information the server-side has to process, whether it is possible to extend concern reuse at the client-side and thus improve caching options, whether such concern separation allows concurrent processing, can improve UI page load time and responsiveness for data presentations.

Later in this work, we show that it is possible to transmit UI concerns separately to clients, which impacts all the above perspectives. At the same time the client-side becomes partially responsible for the UI assembly, which reduces server-side resources required for information processing. In addition it gives the client the ability to selectively decide which concerns to cache and reuse and which requests form the server-side.

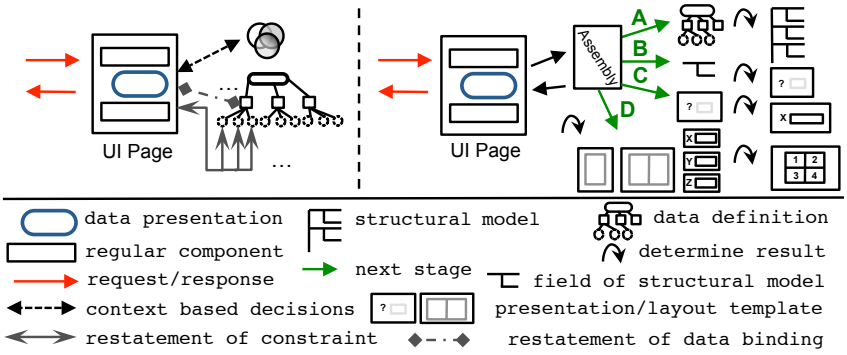

Fig. 3. Conventional approach binding and restatements (left), Code-inspection-based approach life cycle (right). Assembly stages: A - code-inspection to passed data, B - selection of a field presentation, usually determined by field type, or custom rule, C - field presentation content resolver in case we use template for stage B, D - layout decoration.

3. Distributed Concern Delivery (DCD)

In order to describe the DCD design, we put into the contrast the conventional design UI approaches and iteratively extend them towards DCD. 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 Figure 3 (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 Figure 3 (right side). It shows an extension to the left side of Figure 3 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 (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 extension to the assembly in Figure 4 affects the stages A-D. The extension can be seen in consideration of join points, pointcuts and advices (see the legend of Figure 4). 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 con-

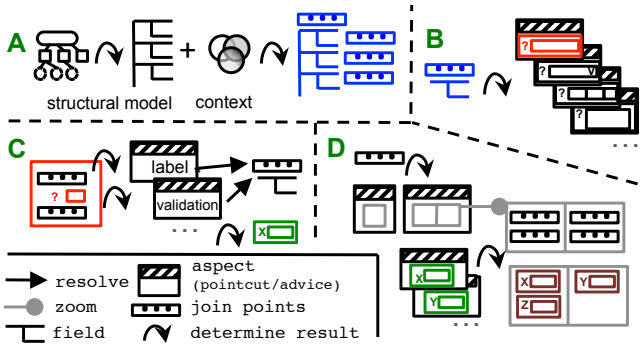

Fig. 4. AOP-based extension to the code-based inspection assembly

text and join point representation is resolved each time it 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 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. As mentioned earlier in the larger case study in [4] only 28 aspects were needed showing the generalization.

The selected presentation template is interpreted using similar mechanisms. Stage C in Figure 4 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 Figure 5 (left side) shows the services provided to clients, when considering the AOP 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 Figure 2a) 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 DCD, the concern weaving process should be partially pushed to client-side. The application data definitions (or data transfer objects [13]) 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 [20]. 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 Figure 5 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 multiple 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.

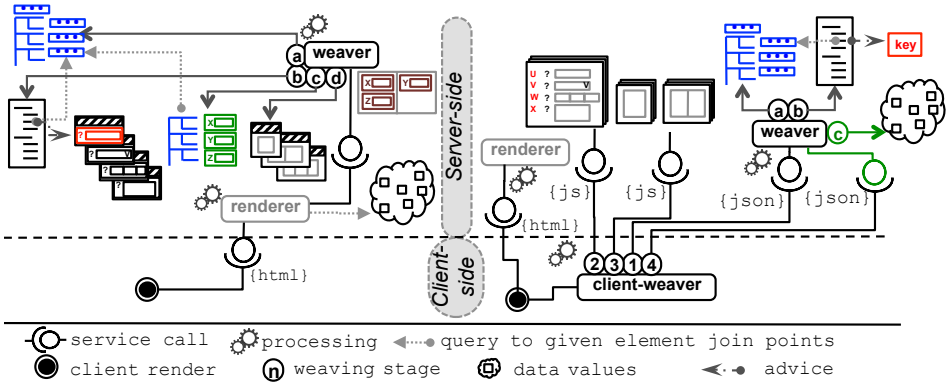

Fig. 5. Services provided by the AOP-based UI design (left-side) / the DCD UI design (right-side)

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 Figure 5, 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 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 DCD are client-side concern reuse, possible caching of UI concerns, reduction of repetitions in the delivery, lessen server side involvement in UI rendering.

4. Evaluation

We implement a DCD UI prototype. We use AspectFaces library[5] to design the server-side weaver that includes the data definition inspection and AOP transformation. It is used by the web services providing the join point model and data values for data instance(s) given the service call parameter. The service output conforms system security and context. Additionally, we implement service that consumes data from clients. Next, we implement the client weaver, a JS library, responsible for UI component assembly and interaction with services. The JS library consists of presentation and layout templates and the client weaver. The implemented prototype is not specific to a particular application, although the amount of considered widget types is limited.

In order to compare the DCD UI approach with a conventional approach with respect to data transmission, page load time, caching and server involvement. We consider an existing production-level web application that uses the client-server architecture. We extract its subsystem for user account management and consider it for the evaluation. In particular, we select a page for the user data manipulation and compare the existing solution with a version that uses the DCD UI design. The considered application bases on the Java EE 6 platform with the JSF 2.1.18 [3] framework for the UI design together with PrimeFaces library 4.0.7 and uses JDK 7. Production system static resources (JS/CSS/Images) are used in the evaluation, although in order to receive equivalent resource transmission, we remove all image references and icons from the evaluated page and place there a single image that represents merged icons.

We deploy the application to a server at Baylor University in Texas, US with the parameters of 8 cores of 2.4 GHz, 16 GB RAM and network access 645/185 Mbits/s download/upload. Next, we consider a client with a Chrome web browser (37.0.2062.124) situated in Prague running on 4 cores of 2.3 GHz, 16 GB RAM and network access 6/6 Mbits/s download/upload throughout the experiment. Round-trip time between the client and server hosts is approximately 140 ms. For the load time evaluation, we consider complete page rendering with 30 measurements for which we provide the average load time and a standard deviation. Compression is applied to the resource transmission.

To measure the load times, we use the standardized HTML5 Performance Timeline[6] that provides performance metric data for the given page. Specifically for the DCD approach that assembles the UI at the client-side, we consider the finish times of the UI composition. Thus for the DCD page version 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.

In the evaluation, we consider five page versions. First, we consider the page given by the production application following the conventional design. The page delivers the entire UI in a single HTML document and then requests addition resources such as JS/CSS/Image. The HTML document consists of multiple forms that are assembled together with the rest of the document components at the server-side. The tangled concerns of the data presentation impact its volume.

[5] http://www.aspectfaces.com, 2014
[6] http://www.w3.org/TR/performance-timeline, 2014

Table 1. Evaluation of a page with 2 data presentations with 22 fields

		Conventional			DCD	
		Single JSF	JSF AJAX per Form	JSF AJAX all Forms	per data	merged
Transmission size [KB]	**No-caching**	219	220	220	219	218
Number of resources		10	12	11	15	13
Server processing [KB]		852	855	855	803	803
Page load time [ms]		1187	1456	1334	1101	1078
Page load time std. dev.		88	54	60	54	38

		Conventional			DCD			
		Single JSF	JSF AJAX	JSF AJAX	per data	merged	per data	merged
			per Form	all Forms	cached structure/context		requested structure/context	
Transmission size [KB]	**Caching**	9,7	11,5	10,9	2,4	2,1	6,4	5,6
Number of resources		1	3	11	3	2	5	3
Server processing [KB]		78,4	81,2	81,6	4,8	4,7	18,2	18
Page load time [ms]		580	787	706	480	464	521	497
Page load time std. dev.		85	61	30	32	25	65	58

Next, we consider extension to the above design in a way that it requests each form fragment asynchronously from the server-side through Asynchronous JavaScript and XML (AJAX) calls. The initial document becomes small, and the content is requested asynchronously, which has the potential to improve the delivery time. Unfortunately, as we explore throughout the study, the JSF framework queues the AJAX calls received at the server-side and processes them sequentially to avoid concurrency issues. For the above reason we consider a third conventional version, requesting all the forms at once by AJAX call. This way we load a small initial HTML document and the form content is loaded in parallel with other resources.

The DCD UI design version is evaluated next. A small initial document is loaded with the page definition; forms are the result of the client weaver generation. First, we consider a strategy (per data) that assembles each data presentation individually. Thus, it requests the join point model and data values from the server for each data instance that is presented at the page. Second, we consider a strategy (merged) that aggregates the multiple data element requests into one request for join points and one for data values. No matter the amount of element we issue two requests.

The first evaluated scenario considers a subset of the registration subsystem. It has 2 data elements in the UI presentation, with 22 fields of various widget types, considering client-side input validation. The total compressed content size that must be transferred from the server to a client for given page versions is shown in the first row of Table 1. We can see that the transmitted content size is similar across these approaches. The second row shows the number of requests made by the client. Naturally, the smallest amount has the conventional single page approach and the highest amount has the DCD strategy (per data), although, these extra requests are issued in parallel. In the next row, Table 1 shows the amount of data that the server has to process, compress and submit. The size variation between the conventional and DCD approaches reflects the dynamic part of the requests. Although the total processing size is almost 50 KB larger in the conventional approach, the compression for the transfer reduces the size to an equivalent of all the page versions.

Next, consider that in the conventional cases the server has the responsibility to process and assemble the UI and render it to HTML. DCD delegates the data presentation assembly to the client. The page load time is considered next, in the fourth row. The

Table 2. Evaluation of a page with 5 data presentations with 40 fields

		Conventional			DCD	
		Single JSF	JSF AJAX per Form	JSF AJAX all Forms	per data	merged
Transmission size [KB]	**No-caching**	221	225	222	223	219
Number of resources		10	15	11	23	13
Server processing [KB]		875	879	879	813	813
Page load time [ms]		1338	1745	1462	1237	1197
Page load time std. dev.		102	70	107	65	45

		Conventional			DCD			
		Single JSF	JSF AJAX	JSF AJAX	per data	merged	per data	merged
			per Form	all Forms	cached structure/context		requested structure/context	
Transmission size [KB]	**Caching**	11,6	16,3	12,9	3,9	2,5	9,8	6,5
Number of resources		1	6	2	6	2	11	3
Server processing [KB]		101	108	106	8,6	7,8	28,1	26,5
Page load time [ms]		625	1258	918	503	489	562	539
Page load time std. dev.		69	53	53	45	87	64	60

fourth row shows results of page load times and the UI rendering are averaged over 30 samples, the standard deviation is shown in the row below. It can be seen that the DCD approach outperforms the conventional approaches. The single JSF page approach processes the main page at the server-side in 150-220 ms, and there seems to be space for improvements. Letting the data presentation fragments load asynchronously looks like a possible improvement. Although, in order to do that we first need to fetch the main page, JS libraries and then we can issue AJAX calls. In the first case with multiple AJAX calls the main page server-side processing drops to approximately 45-70 ms; the first and second form server-side processing takes around 80-100 and 45-65 ms, unfortunately, we get the poorest results because JSF queues and chains the AJAX calls. The second single AJAX approach could bring improvements, the AJAX fragment processing takes around 115-135 ms, and unfortunately together with the RTT latency the page load time is 12% slower than the no-AJAX version.

The DCD approach reduces the main page processing to the range of 10 ms, the web-service requests consume from 10-35 ms and are done in parallel, although the client-side has to process them to assemble the UI. The results in Table 1 show that both merged and per-data DCD strategies give 7-9% page load time improvements compare to the conventional approach. When we consider the main HTML page produced by the conventional approach, it has the uncompressed size of 78.4 KB, which is the size processed by the server-side. In the case of the DCD approach, the main document reduces to 3.3 KB and the requested web services give 14.8 KB of information. The compression mitigates the size difference for the transfer, although the server has to process more information in the conventional case.

Next, let us consider web-browser caching that is usual in modern web-browsers. UI design should take its advantage as it considerably impact page loads. In the bottom part of Table 1 we consider the above measurements with cache enabled. The first row of the bottom part of Table 1 shows large reduction in the transfer size. Basically, we cache all static resources (CSS/JS/Images), but we can see for all the conventional approaches the inability to apply cache beyond this. DCD allows in the addition to cache the join point model, presentation templates and layout as well as the weaver. DCD only requests the main page and data. Note that we considered the join point model cacheable for a given

Table 3. Evaluation of the 3G environments with 22-field page

		Conventional			DCD			
	No-caching	Single JSF	JSF AJAX per Form	JSF AJAX all Forms	per data	merged		
Page load time [ms]		6169	5979	5997	5499	5581		
Page load time std. dev.		326	626	538	286	164		
	Caching	Conventional			DCD			
		Single JSF	JSF AJAX per Form	JSF AJAX all Forms	per data cached structure/context	merged cached structure/context	per data requested structure/context	merged requested structure/context
Page load time [ms]		806	1040	1030	533	522	601	578
Page load time std. dev.		38	54	41	21	20	66	49

session, although it could possibly change based on user actions, time, etc. The client-side has the ability to make an individual decision whether or not to cache a particular resource. The strategy would be influenced by the granularity and the scope of the context-awareness. To cope with situations involving strong context-awareness we also consider a case re-transmitting the join point model.

The conventional approach has to re-transmit all the presentation, layout, data, and structure all tangled together. Since it is tangled, the caching options are limited. The consequent row of Table 1 shows the amount of requests that are being made (we omit cached resources/header requests). The next row shows the server processing content size. This is followed by page load results and the std. deviation. The page load times improve by 17-20% for the DCD approach compared to the conventional approach, when we cache the join point model and 10-14% when we reload it.

Next, we consider the impact of increase in data volume presented at the page. Thus we consider a broader part of the production subsystem and evaluate a page that provides 5 data entities with 40 fields that are being presented in 5 page forms. We may assume that the per-data DCD strategy has to issue 10 more requests for the join point model and data, which gives it a considerable disadvantage, although, this happens in parallel. Table 2 shows the results that consider the same attributes as the previous evaluation. We can see that the server-side processing size further reduces for the DCD approach for both cache-disabled and cache-enabled cases. The cached case transmission sizes scale better for DCD. The page load time of DCD outperforms the conventional approach by 7-11% in the cached-disabled case and by 20-22% in the cache-enabled case that does not request the join point model. When there are context changes and we would request the join point model, then we get 10-14% load time improvement. The volume of presented data on the page does not seem to impact the efficiency of the DCD approach.

Next, consider what happens when the network conditions change. We evaluate a scenario of a user with a mobile device that uses (384 Kbps) 3G networks with limited bandwidth and latency. In the evaluation, the network bandwidth is restricted to 384 Kbps on the client-side and an extra latency 20 ms is added using Mac tool called Speed Limit. Table 3 shows the page load times for both cache-enabled and cache-disabled cases. While the cache-disabled case shows DCD UI performance 9-11% better than the convention page, the cache-enabled case shows 34-35% better performance when we reuse the join point model and 25 - 28% when the context changes are considered.

The above examples demonstrate enhanced caching options brought by the DCD approach, which has a positive effect on page load time. Furthermore, concurrent request

handling positively impacts page load times for the cache-disabled scenarios. The UI assembly delegation to the client-side reduces the server-side efforts, for example in our evaluation the total size of information the server has to process is reduced by 6-7% for the cache-disabled case. When we consider only the dynamic part of the page then the reduction represents 76-82%. When we apply caching, we can further cache the join point model and thus the reduction extends to 92-94%. From the study, we can notice that both the per-data and merged DCD strategies bring improvement to the transmission size. The per-data strategy extends the amount of requests that can be handled concurrently, but also brings overhead with extra requests and HTTP headers.

Next, we discuss the result outcome from the study, possible generalization of results and threats to validity:

Internal validity: To mitigate the impact of network fluctuation to the measured results of page load times we averaged 30 samples of the same scenario, where we re-run each measurement right after each other with 5 second delays. The measurement involves HTML5 Timeline that mitigates skew results. To avoid false times related to client-side execution, for the DCD 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.

External validity: Similar to [22] 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 DCD results show extended ability to reuse concerns at the client-side and to apply caching, which both reduces the server-side involvement in the UI rendering and impact on reduced UI description volumes. The constellation of client-server represents one particular scenario to provide real-world results rather than a wide spectrum of constellations. At the same time, we consider restrictions to the network conditions to evaluate mobile-like 3G environments. We could use laboratory environment for the study, although the goal was a demonstration on a real environment to provide practical impact. To avoid single application framework perspective we consider two alternative frameworks and draw the impact from the perspective of the transmitted content volume. Similarly to [26], this case study serves as a demonstration of DCD impact on performance and transmitted volumes when compared to conventional concern-tangling applications. The study considered the ability of a conventional web-browser Chrome, although its alternatives provide similar results.

5. Related work

Various approaches have been introduced to simplify development of complex UIs. They can be divided into model-based, generation-based, inspection-based, and AOP-based. Each of these offers certain advantages for UI development; however, they typically fail to address simplified UI maintenance or complex situations, such as context-aware UIs adapting during runtime. In terms of client-server communication, conventional ap-

proaches transfer large, perhaps unnecessary, amounts of data, which negatively impacts responsiveness.

Model-driven development (MDD) [28] suggests that a model is the sole source of information, and the resulting source code is generated [29] using the model through a set of transformation rules. The main advantage is the reduction of information that must be restated manually [8] for different perspectives. MDD can be applied to context-aware UIs [21]. A deep explanation of model-to-model and model-to-code transformations in the area of model-based UI development [31] [32] notes that when we aim to describe different concerns through multiple models, MDD does not provide a standard integration mechanism to do so. Although, MDD can be used to capture complex UIs, various contexts, and provide adaptive features, the model-to-code transformations may struggle from the performance perspective [21]. Transformations are usually performed at compile time, since they tend to be time consuming [21]. The disadvantage of compile-time transformations is that they produce source code and descriptions for all possible context states, many of which may never be used [25].

Next, MDD suffers during adaptation and evolution management [25]. It handles basic situations well, although when considering variations and customization, the modifications often take place in the UI code [8] rather than in the model. This leads to difficult MDD maintenance, since manual changes cannot be removed upon the model-to-code transformation. An approach dealing with synchronization between models, generated code and runtime systems is elaborated in [12]. Most of the UI designs based on MDD do not consider the correlation with other parts of the system, such as persistence or business logic subsystems [4]. This deficiency is noted in the research discipline of human-computer interaction [21]. In such cases, information captured by models must correspond to information captured by the subsystems. When one part changes, the other must reflect the change to avoid inconsistency errors; unfortunately this now must be handled manually [4].

The difference from our approach is that we do not explicitly enforce any explicit model to be the central source of information for the UI generation. This mitigates the necessity to learn a new model, and avoids possible correlation issues with the data definition objects. Our approach considers the data definitions and context to be a subject of inspection from which an ad-hoc model is built. Next, the model transformation rules are usually strict and derived at compile time, in our approach an aspect-oriented mechanism applies integrating custom runtime information. The product of MDD is usually a tangled UI description, in our approach we let concerns divided for the server-client interaction and client becomes responsible for concern composition. Thus MDD could possibly generate large amount of application states and produce large UI descriptions, in our approach each state would be provided incrementally.

The use of a DSL [24] is common for UI model description, even for direct specification of UIs [16, 24]. Consider the Java EE standard for component-based development JSF [3] in Listing 1.1. The DSL provided by JSF brings simplification to the UI description [4], as oppose to a GPL. Typically, it is transformed to the target UI language, such as HTML and JavaScript (JS). DSLs naturally fit to UI descriptions, but they may bring weak type safety, which complicates maintenance, since it is easy to introduce errors [17]. For example, a DSL description may reference data, their fields and constraints that are already described in the application through a GPL [11]. However, referencing a GPL

component from a DSL requires certain restatement with a negative impact on maintenance as shown in Listing 1.1. Similarly to MDD, the DSL-to-target code transformation takes place at the server-side, and usually does not deal with concern separation, which leads to large volumes of information transmitted to clients, the inability to logically separate information at the client-side for the caching purposes. On the other, hand DSL languages fit well for description of particular concerns and are usually used for AOP pointcuts and advices.

Another approach [4, 17, 18] addresses information restatement by utilizing code-inspection and metaprogramming using introspection. Usually GPL data definitions are inspected internally by drawing a structural model, that is transformed to UI descriptions with all data / constraint references resolved through the model. Such approach can be compile time or runtime. It avoids human-errors related to inconsistencies or typological errors since information from data definitions propagate to the result of transformation, the UI description. In comparison to the above approaches, this approach works at runtime, although, it does not address cross-cutting UI concerns, and the product is derived at the server-side. Our approach uses code-inspection to obtain join points from data structures, as the effect, information from data definitions avoids inconsistencies. The code-inspection is an essential part for our approach, although it is just the initial stage.

Approaches discussed so far only partially or indirectly addressed cross-cutting concerns. One possible solution to address cross-cutting concerns is Generative Programming (GP) [10, 30]. The aim is to emphasize domain-specific methods to address certain concerns and their integration with co-existing GPL components. GP can be defined as type of programming that generates source code through generic code fragments or templates [10], which is not far from MDD, although it is not tight to models. The goal is to address gaps between program code and domain concepts, support reuse and adaptation, simplify management of component variants and increase efficiency. The generation is carried out at compile time. Its use for UI [30] considers abstract UI specifications. The concept consists of three parts: a DSL for UI description, configuration generator that automates the product UI assembly and an extensible collection of elementary components available for the assembly. The configuration generator considers various transformation rules. It takes the problem specification (in a DSL) and UI components from the target language, and it assembles them together to produce the result. It usually produces a large number of component variants for specific requirements. In a case study, given in [30], a system combines two hundred UI features resulting with variability of 5×10^{17} prototypes. However, it is questionable whether such a large number of feature-aware components is reasonable and could be ever used; all the states are generated at compile time and composed physically. Consider what happens if such large amount of states needs to be provided to clients, this can demand large amount of resources, mostly when concern combinations grow exponentially [25].

Our approach similarly uses GPL components and DSL extension together with an assembly. The difference is that it applies code-inspection to introduce join points. Next, it considers runtime context as it uses runtime generation. The integration uses AOP mechanisms and there is not generation that would produce all possible states at once. In our approach we generate one UI state that reflects particular conditions for given user and context each time UI is requested. Next, we keep the concerns divided for the client interaction.

Besides the AOP approach we introduced, alternative AOP approaches were proposed usually to extend capabilities of other existing approaches. In [25] the authors apply AOP to MDD to support adaptive features at runtime. This work suggests that MDD approaches do not naturally fit into adaptive systems because they lack the runtime information, which determines the model-to-code transformation. The MDD runtime transformation might be inefficient [32] for complex situations [21]. The compile-time transformation may struggle with the exponential grow of hypothetical/possible situations for which it generates the code. In [25] the authors suggest using four runtime models that represent main system data that are manipulated at runtime to accomplish adaptations. Unfortunately, the description of the aspect-oriented conceptual model [33], weaving process, and context is sparsely described in [25] to provide deeper analysis. Furthermore, no performance consideration is given to the manifest approach effectively for production systems.

The approach we extend in this paper [4] was described in Section 2. It applies code-inspection and the AOP transformation utilizes separation of concerns for data representations. It applies to code-based development and builds on existing enterprise system standards, the approach shows the performance comparable with existing conventional technologies [4] as well as the capability to design context-aware UIs. The runtime integration can reflect changing conditions and the output is not limited to a particular technology. In [21] is provided integration with UI Protocol to stream platform-specific UIs and to apply automated element distribution at users screen based on metrics. The drawback of the approach is it limited view on client interaction; the UI is generated at server-side, which degrades the advantages of concern separation form clients' perspective. In [6] we introduce the idea of concern separation applied the UI delivery and provide preliminary evaluations. In this paper, we elaborate and extend the approach details mostly for particular stages. The approach capability is extended with resource aggregation to face the growing amount of requests, while lowering the transmission size. The limitations and advantages are put to contrast with alternative AJAX-based UI approaches, where possible integration may provide synergy.

Existing research in UI rarely or indirectly addresses optimization of UI delivery to clients or client-side caching. On the other hand, there exist contemporary UI frameworks that address caching and UI delivery. Specifically we consider the Google Web Toolkit (GWT) [15] and AngularJS [1].

GWT framework provides abstraction in a way that developers describe the UI in GPL that compiles/transforms to DSL. Specifically Java translates to JS. The claimed advantage is the improved type safety, although this is only partially truth. GWT similar to other UI frameworks consider its task to design UI and to provide composeable components for the UI. The co-existing data definitions are referenced through GPL in UI descriptions, although consider Listing 1.2 and the field name or field annotations [11] commonly used in Java Enterprise Edition. There is no mechanism in GWT to lookup annotation and its properties or field names while preserving type safety. The widget selection is also left to the designer, thus changes in data definitions impact the correlation with the UI.

The GWT abstraction approach uses GPL language although others could argue that UI is the domain for DSL languages with different idiom. The abstraction of GWT could be placed to relation with MDD or GP. It generates the UI JS code at compile-time optimizing its options for all major web-browsers to allow placing the entire application logic to JS that is sent to client. Naturally, this gives the client a UI that loads once with all pos-

sible state transitions, which gives the feeling of a standalone-like application. The data fraction is intended to be loaded separately through web-services. The targets of the GWT applications are interactive and living websites, such as interactive console, email client, task management where usually limited amount of page transitions exists. The nature of GWT does not fit information systems with multiple pages and forms, because usually the whole application logic loads at once, although it is possible to partition the provided states with incremental load.

Among the disadvantages, Java may use different idioms than JS considering web applications, the abstraction makes the debugging hard, and the produced JS can enormously grow, which is expectable considering that all the application logic is being sent. The GPL nature does not address cross-cutting concerns and a lot of manual work is left to designers through the information restatement. We see the limitation of GWT usage to a certain kinds of applications and its inconvenience for data management systems. The advantage is that GWT separates out data fraction. In our approach we separate larger amount of concerns. GWT provides the client with the entire state transitions generated at compile time. In contrast, our approach provides one particular state to the client; the state is resolved at runtime. Our approach fits to data-oriented systems and GWT fits better to interactive single page applications. Our approach does not bring high level of abstraction, although it could integrate it. Later in in Section 6 we provide detailed comparisons with GWT and our approach.

AngularJS is a UI framework that brings similar to GWT the separation of data fraction from the rest of the UI; in addition to it AngularJS brings string templating mechanisms that resolved by a "HTML compiler" at the client-side. This is not far from our approach as we intend to provide the client multiple concerns and let them integrated at the client-side. Similar to GWT also AngularJS considers its task to provide UI constructs and thus high amount of restatements between co-existing data definition and it its UI presentations exists. Furthermore, AngularJS is an example of languages with weak type safety. As oppose to GWT AngularJS is low level and has limited abstraction, which allows low-level optimization and easy integration with other approaches. AngularJS, in the contrast with GWT, goes the direction of incremental state requests, which fits to data-oriented, multi-page systems. Its templating mechanism is promising direction; on the other hand, the amount of template integration is limited. For instance, to dynamically integrate template into a template is complex. In comparison to JSF templates, it does not give the decoration option, which would extend the expressiveness. Similar to GWT, it involves DTO for data values, which extends development and maintenance efforts. As in GWT the designer is responsible to prepare the data presentation define the structure, populate selection/option values, etc.; with the connection of weak type safety this brings considerable efforts. More detailed comparison with our work is discussed in Section 6.

Indirect improvements to the web resource delivery are addressed by HTTP protocol mechanisms, or by resource distribution across the network. HTTP allows clients to open and reuse multiple TCP connections to the server so the nature of HTTP supports concurrency. Next, it can apply content compression for the transmission and supports client-side caching of resources. HTTP caching applies mostly to static resources such as CSS, images, and JS. Alternative, HTML5 Local Storage involves reuse of JSON and XML received from asynchronous calls. In addition HTML5 brings mechanisms to receive performance statistics through Performance Timeline. Usual strategies to improve

UI page load involve resource merging and content obfuscation. To mitigate the impact of underlying network delay, servers apply geo-distributed caching of static resources called content-delivery networks (CDNs), such as Akamai [27]. With our approach we aim to extend caching option for data presentations and thus the HTTP abilities are crucial mechanisms that are involved, in this work we further involve resource merging for the same concerns of multiple different elements presented by the same page. When considering context-unaware UIs, we could consider caching of certain UI concerns at CDNs, although we leave it for future work. Similarly to CDN, another optimization approach is brought by cooperative-web cache (CWC) [7]. The idea is to involve clients in the UI delivery with sharing cached resources. Experiments provided by [7] address static resource sharing among clients using an overlay peer-to-peer network to improve page load times and server scalability. Unlike CDNs it supports natural scalability and free services; however, it must deal with content invalidation and mechanisms preventing sharing corrupted data from malicious clients. Considering context-unaware UIs we could share certain UI concerns by the overlay, but this is left for future work experiments.

Extensions to HTTP [23, 34] are the subject of research of Structured Hypertext Transfer Protocol (STTP) [34] and HTTP-MPLEX [23]. STTP introduces new kind of messages to control the resource transmission for a particular web page. HTTP-MPLEX employs a header compression and response-encoding scheme for HTTP. Similar to STTP, HTTP-MPLEX multiplexes multiple responses to a single sustained stream of data to speed response times. While this might work for laboratory experiments for a particular page, we consider this approach hard to use for complex and large context-aware systems while considering HTTP caching mechanism or resource distributions.

6. Comparison with AJAX-based approaches

The evaluation in Section 4 considered an existing production-level application with a conventional approach and the comparisons with DCD. Next, we consider DCD comparison with AJAX-based approaches. In particular we consider GWT and AngularJS. Similar to Section 4 we implement three prototypes that base on production application (the page version with two data elements). Since the third party libraries would be alternatives to the solutions in the other AJAX-based approaches, we modify the application so that we eliminate all the static resources relevant to the production application and only consider resources needed for the particular prototype. Besides the approaches comparison we consider extension of GWT and AngularJS with DCD and discuss the implications.

AngularJS changes the UI design towards DCD. Specifically, it suggests to split data values from the rest of the UI page. The values are provided in JSON format similar to DCD approach. The difference is that the data value generation and UI matching is left for manual intervention. Usually, DTOs are designed which brings restatement on the server-side. The UI page then binds the data, which is again left to developers. AngularJS Forms do not bring any simplification to the design and is similar to JSF. The developer has to make the widget selection, apply data binding, enforce constraints and validation as well as populate the selection values for options. This is error-prone and tedious task mostly due to limited JS type safety. Even though, AngularJS supports client-side templating mechanism and provides a HTML compiler to apply the templates to provided values it does not address form integration with this idea. DCD mechanism considers the

templating from a different perspective, for example to include form fields. We believe that AngularJS is missing template support for decoration, similar to JSF, which would simplify the template use for more complex task such as form fields. On the other hand, what AngularJS provides is a client-side data model that simplifies population and derivation of data values in forms, and further extends form management with strong validation framework.

Our prototype application that implements the considered UI page with data values separated to JSON has altogether the total size of 58.9 KB in 4 requests as shows Table 4 in the first row. The majority is the AngularJS library. The main document has 5.8KB (30.0KB uncompressed); the data values supplement the main document. The cached version has to load the main page and data, which is 6.4KB. In our evaluation we go beyond the comparison and integrate the DCD idea to the AngularJS. The join point model is separated out and the weaver builds the data presentation from AngularJS components that allow seamless integration with the AngularJS validation and data value manipulation. Our integration prototype is not limited to the evaluated page and is capable of presenting any data element provided by the server-side service. On the other hand, we provide limited amount of validation. The page statistics changes to 64.2 KB in 6 requests (with the weaver 6.1KB). The main document reduces to 1.6KB (3.6KB uncompressed); the join point model has 3.4KB. The consideration of cached UI page with unchanged context, or context-insensitive pages need to transfer 2.2KB, the context-aware cached page needs 5.6KB to load. The summary and additional information are provided in Table 4, second row. This approach extends the total size for the uncompressed page due to the weaver size, although it amortizes over the time when cached. Next, the cached page for the context-unaware situation reduces the transmission by 65%, the context-aware situation is reduced by 12%, although this is not the main benefit. The advantage is reduced development and maintenance efforts at both server and client sides. The join point model and data values are automatically derived, the client weaver then populates the data presentation structure, binding, constraints, validation, and selection values, which greatly reduces the efforts and possibility for human-errors. DCD enforces correlation between server and client sides.

An application where data definitions do not reflect the UI can still benefit and apply. There are three possibilities, one that involves DTO, second that requires correlation among DTO and data definition structure and a last one merging and filtering multiple data definitions. The first option would consider DTOs for the inspection, the extended information such as constraints and validations [2, 11] would need to apply to DTOs. The second would apply the derived join point of multiple data definitions to the structure given by a particular DTO. Last, the UI designer indicates, which definitions participate in the UI and then applies a filter set at the UI level.

The same page designed with GWT pushes the design to high abstraction since the entire application is designed in a GPL Java. As stated in Section 5, GWT suits better to one page applications that can keep states for offline work, but does not suit to multi-page applications, because it usually compiles the entire page state transitions to JS that is sent to the client. The designed page is compiled to JS, which is a one time process that produces multiple versions of the UI for support of various web browsers. The GWT data presentation uses GPL, although it still consists of restatements, such as constrains and validations, it has repeated decisions, manual selection value population, etc. The

Table 4. AJAX-approach comparison

** JP - Join points*	Transmission size [KB]	Resources	Server processing [KB]	Transmission size [KB]	Resources	Server processing [KB]
	no caching			caching		
AngularJS	58.9	4	174	6.4	2	31.1
AngularJS + DCD (with JP)	64.2	6	188	2.2 (5.6)	2 (3)	4.7 (16.6)
GWT	55	4	152.8	5.5	3	11.8
GWT + DCD (with JP)	54.9	5	155.8	5.5 (9)	3 (4)	11.8 (24.5)
DCD (with JP)	25.2	6	87.2	1.8 (5.3)	2 (3)	3.7 (16.4)

approach is similar MDD process where the GPL Java is the model and the JS is the target language. The compilation applies strategies to obfuscate and optimize the UI pages. Our prototype with manually designed data presentation and divided data values, similar to AngularJS, consists of the total of 55 KB with 4 requests, as shows Table 4 third row. The main page has only 1.4KB (3KB uncompressed). The cached version issues 3 requests, the main page, non-cacheable part of JS and the data altogether 5.5 KB. We consider the application of the DCD similarly to AngularJS to receive the automated data presentation, binding, etc. The difference is that the weaver and templates consider GPL Java. The DCD GWT prototype has the size of 54.9KB with 5 requests. The join point model is separated out, which deduces the produced JS size. We see that the compilation in the original GWT application did an optimization comparable to the size of join point model in the DCD version. The cached version for context-insensitive UI has the equivalent size 5.5KB; the context-aware situation would request the join point model with the total of 9KB. The summary is provided in Table 4, fourth row.

We can see that GWT brings significant optimization to the UI. This requires high level of abstraction and compilation. The GWT targets interactive live web pages with limited amount of page navigation so that all application states are sent at once to the client. The DCD as well as AngularJS fits better to system where large amount of page exists, such as information systems or enterprise software applications. The more detailed comparison with GWT would require a larger application to compare.

The same application prototype using solely DCD need 25.2KB in 6 requests to load the UI page, the cached version needs to transmit 1.8/5.3KB for the context-insensitive / context-aware situation as shows Table 4 last row. The AngularJS integration looks promising for practical use as it has large community and DCD integration can bring simplification to the development and maintenance efforts.

7. Conclusion

In this paper, we discuss issues related to conventional UI designs, which source from tangled concerns. Tangled concerns are responsible for increased development and maintenance efforts and weak readability. We identify that concern separation can provide benefits to the client-server communication as well as to the client abilities. Concerns that are tangled together can hardly be reused, which disallows the possibility to cache them at the client side and reduce the network communication demands. Existing concern separating approaches separate the concerns only at the server-side and the separation gets lost upon rendering. We research the distribution of concern weaving across the client and server-sides to maintain the concern separation at the client-side and provide extended abilities

to clients. We consider DCD approach, as an extension of the AOP-based UI approach providing the benefits of automate code-inspection and AOP-based transformation.

This approach allows us to reuse sedately provided concerns at the client-side and apply caching. The concern separation reduces the volume of transferred UI related information through reduced restatements. The partial delegation of the UI assembly from the server to client reduces the required server-side resources to render a particular data UI presentation. Next, the approach supports concurrency and the nature of the approach support context-aware UI features.

We evaluate DCD approach on a case study that bases on a production system and demonstrates the benefits. The study results show reduced page load times for the considered scenario. Next, it shows reduced volume of UI data presentation description that must be processed by server-side. The reduced data transfer is evident for the client cache-enabled scenarios, which shows the extended client-side caching abilities over the conventional approach.

We also consider alternative AJAX-based approaches AngularJS and GWT, which bring different perspective to the UI development, although in both approaches are left gaps regarding the data presentation UI design, mostly regarding to involved development efforts, maintenance, repeated decisions and information restatement. From the study can be seen the differences of the approaches. While AngularJS represents incremental state client-request approach, the GWT provides the entire application state space in one request. Both approaches also present a different level of abstraction. Upon DCD integration to both approaches we can see considerable synergy with AngularJS although limited advantages are given to the GWT approach, which shows DCD to better fit to incremental state client-request approaches.

Contribution of this paper is the detailed description of the DCD approach with discussion of its benefits. The advantages are demonstrated on a case study. Co-existing AJAX-based approaches are considered from the DCD perspective and the provided integration shows the possible synergy mostly for incremental state client-request approaches.

Although the results of the case study are promising, we still must consider DCD limitations. The design approach fits to data presentations; it builds on the top of other approaches that deal with page-flow, and other UI tasks. One of the promising future directions is the integration with AngularJS. The underlying AOP UI design approach interacts with development standards and third parties for security, context-awareness, etc., although it is only aimed for data presentations.

Regarding future work we have a preliminary client DCD prototypes involving standalone and mobile based clients. The join point model and data values are enough to determine the data presentation on such platform-specific clients. At the same time this brings usability advantages to end users. This can also lead to the ability to describe UI data presentations in platform-independent format [5], open web applications to further reuse and simplify native client development.

Next, we could consider separating out the static and dynamic particles of the join points, which would allow caching the static particles for a long period of time. This approach could go the direction of GWT when the entire static particles load at once and persist in cache. This would also allow considering the CDN and CWC integration.

The discussion of AngularJS and GWT approach gives two different views on UI development, on one side full state space given to client, on the other side an incremental

approach. Is this the only possibility or does there exists even finer granularity that could be determined at runtime basing on client abilities, battery consumption, etc.? What if the client uses a watch but the other uses a desktop? Perhaps with GWT one client might receive usable UI but the other that is above the ability of the watch to be processed. Perhaps AOP abilities could provide the granularity to choose the appropriate strategy to interact with the client, all with the low development and maintenance efforts.

The AngularJS DCD integration will be considered more closely with aim to push for AngularJS plugin to contribute the community.

Our preliminary research in the area of application business rule allows us to apply inspection not only to data definitions but also the business rule definitions, and such definition could be provided as separated concerns, which would allow to reuse and transform existing business rules in different subsystems. For instance, consider the client-side UI business rule integration allowing (for certain rules) applying rule resolution at the client-side. This would improve the UI usability. The UI might not be the only target domain for the DCD approach; it might be used in middleware integration.

Acknowledgements. This research was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS14/198/OHK3/3T/13.

References

1. AngularJS documentation (April 2015), http://angularjs.org
2. Bernard, E.: JSR 303: Bean validation (Nov 2009)
3. Burns, E., Griffin, N.: JavaServer Faces 2.0, The Complete Reference. McGraw-Hill, Inc., New York, NY, USA, 1 edn. (2010)
4. Cerny, T., Cemus, K., Donahoo, M.J., Song, E.: Aspect-driven, data-reflective and context-aware user interfaces design. Applied Computing Review 13(4), 53–65 (2013)
5. Cerny, T., Donahoo, M.J.: Separating out platform-independent particles of user interfaces. In: Information Science and Applications, pp. 941–948. Springer Berlin Heidelberg (2015)
6. Cerny, T., Macik, M., Donahoo, M.J., Janousek, J.: Efficient description and cache performance in aspect-oriented user interface design. In: Proceedings of the 2014 Federated Conference on Computer Science and Information Systems, Warsaw, Poland, September 7-10, 2014. pp. 1667–1676 (2014)
7. Černý, T., Praus, P., Jaroměřská, S., Matl, L., Donahoo, M.: Towards a smart, self-scaling cooperative web cache. SOFSEM 2012: Theory and Practice of Computer Science, LNCS 8327 pp. 443–455 (2012)
8. Cerny, T., Song, E.: Model-driven Rich Form Generation. Information: An International Interdisciplinary Journal 15(7, SI), 2695–2714 (JUL 2012)
9. Chinnici, R., Shannon, B.: JSR 316: Javatm platform, enterprise edition 6 (java ee 6) specification (Dec 2009)
10. Czarnecki, K., Eisenecker, U.W.: Components and generative programming (invited paper). In: Proceedings of the 7th European software engineering conference held with the 7th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 2–19. ESEC/FSE-7, Springer-Verlag, London, UK, UK (1999)
11. DeMichiel, L.: JSR 317: JavaTM persistence API, version 2.0 (November 2009)
12. Djukić, V., Luković, I., Popović, A., Ivančević, V.: Model execution: An approach based on extending domain-specific modeling with action reports. Computer Science and Information Systems 10(4), 1585–1620 (2013)
13. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
15. Hanson, R., Tacy, A.: GWT in Action: Easy Ajax with the Google Web Toolkit. Manning Publications Co., Greenwich, CT, USA (2007)
16. Karu, M.: A textual domain specific language for user interface modelling. In: Sobh, T., Elleithy, K. (eds.) Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering, Lecture Notes in Electrical Engineering, vol. 151, pp. 985–996. Springer New York (2013)
17. Kennard, R., Edmonds, E., Leaney, J.: Separation anxiety: stresses of developing a modern day separable user interface. In: Proceedings of the 2nd conference on Human System Interactions. pp. 225–232. HSI'09, IEEE Press (2009)
18. Kennard, R., Leaney, J.: Towards a general purpose architecture for ui generation. Journal of Systems and Software 83(10), 1896 – 1906 (2010)
19. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.M., Lopes, C.V., Maeda, C., Mendhekar, A.: Aspect-oriented programming. In: In ECOOP'97-Object-Oriented Programming, 11th European Conference. vol. 1241, pp. 220–242. Springer (June 1997)
20. Laddad, R.: AspectJ in Action: Enterprise AOP with Spring Applications. Manning Publications Co., Greenwich, CT, USA, 2nd edn. (2009)
21. Macik, M., Cerny, T., Slavik, P.: Context-sensitive, cross-platform user interface generation. Journal on Multimodal User Interfaces pp. 1–13 (2014)
22. Matthijssen, N., Zaidman, A., Storey, M., Bull, I., van Deursen, A.: Connecting traces: Understanding client-server interactions in ajax applications. In: Program Comprehension (ICPC), 2010 IEEE 18th International Conference on. pp. 216–225 (June 2010)
23. Mattson, R.L.R., Ghosh, S.: HTTP-MPLEX: An enhanced hypertext transfer protocol and its performance evaluation. J. Netw. Comput. Appl. 32(4), 925–939 (2009)
24. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (Dec 2005)
25. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. Computer 42(10), 44–51 (Oct 2009)
26. Nurseitov, N., Paulson, M., Reynolds, R., Izurieta, C.: Comparison of JSON and XML Data Interchange Formats: A Case Study. pp. 157–162. CAINE (2009), http://www.mendeley.com/research/comparison-json-xml-data-interchange-formats-case-study-4/
27. Nygren, E., Sitaraman, R.K., Sun, J.: The akamai network: A platform for high-performance internet applications. SIGOPS Oper. Syst. Rev. 44(3), 2–19 (Aug 2010)
28. Perez-medina, J.l., Dupuy-chessa, S., Front, A.: 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). pp. 84–97. Springer, Berlin (7-9 Nov 2007)
29. Perišić, B., Milosavljević, G., Dejanović, I., Milosavljević, B.: Uml profile for specifying user interfaces of business applications. Computer Science and Information Systems 8(2), 405–426 (2011)
30. Schlee, M., Vanderdonckt, J.: Generative programming of graphical user interfaces. In: Proceedings of the working conference on Advanced visual interfaces. pp. 403–406. AVI '04, ACM, New York, NY, USA (2004)
31. Sottet, J.S., Calvary, G., Coutaz, J., Favre, J.M.: A model-driven engineering approach for the usability of plastic user interfaces. In: Engineering Interactive Systems, pp. 140–157. Springer (2008)
32. Sottet, J.S., Calvary, G., Favre, J.M.: Models at runtime for sustaining user interface plasticity. In: Models@ run. time workshop (in conjunction with MoDELS/UML conference) (2006)
33. Stoerzer, M., Hanenberg, S.: A classification of pointcut language constructs. In: Workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT) held in conjunction with AOSD (2005)
34. Swen, B.: Outline of initial design of the structured hypertext transfer protocol. J. Comput. Sci. Technol. 18(3), 287–298 (2003)

Tomas Cerny is a doctoral candidate at the Faculty of Electrical Engineering of Czech Technical University (FEE, CTU) in Prague, Czech Republic. He received his Bachelors and Masters degrees from FEE, CTU, and M.S. degree from Baylor University. Since 2009 works as an Assistant Professor at FEE, CTU. His area of research is software engineering, separation of concerns, model-driven development, enterprise application development and networking.

Miroslav Macik is a doctoral candidate at the Faculty of Electrical Engineering of Czech Technical University (FEE, CTU) in Prague, Czech Republic. He received his Masters degree from FEE, CTU. His area of research is user interface, accessibility, computer-aided hospital navigation and usability.

Michael Jeff Donahoo received his B.S. and M.S. degrees from Baylor University and his Ph.D. in Computer Science at the Georgia Institute of Technology. He is currently an Associate Professor of Computer Science at Baylor University where he conducts research on networking, security, and enterprise application development.

Jan Janousek received his Master's and Ph.D. degrees from Faculty of Electrical Engineering of Czech Technical University in Prague (CTU). In 2010 he habilitated at Technical University FIT in Brno, Czech Rep. Jan is currently an associate professor and the head of Department of Theoretical Computer Science at Faculty of Information Technology, CTU, Prague. His research interests include processing trees and strings, compiler construction, formal languages and automata theory, and attribute grammars.

Received: December 2, 2014; Accepted: May 28, 2015.