DOI: 10.15439/2014F244
ACSIS, Vol. 2

Efficient Description and Cache Performance in Aspect-Oriented User Interface Design

Tomas Cerny*, Miroslav Macik†, Michael J. Donahoo‡ and Jan Janousek§
*Computer Science, †Graphics and Interaction at Czech Technical University,
Charles square 13, Prague 2, Czech Republic, Email: {tomas.cerny}*, {macikmir}†@fel.cvut.cz
‡Computer Science at Baylor University, Waco, TX
One Bear Place #97356, , 76798-7356, USA, Email: jeff_donahoo@baylor.edu
§Theoretical Computer Science at Czech Technical University
Thakurova 9, Prague 6, Czech Republic, Email: jan.janousek@fit.cvut.cz

***Abstract*—Increasing demands on web user interface (UI) usability, adaptability, and dynamic behavior drives ever growing development and maintenance complexity. Conventional design approaches scale poorly with such rising complexity, resulting in rapidly increasing costs. Much of the complexity centers around data presentation and processing. Recent work greatly reduces such data complexity through the application of Aspect-Oriented UI (AOUI) design, which separates various UI concerns; however, rendering in conventional and even AOUI approaches fails to maintain this separation, often resulting in high repetitions of concern fragments due to tangling. Even worse, mixing of dynamic and immutable components greatly limits caching efficacy as each have differing lifetimes. We extend AOUI design to push down concern separation to rendering, which reduces description size, through repetition reduction, and enables separate caching of individual concerns. Our results show considerable size reduction of UI descriptions for data presentations, faster load times and extended caching capabilities.**

I. Introduction

ENTERPRISE web applications have became common for domestic and international business in the last two decades. Users expect that enterprise applications support various browsing devices and provide attractive, usable, and fast UIs. Usability and speed often work in contradiction for web applications. Features to enhance usability often increase application size, which slows responsiveness, particularly in low bandwidth network such as those used for mobile access.

Despite clear expectations from UIs of enterprise web applications, conventional design approaches struggle from multiple deficiencies. For example, the UI descriptions for data presentations must restate information [18] from lower layers of the application, in order to extend them. This brings the risk of mistype errors caused by inconsistencies. Furthermore, modifications to the application data definitions require manual changes to the UI. The complexity is mostly evident when the UI description uses Domain Specific Languages (DSLs) [26] with limited type-safety. Furthermore, conventional approaches realize multiple UI concerns tangled together in a single component [5]. This not only limits component readability, but mostly limits its reuse, since such the component is strongly specialized. This “multi-concern component solution” results from the inability of conventional approaches to capture different concerns separately [33]. Such inability is also evident in object-oriented (OO) design [20]. Providing UI for given data in two slightly different situations (e.g., normal and mobile version of a website) may require to implement two similar UI components that differ only in details [5], [24], but requiring their separate maintenance. The UI development efforts are apparent from research [5], [18], which shows that approximately 48% of application code and 50% of development time is devoted to implementing UIs. This percentage grows with UI abilities and context-specificity.

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

Next, consider UI delivery to remote clients. In most cases, the UI is expressed as HTML and streamed to clients over the Internet. Although, supplemented with immutable resources such as images, stylesheets or JavaScripts the description itself is provided as a single piece of information. Such a single block of information has limited caching options and its size might be extensive. As stated above, UI components that present application data tangle multiple concerns together. This concern mix is also evident in the HTML streamed to a client. For example, an HTML data form mixes together field presentation, form layout, data binding, field validation, etc. Client web browser interprets the delivered HTML to present the UI description to the user.

Aspect-oriented design for UIs [5] reduces information restatement and supports separation of UI concerns for components presenting data [24]. The reduction of restatement is addressed through automated code-inspection [19] that supplies information for transformation to the UI. The aspect-oriented transformation involves integration of various UI concerns. This approach works at runtime and thus considers both static and contextual information. Each data presentation is assembled on demand, based on a given data instance. Concerns are captured individually and integrated based on contextual conditions. Individual concerns can thus be reused across different presentations and data. The outcome is significant UI code reduction and an assurance of correlation between the data definition and the UI presentation, which eliminates errors introduced by human factor [5], [9].

Since it is possible to capture UI concerns separately at the server-side, they can be also delivered individually to the client. The benefit at the server-side is the increase of concern reuse and thus UI code reduction. The delegation of the UI component assembly to the client-side could considerably decrease the amount of transferred information.

Efficient client-side caching of "tangled" HTML is complicated or even impossible. When only a single concern changes, the entire fragment must be transmitted again. Individual delivery of concerns to clients addresses reduction of replicated information in UI descriptions and makes it possible to cache certain concerns at the client. In this paper, we apply aspect-oriented UI (AOUI) design and research the impact of split concern streaming on the UI load time, transmission size and content caching. Our empirical results show considerable reduction of the UI description size, and the ability to cache individual concerns, which significantly reduces page load times for repeated visits. We evaluate our work by comparing the proposed approach with the conventional approach with respect to transmission content size, load time and caching.

The remainder of this paper is organized as follows: Section 2 describes the background of user interface designs. Section 3 provides an overview of existing work. AOUI approach is presented in detail in Section 4. Section 5 introduces distributed version of the approach. Its evaluation is discussed in Section 6. The final section presents our conclusion and future work.

II. Background

Enterprise system architecture [15], [22], [13] divides responsibilities into layers. For example the Java Enterprise Edition (Java EE) specification [10] divides the application into persistence, business logic and presentation layers. Developers implement each layer using a General Purpose Language (GPL). It is common practice for the presentation layer to use component-based UI approaches [2], which may involve a DSL to better describe a UI; unfortunately, such languages have limited type safety. Each layer has well defined responsibilities, and provides mechanisms to capture certain concerns. A concern [14] [20] can be understood as a set of information, which has some effect on the source code. For example, consider the concerns of data persistence, UI presentation, security, etc. Even though each layer has defined responsibilities and captures given concerns, there exist concerns that do not fit into a single layer but instead cross-cuts multiple layers. These cross-cutting concerns are responsible for tangled source code [20], and GPL languages, including OO, do not have mechanisms to effectively handle them [20], [21] so as to provide readability, maintenance or centralization. The result is that an individual concern, spreads throughout the source code and cross-cuts other concerns. Common examples of such concerns are exception handling, logging, and security as illustrated at Figure 1.

Aspect-Oriented Programming (AOP) provides an effective solution to this problem. [20], [21]. AOP suggests that, in addition to GPL components, there exists an additional concept

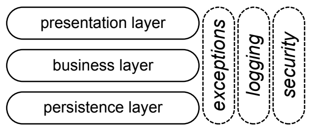


Fig. 1. Cross-cutting concerns in 3-layer enterprise system architecture

called an aspect. A particular program is then implemented using GPL and aspects. An aspect captures cross-cutting concerns separately from the GPL components. The way GPL components are connected with aspects is the main AOP contribution. An aspect consists of two parts: pointcut and advice. The pointcut specifies a situation, location or context under which the aspect is woven into the GPL component. The advice is the concern definition specified either in GPL or a custom DSL. In order to effectively address location in GPL components, AOP defines the concept of join-points. A join-point can range from code location specified by name or a wildcard, method invocation based on method name, annotation, or even a particular application context. Naturally, we can divide join-points into static and dynamic, with the difference based on whether they are activated only by location in the code or whether a runtime condition must hold to activate it. An example can be seen in enterprise systems when handing security with an AOP approach. When a user invokes an action from the UI, this action goes through an action controller [22], which has a method to implement the action. Often, this method has a security annotation determining user access, such as a user security role. Before the actual method is called, the security annotation acts as a join-point that activates a security aspect. This join-point is specified in the security aspect point-cut, and its advice looks into the application context to determine whether the actual user is logged into the system and whether he/she has sufficient permission, given by the security annotation, to be eligible to call the action. If not, the advice throws a security exception; otherwise it delegates the call to the controller method. The same security aspect applies in the UI to determine whether or not to render a given action button for a particular user, etc.

Tangled concerns can be found in the UI as well [5], [4]. Conventional design approaches do not address them separately but together in a single source code. This is directly responsible for low code reuse and readability as well as for high development and maintenance efforts. Various concerns that play a role in the UI can be considered independently as shown by Figure 2a. Unfortunately, because of limited GPL and DSL constructs, all concerns must collapse together into a single UI component, a single source code as depicted at Figure 2b. This results with tangling of all involved concerns, which makes individual concern localization difficult. Consider the concern collapse in Figure 2b at the sample code in Listing 1 while considering concerns from Figure 2a. Its graphical representation sketch is shown in Figure 3. The resulting source code is very specialized with limited reuse. At the same time, we must consider that such UI code restates information from the data definition [19], which introduces interdependencies that must be maintained. Because of limited

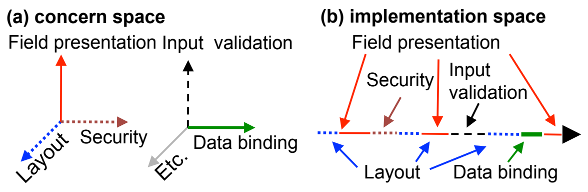


Fig. 2. (a) Concerns as orthogonal dimensions /
(b) Implementation space in a single dimension with tangled concerns

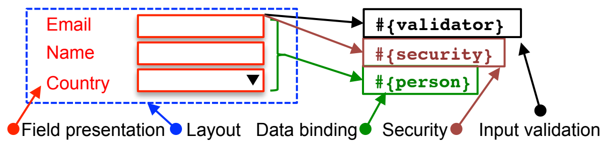


Fig. 3. Graphical sketch of Listing 1 denoting concerns from Figure 2

type safety in DSLs, it is easy to introduce an error while restating information [9]. The situation becomes worse when multiple presentations exist for given data [4]. The limited reuse forces us to maintain multiple, very similar components. Moreover, when we consider conventional approaches and aim to design adaptive or context-aware UIs, we end up with even more similar UI components that we must manage and update each time the data change [24].

The AOP approach for UIs [5] considers the data definition to be the GPL component that is being presented in the UI. In order to present data in UI, it considers individual UI concerns (e.g., Figure 2a) and weaves them together at runtime upon request to assemble the UI presentation. Such data presentation reflects the actual data definition and the particular application/user context. Data definitions, normally OO classes, define data fields and their constraints that act as static and dynamic join-points for the "data to UI" transformation. Dynamic join-points are further extended with the application context. A particular base presentation for each data field is selected based on these join-points. Subsequently, this base presentation is extended with field-level concerns through various aspects. Once all data fields have determined the presentation integrated with other concerns, the layout is woven through the fields. The resulting component reflects the data definition, context and considered concerns. Thus issues, such as information restatement or multi-component management, are eliminated. The result is that no physical UI component for data presentation exists; they are assembled on demand where each concern can be maintained and vary individually based on the user/application context. The advantage comes when we want to present novel data in the UI. All concerns are be reused and thus scaling-up the data model size does not impact the UI concern size or the UI management. What impact the size of the individual concern space are custom presentation layouts for given data, although it is possible to design generic set of layouts reusable among data.

When we consider client-server communication over HTTP, we must be aware that all the concern assembly to receive data presentation takes place at the server-side. Basically, at the server-side all concerns tangle together through an aspect-

Listing 1. Sample source code for UI form reflecting Figure 2 (b)

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

weaver that produces UI descriptions, eventually translated to HTML. This tangling may produce large and complex HTML, containing repetitive information impacting the content size. The server transmission outcome of conventional approaches is similar, if not the same, to the outcome of AOP-based UIs. While compression as well as caching of static resources can be applied, there are two issues. First, compression, although addressing repeating patterns, is not aware of the content logical structure, and thus it addresses small repetitive fragments rather than large concerns. Second, it does not allow to cache immutable information occurring in the delivered UI description. In this work, we show that it is possible to improve the transmission content size and caching with AOP-based UIs. Since AOP has constructs to separate individual concerns, it is possible to stream separately to the client and let the client perform the assembly at the client-side. This reduces the repetitive information from the transmission and, at the same time, an individual decision on concern caching and reuse can be made. Such changes in the UI delivery impact the UI transmission and presumably reduce the delivery time.

III. Related work

A. UI design approaches

Various approaches have been introduced to simplify development of complex UIs. These approaches 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 UI maintenance or complex situations, such as context-based UIs adapting during the runtime. In terms of client-server communication, conventional approaches transfer large, perhaps unnecessary, amounts of data, which negatively impacts the communication and response times.

Model-driven development (MDD) [29] suggests that a model is the source of information, and the resulting source code is generated using this model together with a set of transformation rules. The main advantage should be reduction of information restatement that must be handled manually [9] for different perspectives. In [23] MDD is applied to distributed UIs, with description of a workflow that uses a task-centred approach described through the Concur Task Tree (CTT) notation [1]. This allows the description of environment and given context. MDD can handle multi-context UIs, for example, in [3] authors split the context into user, platform and environment parts. At the same time when we aim to describe different concerns through multiple models, MDD does not provide any standard integration mechanisms to do so [31]. Sottet et al. [31], [32] provide a deep explanation of model-to-code and model-to-model transformations relevant to UI MDD. Although, MDD can be used to capture complex UIs, various contexts, and adaptive features, the model-to-code transformations may struggle in the performance perspective [24]. Transformations are usually performed at compile time as they tend to be time consuming [24]. Compile-time transformations produce source code and descriptions for all possible context states, which might not be fulfilled by the user [27]. Next, the transformation takes place at the server-side, and thus the result may contain tangled UI descriptions possibly containing repetition. The MDD-based UI design may struggle from further issues. In [27], the authors observe that it suffers during adaptation and evolution management. Such design handles base situations well, although when context variations or customizations are needed, the modification often take place in the UI code [9] rather than in the model. Manual changes are lost the next model-to-code transformation, which then leads to difficult maintenance [9]. Another issue, presented by [5], arises when the MDD applies solely to the UI but not to other parts of the system, such as persistence or business logic subsystems. In such cases, information captured by the model must match to information captured by the rest of the system. When only one part of the system changes, another part may lose compatibility and may need to address the changes manually. Such an approach is unfortunately very common in the research discipline of human-computer interaction [24].

The use of DSLs [26] for the UI model description is common, although, DSLs tend to provide weak type safety, which extends maintenance efforts, since information change propagation becomes tedious and error-prone in a manual process. DSLs are often used to directly specify UIs [26] [17]; a practical example is the Java EE standard for component-based development called JavaServer Faces (JSF) [2]. The DSL brings simplification to the UI description [5], as oppose to GPL. It is transformed to the target UI language, such as HTML. DSLs naturally fit to UI descriptions, but through their weak type-safety, it is easy to introduce errors related to restated information from lower application layers [18]. For example a DSL description may reference data, their fields and constraints that are already described in the application through GPL [12]; however, referencing a GPL component from DSL requires certain restatement with a negative impact on maintenance. Similarly to MDD, the DSL-to-native code transformation takes place at the server-side, thus the transmission streams the produced tangled code.

Another approach addresses information restatement by utilizing code-inspection and meta-programming [18], [5], [24]. It inspects data GPL definitions and from the result composes a structural model. This model is transformed to UI descriptions with all data/constraint references resolved through the model; this avoids human-errors related to inconsistencies. The output can be in the form of DSL, such as JSF. In comparison to the above approaches, this one works at runtime, although, it does not address cross-cutting UI concerns. Similarly, the product generated at the server-side is not different from the product produced through DSLs or MDD.

One possible solution that addresses tangled code and cross-cutting UI concerns is Generative Programming (GP) [11], [30], which emphasizes domain methods and integration with GPL. GP can be seen as programming that generates source code through generic code fragments or templates. 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, although, happens at compile time. The use of GP for UI is demonstrated at [30] through abstract UI specifications. An application that uses this 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 takes the DSL specification and assembles implementation components from them and from the available components. Such an approach allows production of a large number of system variants for specific requirements. In a presented case study, a system combined two hundred features in the UI, with a resulting variability of 5×10^{17} prototypes. It is questionable whether such a large number of feature components is reasonable and could be ever used, although all states are physically generated at compile time and statically allocated. The nature of the compile-time assembly makes it hard for use with future adaptive systems that need runtime information to base its decisions on [27].

The AOP approach has been applied to extend capabilities of existing approaches. In [27] 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 should be considered to influence model-to-code transformation. As mentioned in [32] the MDD runtime transformation might be performance inefficient for complex situations [24]. Some suggest that MDD UI transformations may generate all possible application states and configurations for hypothetical/possible situations. In complex systems, this can grow exponentially. Also, MDD-based systems suffer and become impractical in evolution management of system adaptation. [27] thus suggests using four runtime models that represent main system data that are manipulated at runtime. These models are responsible for system runtime adaptations and generation of application components. The work describes the aspect-oriented conceptual model [33], weaving process and context very sparsely, and no performance consideration is given to the manifest approach effectivity for production systems. Alternative aspect-oriented UI design, based on conventional UI designs and enabling both code inspection and separation of concerns for data representations, is given at [5]. Similar to inspection-based approaches, meta-programming determines a structural model (join-point model). Subsequently an aspect-oriented transformation of the model to the UI enables integration of various, separately defined, UI concerns. Although, the aspect weaving happens at runtime, it takes place at the server side and thus the UI transmission to clients is no different from the above approaches.

As shown above, existing research in UI fails to address effective UI transmission to clients or optimization of client-side caching. One of the contemporary UI frameworks, although, does address caching by compiling the GPL UI descriptions to client cacheable resources. The Google Web Toolkit (GWT) [16] suggests describing the UI in the type-safe Java language and compile it to a JavaScript (JS) UI description. Note well, that even GPL UI description consists of restated information from application lower layers, such as data field descriptions and their constraints. For example, to design a UI representation for a given data field, the developer must select an appropriate component, bind it with the field and manually restate the component constraints already defined at the field, through annotations [12]. As mentioned in [33] and [20], GPL languages do not effectively handle cross-cutting concerns; consequently, GWT tangles them together. The GWT produces a JS UI description at compile time, which is similar to the MDD approach. It produces multiple versions of those descriptions to support various end-devices. It consists of code fragments that can be cached as well as these that cannot. GWT struggles from the same disadvantages as MDD, and thus it does not fit to adaptive UIs. For instance a UI page that presents many context-based variations compiles all possible states to a single description and ships it to the client not matter whether the user uses a single UI state or multiple. This becomes obvious with complex adaptive systems [27] with combinations of states and configurations that grow exponentially. In our work, we suggest transmitting UI concerns separately to clients. This allows transmitting only the actual state needed by the client, and at the same time, each concern may change individually, avoiding exponential grow.

B. UI delivery to the client

The standard client-server communication for web systems is based on the HTTP protocol that provides the core mechanisms to improve the transmission. First, the TCP-based protocol supports connection persistence so multiple resources can be loaded from a single server. Connections are reused, rather than reopened, which requires additional overhead. Multiple connections may exist from a client to a single server. HTTP supports content compression to reduce the transmission content size. Furthermore, it supports content caching at the client-side with time-based invalidation. The caching applies mostly to static resources such as CSS, images, and JS. In [7] authors show that the average contemporary web systems consists of about 90% static and cacheable resources. To further reduce the transmission, UI developers may apply content obfuscation and resource merging [6]. To mitigate the impact of client distance, servers often apply geo-distributed caching of static resources called content-delivery networks (CDNs), such as Akamai [28].

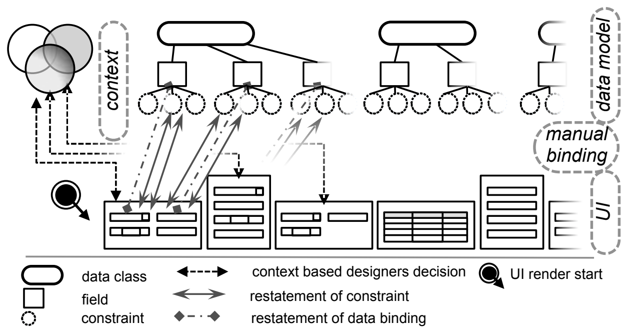
Structured Hypertext Transfer Protocol (STTP) [34] extends HTTP to include new messages to control the resource transmission for a particular web page. A similar approach, HTTP-MPLEX [25], employs a header compression and response encoding scheme for HTTP. Similar to STTP, it multiplexes multiple responses to a single sustained stream of data to speed response times and improve application layer use of TCP. While experiments show performance improvements with these protocols, they do not consider resource distribution through CDNs, caching, or variations. Another optimization approach is brought by cooperative-web cache [7], [8]. It involves clients with cached resources in participation in an overlay peer-to-peer network, which allows clients to share these resources in the overlay. Unlike CDNs it supports natural scalability and free P2P services; however, it must deal with content invalidation in the overlay and mechanisms to disable and prevent malicious clients from sharing corrupted data.

IV. AOP-BASED UI ASSEMBLY

The development and design approach of aspect-oriented UIs (AOUI) [5] is considerably different from conventional approaches. In order to describe the AOUI design, we first illustrate the UI design with conventional approaches and describe its dependencies to data definitions. Next, we sketch the AOUI design differences and describe it genericity and relaxed dependencies.

Figure 4 demonstrates the conventional UI design with the data model at the top with individual data classes with fields and given constraints as well as application context. The bottom presents a sketch of various presentations of given data. These presentations are rendered in the application based on the context. Each presentation has physical code representation and consists of multiple elements that bind to a particular data field or its constraints. The UI presentation has to restate field names as well as the constraints in its source code, which increases UI component coupling. Each such component is specialized to display specific data, and we can hardly assume that such UI component could be used for another data class. Such coupling must be seen from the perspective of system maintenance, thus when a given data class changes, for example a new field occurs, we must manually reflect it in related UI components. When the UI component description uses DSL, limited type safety may not provide any enforcement mechanism on the compatibility with data. We can summarize the disadvantages as follows: no automated data change propagation to the UI, limited type safety does not prevent human errors, limited separation of concerns and limited reuse, data class presentation requires to design a custom UI component. To see the difference with the AOUI design, consider the case when we aim to render data in the UI. In order to do that, we need to know the physical location of a particular UI component (denoted via UI render start mark at Figure 4), and this components then uses its configuration to cooperate with the data class, and its fields based on matching names.

Next, consider the AOUI design in the same illustration of data classes and the application context at the top and data presentations at the bottom Figure 5. First, note that with AOUI


context
data model
manual binding
UI
data class
field
constraint
context based designers decision
restatement of constraint
restatement of data binding
UI render start
Fig. 4. UI design with conventional approach

there is no physical location of the UI presentation, because it does not exists. Instead it is generated on request by passing a data instance reference to an aspect weaver (denoted via start mark in Figure 5). The AOUI aspect weaver inspects the given instance data class, denoted by ①, and produces its structural (join point) model for given data. This model is created once upon the first use and consists of class and field information and their constraints. Upon each use, a clone model is made and modified/restricted according to user context. For example fields not relevant to given context in the UI are eliminated or restricted based on security rights. Additional elements from the application context can be exposed to this model as well. Each model element then acts as a join point for the subsequent transformation stages. In the next stage ②, the aspect weaver considers generic mapping rules that select a presentation for each particular data field based on its properties in the structural model. These rules are not specific for a given data class or a field; instead they only bind to model elements, which may occur at any data field. This brings genericity and reusability among fields and data classes. These rules become reusable among data classes or even among different systems. Each rule consists of two parts, a specification of structural model elements in a query (a point-cut) and an advice in the form of a presentation template. A rule applies to a field, which has a given constellation of specified elements. For example a query specifies a text-typed field with maximum length greater than 255 letters. The selected rule then applies an advice that selects a presentation template for the field matching the given query. This presentation template consists of a description for a basic field presentation in the target UI language. It also contains extensions in the description through which it is possible to integrate other aspects to it, providing concerns weaving. Each template aspect consists again of a point-cut that uses the same constructs as mapping rules referencing the structural model and advising how to integrate the concern. Often it embeds selected structural model element values to the output. In the stage ③, after all data fields have resolved presentation, a layout template is selected based on the context and integrated to the field presentation. This results with stage ④ that provides the presentation for given data instance and current application context and renders it to the UI. The most important benefit is that rules and presentation templates are generic and not dependent on specific data, which allows us to scale-up the

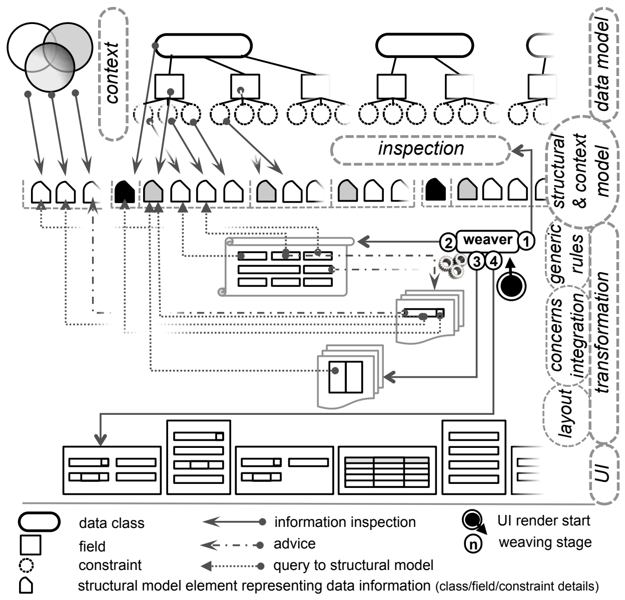


Fig. 5. UI design with AOUI

data model without an impact on the size or efforts related to the UI. In other words, a new data class passed to the AOUI aspect weaver is displayed based on the existing rules and templates. Data changes do not cause inconsistency in the UI because the change is reflected in the ad-hoc structural model that influences the selection of a given transformation rule as well as it subsequent aspect integration in the presentation template. Novel constraints apply to the UI according to the their occurrence in templates. In the [5] authors show that among 63 data classes in a production system with about 473 fields, only 28 transformation rules and presentation templates are needed. They are reused, and various concerns integrate to it based on given context. The benefits can be summarized as: Code volume reduction, constraint enforcement, separation of concerns, data independence, concern reuse, reduction of restatement. Furthermore, it is easy to integrate new concerns and thus support context-aware or adaptive UIs while not introducing complexity to the UI design.

V. Distributed AOP-based UI assembly

Conventional design approaches stream the UI description as a single block of information. The AOUI design untangles UI concerns for components reflecting data and reduces development and maintenance efforts. Such weaving takes place at the server-side and its product, with weaved-concerns, is streamed to clients. This is equivalent to conventional approaches. One may assume that content compression over HTTP solves the inefficiency, although with no doubt, it does not improve caching options. To the contrary, consider a solution where concerns (such as these from Figure 2 a) are streamed separately to clients. This might seem as an addition overhead as we need to handle multiple connections. On the other hand, this may eliminate repeating patterns in the transmitted content and enable caching for given immutable concerns.

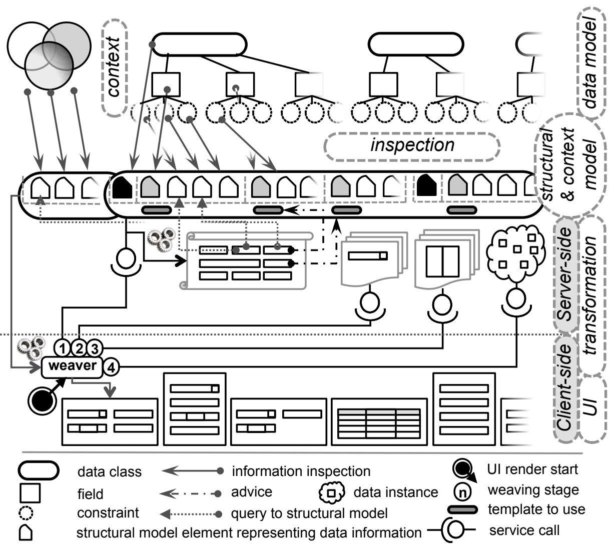


Fig. 6. UI design with dAOUI

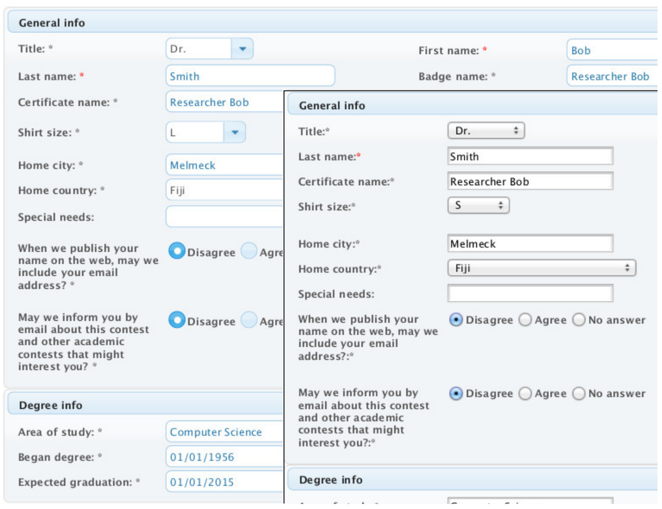
In order to design distributed AOUI (dAOUI), the process of concern weaving should be partially pushed to the client-side. The application data model (or data transfer objects [15]) and application context are part of the server-side, and thus the inspection part takes place there. This gives a structural and context model, which could be streamed to the client. Some model elements might not be relevant to presentation or to a particular user. For example, there might be internal fields such as primary key, version field, etc., and these might not be relevant to the UI or to a particular user rights/tasks. AOUI handles this through data model constraints as well as through the context. To provide a particular user only the model elements relevant to his session and rights, the context is applied to the requested subset of structural model. This eliminates elements that would not apply for the UI composition, and this subset is streamed to the client. The selection of a particular presentation template for given data field could be done at the client-side. This increases the complexity of the client weaver since it must be aware of transformation rules and these may need to have access to internal server-side information to resolve rules to and make a decision. Thus keeping this responsibility at the server-side and providing the result to the client reduces the client weaver design complexity. Figure 6 depicts the responsibility assignments between server and client sides through service calls. Each client needs to have access to the structural and context model for the given data it represents in the UI ①. It also needs access to presentation templates ②. The decision on template selection made on the server-side is delivered together with the structural and context model ①. The client uses a particular presentation template suggested by server. Each template content is resolved towards the structural and context model of a given data field. Layouts ③, similar to presentation templates, are provided to the client-side for integration to the data UI presentation. Other concerns might be provided as separate services and integrated either at the server-side through transformation rules or via client-side presentation templates. Each client composes the UI data component based on provided concerns that are influenced by system context. The server also provides the actual data instances to the client ④. These data are displayed in the assembled UI component. The data submission uses HTTP POST or GET mechanisms or a web service.

The life-cycle for the web systems works as follows. First, the user requests navigation to a particular page or a dialog. This page consists of description elements from conventional UI design. The difference is for components representing data. Such components are replaced by custom tags interpreted at the client-side (for example a JS call). The tag indicates which data to display and what settings (context) should apply for the UI component assembly. Such content, with no data physical representation in it, is transmitted to the client-side. When client interprets the delivered content, it interprets it ordinarily. Custom tags are interpreted through a client-side weaver that requests given concerns from the server-side. Provided responses consider user rights and security. As depicted at Figure 6, the weaver assembles the UI representation of a data instance (given by the custom tag) conforms the structural model, application context and settings provided together with the data reference. The weaver may either request a particular concern from the server-side or may reuse it from its cache. For example, presentation templates will hardly change throughout a long period of time, or given data fields are immutable in given context over the time or throughout a user session/conversation.

VI. Evaluation

In order to evaluate our approach, we consider existing production level enterprise web application based on Java EE 6 platform with JSF [2] framework for the UI design. For our evaluation, we consider a subsystem for user account management. We evaluate the existing solution regarding data transmission, page load time and caching. Next, we implement the same subsystem with dAOUI design. Specifically, we design REST services at the server-side and a JS library responsible for UI component assembly, interacting with these services. These services include a service to obtain structural and context model for given data, a service to obtain the actual data from a given instance conforming system security and finally a service to handle data manipulation from the client-side. Next, we provide a JS package with presentation and layout templates and the client-side aspect-weaver. The dAOUI prototype is evaluated using the same criteria.

Figure 7 at outer left shows a sample UI subsystem considered in the evaluation. The same result can be designed with both conventional and AOUI approaches [5]. The difference relates solely to the server-side design; the content transmitted to the client-side is equivalent for both approaches. The dAOUI prototype is shown at inner right of Figure 7. The differences are that the conventional UI uses JSF, which is transformed to HTML through the framework and transmitted to the client-side all together. The second UI consists of JSF for the page without components representing data. Instead it uses a JS library that interacts with the server REST services



General info

Title: * Dr.
First name: * Bob
Last name: * Smith
Badge name: * Researcher Bob
Certificate name: * Researcher Bob
Shirt size: * L
Home city: * Melmeck
Home country: * Fiji
Special needs:
When we publish your name on the web, may we include your email address? * [✓ Disagree] Agre
May we inform you by email about this contest and other academic contests that might interest you? * [✓ Disagree] Agre

Degree info

Area of study: * Computer Science
Began degree: * 01/01/1956
Expected graduation: * 01/01/2015

General info

Title:* Dr.
Last name:* Smith
Certificate name:* Researcher Bob
Shirt size:* S
Home city:* Melmeck
Home country:* Fiji
Special needs:
When we publish your name on the web, may we include your email address?:* [✓ Disagree] Agree No answer
May we inform you by email about this contest and other academic contests that might interest you?:* [✓ Disagree] Agree No answer

Degree info

Fig. 7. Evaluated UI subsystem designed with conventional/AOUI approach in the outer left position, the dAOUI approach in the inner right position

through JSON format to assemble the data presentation components and embeds them to the UI. In both cases UI panels, controllers and page navigation logic are equivalent.

A. Network transmission and page load times: The base case

To evaluate network transmission size and page load times, we use the UIs described above. In the evaluation of page loads, we consider complete page rendering. We must emphasize that both UIs utilize equivalent static web resources (JS, CSS and images). These same resources are part of the production system, so with the outcome we can consider a realistic impact on a production system. The dAOUI version has additional JS libraries, but on the other hand, the initial HTML page does not contain descriptions for data components. Instead, it consists of a JS initiation calls to assemble data components based on settings given in the HTML page. The considered data components at the page at Figure 7 consist of 23 fields of various data types given by the production system.

The conventional approach page produces 1458 KB to render the UI, although with HTTP compression the transmitted content reduces to 329 compressed KB (cKB). The main HTML document is 86 KB (11.1 cKB), and the rest 1372 KB are static resources. To download and render the UI page with compression takes 2.3 sec (average over 10 samples with standard deviation $\sigma = 0.23$). The download uses gzip compression with no network restrictions. The dAOUI page produces 1368 KB (311 cKB). The main HTML document has the size only 2.9 KB (1.2 cKB); additionally there is 10 KB (3 cKB) of JS and four calls to REST services with a total size of 13.6 KB (4.9 cKB). The page load reduces to 1.73 sec ($\sigma = 0.14$), an almost 0.6 second (or 25%) time reduction.

Next, we consider caching. All static resources are cached at the client-side. The conventional approach page with cached resources requires 86 KB (11.1 cKB) to be loaded from the server, and the page load time reduces to 1.67 sec ($\sigma = 0.24$). The dAOUI allows us to cache the weaver, presentation and layout templates, as well as the data structure, which is immutable. This reduces the transmitted content down to 3.9 KB (2.1 cKB), and the page load time needs only 0.79sec ($\sigma = 0.07$), representing reduction of almost 0.9 sec, which is less than 50% of the original wait time. The summary can be seen from Table I.

TABLE I
BASE EVALUATION CASE, TRANSMISSION OF UIS WITH 23 FIELDS

No network throttling	No-cache		Cached	
	Size (KB)			
		Compressed		Compressed
Convent. approach	1458	329	86	11.1
Distrib. AOUI	1386	311	3.9	2.1
	Load time (using compression)			
	(sec)	(relative)	(sec)	(relative)
Convent. approach	2.3	1	1.67	1
Distrib. AOUI	1.73	0.75	0.79	0.47

TABLE II
EXTENDED EVALUATION CASE, TRANSMISSION OF UIS WITH 42 FIELDS

No network throttling	No-cache		Cached	
	Size (KB)			
		Compressed		Compressed
Convent. approach	1484	331	110	13.9
Distrib. AOUI	1394	315	6.9	3.8
	Load time (using compression)			
	(sec)	(relative)	(sec)	(relative)
Convent. approach	2.54	1	1.99	1
Distrib. AOUI	1.89	0.74	1.01	0.51

B. Increasing the UI size: The larger case

The production system on which our study is based has the person account page considerably larger than what we considered above. Next, we consider the impact related to data size extension. The extended UI has 42 fields at the page.

The conventional approach page extends to 1484 KB (331 cKB) out of which 110 KB is the HTML document (13.9 cKB). The page load time is 2.54 sec ($\sigma = 0.33$). The dAOUI size has in total 1394 KB with the HTML document 5 KB (1.4 cKB) and JSON calls 19.3 KB (8.4 cKB). The compressed transmission size has 315 cKB. The page load time for dAOUI is 1.89 sec ($\sigma = 0.17$), representing a reduction of 0.65 sec, which is similar to the previous evaluation with reduction to less than 75% compared to the conventional approach.

The cached-enable evaluation of the conventional approach design consists of the total transmitted size of 110 KB (13.9 cKB) with load time 1.99 sec ($\sigma = 0.38$). The cached dAOUI has 6.9 KB (3.8 cKB) and a load time of 1.01 sec ($\sigma = 0.15$). Similar to the previous evaluation, the reduction of wait time is almost 1 second and represents almost 50% of the load time. The summary can be seen in Table II.

C. Throttling the network: 3G/DSL users case

The next evaluation throttles the network conditions to evaluate behavior for both mobile users with a 3G network and DSL users. For the 3G evaluation, we restrict the network bandwidth to 384 kbit/s and set network delay to 20ms. Such network restrictions allow us to emulate network conditions for users with mobile devices. We evaluate the base page with 23 fields. The load time significantly grows to a barely usable system. The page using the conventional approach requires

TABLE III
3G AND DSL CASE, TRANSMISSION OF UIs WITH 23 FIELDS

	Load time (using compression)			
	(sec)	(relative)	(sec)	(relative)
384kbit/s 20ms delay	*No-cache*		*Cached*	
Convent. approach (sec)	18.89	1	2.72	1
Distrib. AOUI (relative)	15.88	0.84	1.07	0.39
768kbit/s 10ms delay	*No-cache*		*Cached*	
Convent. approach (sec)	9.45	1	1.79	1
Distrib. AOUI (relative)	8.28	0.88	1	0.49

TABLE IV
GWT COMPARISON, TRANSMISSION OF UIs WITH 23 FIELDS

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

18.89 sec to load ($\sigma = 2.16$). The dAOUI requires still a long time 15.88 sec ($\sigma = 0.57$). The reduction represents 3 sec, which is around 85% of the original load time. Caching becomes a "must have" for these kinds of mobile users. With caching, the conventional page loads within 2.7 sec ($\sigma = 0.42$) compare to the dAOUI with 1.07 sec ($\sigma = 0.19$). This represents a reduction of 1.6 sec, which is less than 40% of the original load time. The summary is at the top of Table III.

Next, we evaluate the situation for DSL users with bandwidth to 768kbit/s and delay 10ms. The conventional page needs 9.45 sec to load ($\sigma = 0.84$). The dAOUI reduces the load time by 1.1 sec, down to 8.28 sec ($\sigma = 0.36$). The reduction in percentage is similar to the 3G version and represents slightly less than 85% of the original load time.

Caching significantly helps for consequent page loads. The conventional page loads within 1.79 sec ($\sigma = 0.24$) compared to the dAOUI with 0.87 sec ($\sigma = 0.08$). The dAOUI is 0.9 sec faster and less than 50% of the load time. The summary can be seen at the bottom of Table III.

D. Comparison with GWT

The GWT introduced in the related work targets improvements to UI caching. To compare it with our approach, we implemented the 23-field prototype application with GWT. Note well, that the evaluated dAOUI page at Figure 7 is based on a production system, while the GWT is just a prototype. This results with differences in both prototypes regarding linked JS libraries. While the dAOUI prototype links JS resources related to given JSF component provider, the GWT prototype does not link to any generic JS library. Although, this does not impact the caching statistics, we modify the dAOUI prototype as follows: We reduce the linked JS libraries and only consider libraries related to the functionality of the dAOUI, which makes it equivalent regarding the comparison with the GWT prototype. The dAOUI prototype needs to transmit 161 KB of data to build the UI at the client (41.8 cKB). The GWT version needs 379 KB (102 cKB). The main document in GWT is converted to into JS with cacheable fractions with 141KB (50 cKB) and non-cacheable fraction with 7.2 KB (3.4 cKB).

The cached-enabled evaluation stays the same for dAOUI with transmitted content 3.9 KB (2.1 cKB). The GWT version needs to download the HTML page, displayed data and the non-cacheable JS fragment, which is in total 11.9 KB (5.8 cKB). The results are summarized in Table IV. From the results, we see that UI construction in the form of JS can considerably improve caching at the client-side. The JS presents tangled code through mixed concern, which extends its size. Separately streamed UI concerns can reduce the UI description, and improve caching.

E. Summary

Streaming various concerns separately from server to clients brings reduction to the overall transmitted content, page load times (considering complete UI rendering) and improved caching. In the evaluation, we streamed presentation and layout templates, data structure with applied security as well as the actual data. The dAOUI enables to use cache for concerns that are normally tangled together in conventional approaches. The study on a production system shows the reduction of content size in the range of tens of KBs even when compressed. Even though the dynamic content represents only around 6% (3.5% compressed) of the total content, the transmission size of uncached UI content reduced in total by 5%. With caching that strips the static content, it reduces transmission by 72-81% compare to JSF. With GWT the transmission content with cached resources improves by around 63%. The dAOUI managed to reduce the page load time to the range of 75-90% of what it takes with a conventional approach. This turns even better with caching, which gives reduction in the range of 40-50% compared to the conventional approach. The exact reduction is, although, influenced by many factors including network conditions and the UI itself.

In our evaluation, we reduced the transmitted content sometimes by an order of magnitude compare to the JSF approach. While the GWT approach compiles the UI into a JS and provides a solution with extended caching capabilities, such a solution can be further improved with dAOUI.

VII. CONCLUSION

In this paper, we suggest an alternative design approach for presentations of data in enterprise software systems. Conventional designs mix various concerns together, which results in code that is hard to maintain and reuse. We show that some of the UI concerns do not change over the time, and we may cache them at the client-side to reduce network traffic. Conventional designs fail to offer this ability, so given concern might be tangled with others without the possibly of variability and reuse. AOUI design separates concerns for UI components presenting data and thus reduces maintenance efforts as well as improves reuse of these concerns. With an extension of such design, it is possible to deliver concerns separately over the network to clients and delegate the component assembly to the client-side. This may reduce the transmitted content size that needs to be delivered to the client over the network, but mostly it enables caching and reuse of specific UI concerns at the client-side. From a case study based on a subset of a production system, we provide results showing a reduction of the total transmission size from the server to the client and the page load time. Furthermore, we extend capabilities for caching of UI concerns at the client-side, which further reduces the data transmission as well as page load times.

Although the results from the study are promising, we must consider its limitations. The design approach fits well for data presentations; it builds on the top of other approaches that deal with interaction, page-flow, etc. AOUI easily adapts to development standards and allows integration of third parties for security, context-awareness, etc. The approach does not aim for complete design of UI pages but targets only data presentations. Existing approaches provide a large palette of various field components, suggestion boxes and data manipulations. With this approach, it is necessary to design them as no component library exists. On the other hand, it is easy to integrate HTML5 components or custom presentations for fat-clients. The interaction is not limited to only frontend and backend parts of systems, but it possible to consider middleware communication. Considering AOP's natural ability of concern separation, the design fits well to context-based UIs. At the same time, the AOP development pushes towards different development practices, and designer transition from conventional habits might be difficult. In addition there is a lack of tool and framework support as well as a missing standard for AOUI.

In future work, we aim to extend concerns with business rules integration. Our preliminary work [4] shows that it has large potential. We also look at integration of aspect-oriented design to service oriented architecture (SOA).

REFERENCES

[1] S. Berti, F. Correani, G. Mori, F. Paternò, and C. Santoro. Teresa: a transformation-based environment for designing and developing multi-device interfaces. In *CHI'04 extended abstracts on Human factors in computing systems*, pages 793–794. ACM, 2004. http://dx.doi.org/10.1145/985921.985939.

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

[3] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003. http://dx.doi.org/10.1016/S0953-5438(03)00010-9.

[4] K. Cemus and T. Cerny. Aspect-driven design of information systems. In *SOFSEM 2014: Theory and Practice of Computer Science, LNCS 8327*, volume 8327, pages 174–186. Springer International Publishing Switzerland, 2014. http://dx.doi.org/10.1007/978-3-319-04298-5_16.

[5] T. Cerny, K. Cemus, M. J. Donahoo, and E. Song. Aspect-driven, data-reflective and context-aware user interfaces design. *Applied Computing Review*, 13(4):53–65, 2013. http://dx.doi.org/10.1145/2513228.2513278.

[6] T. Cerny and M. J. Donahoo. Performance optimization for enterprise web applications through remote client simulation. In *Proc. of the 7th EUROSIM Congress on Modelling and Simulation, Prague, CZ*, volume 2. CTU, Prague, 2010.

[7] T. Cerny, P. Praus, S. Jaromerska, L. Matl, and J. Donahoo. Cooperative web cache. In *Systems, Signals and Image Processing (IWSSIP), 2011 18th International Conference on*, pages 1–4. IEEE, 2011.

[8] T. Černý, P. Praus, S. Jaroměřská, L. Matl, and M. Donahoo. Towards a smart, self-scaling cooperative web cache. *SOFSEM 2012: Theory and Practice of Computer Science*, pages 443–455, 2012. http://dx.doi.org/10.1007/978-3-642-27660-6_36.

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

[10] R. Chinnici and B. Shannon. JSR 316: Javatm platform, enterprise edition 6 (java ee 6) specification, Dec 2009.

[11] K. Czarnecki and U. W. Eisenecker. Components and generative programming. In *Proc. of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT intl. symposium on Foundations of software engineering*, ESEC/FSE-7, pages 2–19, London, UK, UK, 1999. Springer-Verlag. http://dx.doi.org/10.1145/318774.318779.

[12] L. DeMichiel. JSR 317: JavaTM persistence API, version 2.0, Nov 2009.

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

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

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

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

[17] M. Karu. A textual domain specific language for user interface modelling. In T. Sobh and K. Elleithy, editors, *Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering*, volume 151 of *Lecture Notes in Electrical Engineering*, pages 985–996. Springer New York, 2013. http://dx.doi.org/10.1007/978-1-4614-3558-7_84.

[18] R. Kennard, E. Edmonds, and J. Leaney. Separation anxiety: stresses of developing a modern day separable user interface. In *Proc. of the 2nd conf. on Human System Interactions*, HSI'09, pages 225–232, Piscataway, NJ, USA, 2009. IEEE Press. http://dx.doi.org/10.1109/HSI.2009.5090983.

[19] R. Kennard and J. Leaney. Towards a general purpose architecture for ui generation. *Journal of Systems and Software*, 83(10):1896 – 1906, 2010. http://dx.doi.org/10.1016/j.jss.2010.05.079.

[20] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *ECOOP'97-Object-Oriented Programming, 11th European Conf.*, volume 1241, pages 220–242. Springer, June 1997. dx.doi.org/10.1007/BFb0053381.

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

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

[23] K. Luyten, C. Vandervelpen, J. V. den Bergh, and K. Coninx. Context-sensitive user interfaces for ambient environments: Design, development and deployment. In *Mobile Computing and Ambient Intelligence: The Challenge of Multimedia*, Dagstuhl, Germany, 2005.

[24] M. Macik, T. Cerny, and P. Slavik. Context-sensitive, cross-platform user interface generation. *Journal on Multimodal User Interfaces*, pages 1–13, 2014. http://dx.doi.org/10.1007/s12193-013-0141-0.

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

[26] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005. http://dx.doi.org/10.1145/1118890.1118892.

[27] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, Oct. 2009. http://dx.doi.org/10.1109/MC.2009.327.

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

[29] J.-l. Perez-medina, S. Dupuy-chessa, and A. Front. A survey of model driven engineering tools for user interface design. In *Proc. of 6th Int. workshop on Task Models & Diagrams (TAMODIA'2007)*, pages 84–97, Berlin, 7-9 Nov. 2007. Springer. dx.doi.org/10.1007/978-3-540-77222-4_8.

[30] M. Schlee and J. Vanderdonckt. Generative programming of graphical user interfaces. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '04, pages 403–406, New York, NY, USA, 2004. ACM. http://dx.doi.org/10.1145/989863.989936.

[31] J.-S. Sottet, G. Calvary, J. Coutaz, and J.-M. Favre. A model-driven engineering approach for the usability of plastic user interfaces. In *Engineering Interactive Systems*, pages 140–157. Springer, 2008. dx.doi.org/10.1007/978-3-540-92698-6_9.

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

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

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