Cluster Computing manuscript No.
(will be inserted by the editor)

On Separation of Platform-independent Particles in User Interfaces

Survey on separation of concerns in user interface design

Tomas Cerny · **Michael J. Donahoo**

Received: date / Accepted: date

Abstract The complexity of User Interface (UI) design grows quickly with the number of application concerns. Such complexity compounds with additional requirement of contextual-awareness (i.e., adapt to user location, skill level, etc.) and support of heterogeneous devices and platforms (e.g., web, mobile app). Implementation support of such a wide-range of orthogonal concerns often results in restatement of a significant portion of the UI description using platform-specific components. Replication requires repeated implementation decision, greatly increasing development costs since each version/context variant may need separate development. Naturally, such replication also produces error prone maintenance because code updates must correlate among all replicas. Using separation of concerns, the application can be decomposed into fine-grain fragments, which we call particles, some of which are platform independent and others are not. Using this decomposition, this paper addresses the above inefficiency by dynamically composing particles at runtime that match user demands, context, and target platform.

Keywords Separation of concerns · user interface · platform-independence · aspect-oriented programming · networking

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

Tomas Cerny ✉
Computer Science, FEE, Czech Technical University,
Charles Square 13, 12135 Prague 2, Czech Rep.,
E-mail: tomas.cerny@fel.cvut.cz

Michael J. Donahoo
Computer Science, Baylor University, Waco, TX, 76798, US,
E-mail: jeff_donahoo@baylor.edu

1 Introduction

Conventional UI designs tend to describe concerns [20] tangled together [25], through self-contained UI definitions [7]. The aim is to capture a particular UI situation as well as to centralize concerns impacting the particular view. For instance, consider the composite design pattern used to describe a particular UI page [16]. Although such design provides well understood composition and integration mechanisms simplifying development of given UI perspective, it does not provide enough flexibility to deal with UI variations that may be needed for context-aware applications. The tax for the "all at one place" description is that each UI variation may be treated as a different UI, restating information [7].

For example, consider a UI page that presents application data as a form, while considering multiple changing concerns [7] [20], such as order of displayed fields, following a particular layout, each field having a specific label and a widget. The user input validation and constraints apply in widget configuration. Furthermore, security concerns enforce access control, and the description binds to a given data instance. Such a description is easy to read from the global UI perspective, although there is no explicit separation suggesting which part of the description deals with presentation, which part is dedicated to layout, and so on.

When application context taken into account to improve individual user experience, the UI should reflect various context-aware situations. Consider the case when the user's access role changes; some fields should disappear, impacting the layout; certain validation criteria may drop or emerge. This situation may lead to additional conditionals applied to the UI description, increasing its complexity. Next, the user changes the screen size (extends window or rotates the screen). The application layout should extend, although layout is usually tangled with the rest of the page elements. This may lead to the introduction of a new page,

considering and restating the same data, constraints, validation rules, conditionals, etc. Furthermore, since the user may move from computer to smartphone, the application may need to support both web interfaces and native client for a mobile platform. The mobile platform may support native UI widget features to increase usability [25] as well as an offline mode. On the other hand, the client application must restate all page concerns, data structures, conditionals, etc. Furthermore, developers might apply repeated decisions in the UI design across multiple platforms using native components for the UI presentation.

A concern-separating UI design approach [7] does not aim to centralize concerns for a particular UI in a self-contained definition. Instead, it aims to divide the UI definition into independent stripes. In order to build the whole picture, traditional approach requires locating and interpreting the self-contained definition; here, all individually defined stripes must be integrated. This seems rather excessive for small and simple UIs, but the increased concern reuse make it beneficial for large UIs [7]. The support for concern variability [25] resolved at runtime, well-reflects users context providing individualized UIs.

The concern-separating approach allows distribution of stripes separately within different delivery channels to clients [8–10]. This positively impacts the UI responsiveness and extends client-side caching abilities [10]. Moreover, it is possible to divide these stripes onto platform-independent and platform-specific, which we research in this paper.

This paper addresses UI design difficulties when dealing with clients on heterogeneous platforms and serving users with different contexts and conditions. It suggests separating out the particles of the UI that are platform-independent from those that are platform-specific. The independent part is provided in a standard, machine-readable format to support its reuse across different platforms, including web, standalone and mobile platforms. Client prototypes of such an UI approach are implemented and evaluated for three different platforms. Next, the particle-separation approach is considered in the perspective of existing web-based strategies with incremental and full state transmission to clients. The outcome suggests that future systems could detect client abilities and available resources and choose an appropriate UI rendering strategy. For instance, basing on battery charge level, available CPU, Memory, etc.

This paper is organized as follows. Section 2 provides a survey on design approaches addressing separation of concerns and introduces basic notions. Section 3 brings related work. The separation of the platform independent particles from the UI is described in Section 4. Three platform prototypes are described in Section 5. Section 6 considers the perspective of incremental and full state scope transmission in web-based strategies. Finally, Section 7 presents our conclusions and future work.

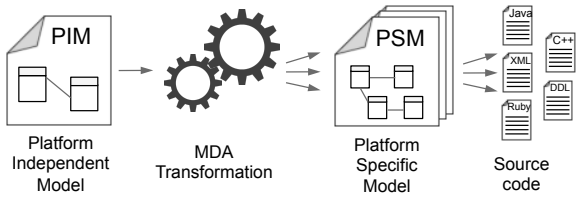


Fig. 1 Model-driven architecture transformation

2 Concern Separation Design Approaches and Notions

Design approaches aim to face problems such as separation of concerns, reduction of restated information, context-awareness, platform-independence, etc. The research on design approaches ranges from Model-Driven Development (MDD) [11], concerns separation through Aspect-Oriented Programming (AOP) [7] often used in conjunction with metaprogramming, Generative-Programming (GP) [12] that aims to combine different sorts of descriptions and proprietary formats or basing on Domain-Specific Languages (DSL) [25].

When considering platform independence the idea of MDD [11] comes to place. It suggests that the central source of information is the model. The rest of the system is generated from the model using transformation rules and templates specific to a particular platform. Thus it is possible to reuse the model and apply the same information towards different platform. MDD supports reuse across different components since information captured at the model-level transform to multiple locations and thus reduce the information redefinitions and restatements.

The platform-independence is mostly apparent from the Model-Driven Architecture [21] that considers multiple model abstractions as shown in Fig. 1. The problem domain captures the platform-independent models (PIM), such as UML. A model-to-model transformation enables to push the PIM information to platform-specific models (PSM) that reflect a particular domain in a given platform. Model-to-code transformation then derives the source code of the given platform in which the application operates. The resulting code is not meant for manual updates, instead the modification must take place at the model-level.

It is possible to use platform-independent models, such as UML to derive UIs for data presentation. Although, the result is not sufficient for practical use since various sorts of perspectives are missing in the model [11]. For instance, consider input validation, constrains, security etc. Fortunately, UML has an extension mechanism that enables to introduce UML profiles. [11] introduces profiles that correspond to industrial standards for persistence, constraints, input validation, presentation specifics or even security at the model-level.

The MDD usually performs the transformation at compile time [7, 11], as it might be time consuming [31]. This might be seen as a disadvantage when it comes to support of

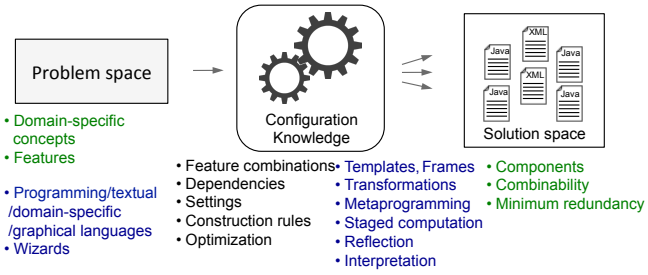


Fig. 2 Sketch of generative programming concept

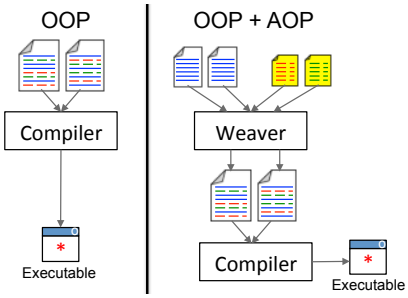


Fig. 3 Compilation of a program that involves aspect weaving

adaptive and context-aware applications. For instance, [4] suggests that future adaptive applications should consider runtime information and adjust the transformation at runtime. With the compile-time approach this leads to production of a large amount of states that extend the volume of produced code. The issue comes when certain dependencies that impact the result form combinations exponentially [7], then the transformation process becomes time consuming and the application demands large memory space.

Another disadvantage comes to place when there is a co-existing source of information that the model must correspond to. In such case the model introduces restatement. The weak correlation assurance mechanisms may lead to errors. Unfortunately, this is often the case when model represents the UI, when the rest of the system is code-based.

MDD has the ability to combine multiple models together, although a generic, multi-model integration mechanism is missing [30]. This limits the MDD ability to address separation of concerns (SoC) [20].

The term *concern* can be understood as any set of information that affects the source code of a particular component. Software design aims to apply SoC as the practice leads to modular applications that simplify development and maintenance [7]. Consequently, this leads to concern reuse, high cohesion [23] of the particular concern as well as to loose coupling among different concerns [23]. This leads to increased readability and developer concentration.

Naturally, some concerns tend to be easy to separate and some not. In particular, cross-cutting concerns are such concerns that tend to be hard to separate and are directly responsible for tangled code, a spaghetti code that mixes various concerns together. The natural design goal is to separate these. Unfortunately, the underlying code that uses traditional constructs and mechanisms has limited ability to effectively deal with these sorts of concerns [22].

The cross-cutting concerns are apparent at code-level, but also at the model-level [27]. Since the generic multi-model integration mechanism is missing, these concerns may lead to tangled information in models.

In general the domain of SoC is addressed by AOP [20] and GP [12]. These approaches suggest describing given concerns through independent constructs or blocks, often involving DSL descriptions. These concern-defining blocks are woven to core components to extend their behavior.

GP, with sketch in Fig. 2, aims to utilize all sorts of domain methods in the integration to code-based systems. In fact the idea is not that far from MDD, the source code is generated through templates and transformation rules. The difference is that the input is not necessarily a model. It can be any sort of information captured by code, DSL, model, etc. The goal is to address the gap between program code and domain concepts, support reuse and adaptation, simplify management of component variants, and increase efficiency. Unfortunately, what stays the same as in the MDD is the compile-time derivation and not well-described integration mechanism.

UI design involving GP [29] suggests to consider three parts: a DSL for UI description, configuration generator that automates the product assembly by taking the DSL specification and assembling the implementation components from it, and an extensible collection of elementary components available for the assembly. In a case study, a system combining two hundred UI features gives the variability of 5×10^{17} prototypes generated at compile-time.

The difference comes with AOP approach. It has very well defined formalization [20, 32]. The program is designed with components and the specifics, variants, extensions, optimizations, etc. are captured through a construct called aspect. Aspect is something that cannot be easily captured in a component. Thus this gives two independent perspectives: components that form the core program and its extensions that are defined separately.

The most enlighten idea of AOP comes with the mechanism the two constructs connect together. AOP defines a novel construct called Join Point (JP). JP is a place in a component program that serves as the place where component and aspect come together. This place can be determined statically at compile time or at runtime. For instance, a method determined by its name can be a JP, annotation can indicate a JP, or even a method call at particular instance determined by a runtime context can be a JP.

The responsibility for aspect and component connection is given to aspect weaver [20]. As stated in [20]: “Aspect weavers work by generating a JP representation of the component program, and then executing (or compiling) the aspect programs with respect to it”. This situation is well captured in Fig. 3 and 4 considering an object-oriented program

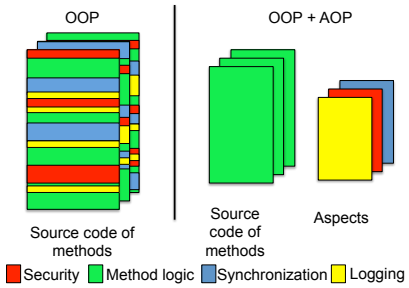


Fig. 4 Cross-cutting concern tangling in OOP vs. AOP separation

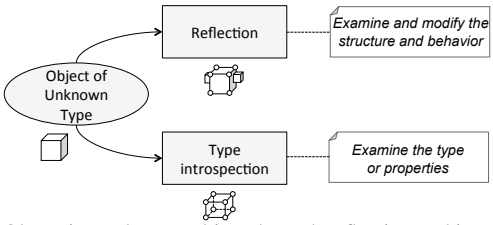


Fig. 5 Observing unknown object through reflection and introspection

and its variant extended with AOP. In order to receive the JP representation it usually utilizes metaprogramming applied to the component program. The metaprogramming captured in Fig. 5 has the ability to either modify the original structure and behavior - reflection - or examine the structural properties - introspection.

Once it has the JP representation, the next step is to integrate, "weave" aspects to it, which produces the result either applicable as component and embedded at runtime to the program or a component source code that is further compiled by regular compiler.

The connection of a particular aspect is as follows. Each aspect is further divided on pointcut and advice [32]. The responsibility of a pointcut is to determine whether the aspect applies to a particular JP. Thus pointcut can be seen as a query that references JPs or even runtime context and resolves to true or false. Such query can follow a grammar with logical, arithmetical, textual, etc. operations and terms given by JP and context. The advice provides the extension information. It does not need to use the same language as the core application. It can be a DSL, generally something the weaver interprets. The advice can be a template similar to these used with MDD.

To summarize the advantages of AOP, we point out the operability at runtime, adoption to co-existing component code with utilization of metaprogramming, with reduced information restatement since the component code acts as model. The MDD rules in the perspective of AOP are given by pointcuts that become generic, and can integrate runtime information. Thus results received from AOP at runtime can adapt to changing runtime context. Furthermore, the AOP is not limited only to code, the input can be a model representation.

In the UI, [7] observes that UI concerns such as presentation, layout, data binding, input validation, etc. tent to tangle together, which limits the UI variations and concern reuse. Suggested concern separation and runtime aspect weaving involving templating simplifies to design of context-aware applications [25]. The case study given by [7] shows the approach use in production drawing a minimal performance impact, while observing considerable reductions to development and maintenance efforts. The study indicates 30% UI code reductions due to concern reuse and reduced information restatement that is automated by the aspect weaver. The study also indicates that generalization of pointcuts can apply using the well-defined JP elements that base on existing development standards, such persistence, constrains, input validation, etc. [3, 13, 14].

The introduced approaches although had one thing in common. When considering web applications or client-server interaction, all approaches applied the result derivation at the server-side. Thus all the advantages coming from more or less efficient concern separation get lost upon UI rendering at the server-side. Thus the client cannot benefit from the separation.

Our previous work evaluates the possibility to maintain SoC for the UI delivery in the client-server architecture for web-applications [8–10]. Our preliminary results indicate that such an approach positively impact the UI responsiveness and brings concurrent processing to the UI delivery. Next, it extends concern reuse, which impacts the volume of transmitted information. Furthermore, the client-side can individually select, which concerns can be reused and which need to be requested from the server-side.

In this paper extend our previous work considering the AOP-based approach, although the above text indicates that MDD and GP has certain similar characteristics.

3 Related Work

Existing UI approaches that target platform independence usually base on abstract UI descriptions that are platform independent and further extended to concrete descriptions while adding the native-client specifics [25]. The UI SoC approaches usually separate out data and from the presentation. Contemporary UI development frameworks such as Google Web Toolkit (GWT) [18], AngularJS [17], etc. suggest such a separation. Data values are provided through a separate channel involving machine-readable formats for the delivery. In general these approaches usually restate co-existing information in the UI descriptions. For instance, it is common to restate information from data definitions in the UI, although this leads to extended development and maintenance efforts due to limited mechanisms to enforce correlation [19]. Furthermore, growing complexity with context-aware UI features usually extend the manual efforts [7].

Various model-based and DSL-based approaches, such as task models with Teresa tool [26], User Interface Pro-

tocol (UIP) [25], XML-based User Interface Markup Language (UIML) [2], MDA-based UI [28], Model-Based User Interface Development (MBUID) [6] and others suggest to use two or even more levels of abstraction for UI descriptions. These usually divide on an Abstract UI (AUI) and a Concrete UI (CUI) with platform-specific information.

The AUI is platform-independent and can be reused for multiple platforms. The CUI is received from a transformation. The existing approaches usually consist of strict transformation rules that are specific to a particular platform. For instance, Teresa [26] determines the result at compile time, although [4] suggests that future systems need to operate at runtime to consider runtime information, mostly when dealing with context-awareness. Next, [7, 24, 25] suggest that the transformation rules should base on a grammar that allows defining generic rules, which brings their reuse. The CUI determines the final UI, it can use an interpret for standalone systems; alternatively it produces an HTML that is sent to web browsers.

An alternative to this brings UIP [25] that streams the CUI to clients at various platforms (C#, iOS, Web, etc.) that interpret the CUI by platform-specific client using native UI components to increase usability. Regarding the UIP, we also point out a considerable disadvantage. A novel platform requires the server-side to change. Thus the provided output does not naturally scale, and from the future reuse perspective this should be addressed.

Although, the above approaches deal with context-awareness or platform independence there exist limitations and issues regarding to practical usage. First, they do not effectively address cross-cutting concerns [27]. Since they base on models or DSLs, a generic, multi-model integration mechanism is missing [30]. Next, the approaches focus on the perspective of UI and thus they do not consider that applications contain separately defined data definitions at different components or subsystems, which leads to coexisting information and information restatement. Even though, the model reduces information restatement in the UI, the model itself has definition independent of the co-existing data definition.

Next, the resulting UI is provided to the client in tangled format, which might be a limitation considering the client-server interaction and our observations in [8–10]. Furthermore, from the perspective of [11], the above UI approaches reinvent the wheel since they do not consider the standards applied together with data definitions regarding to persistence, input validation, etc. Such information should be reused rather than redefined in the UI description. Assuming the model or DSL provides limited type-safety [7], it is fairly easy to introduce an error upon restating information form the underlying data definitions or when changes apply in the evolution.

Furthermore, performance analysis is rarely given by the above UI approaches, which limits its practical use. For instance, UIP [25] employs automated element layout distribution based on metrics, which delays the UI rendering in the range of seconds, although such [7] latency can be hardly adopted to production systems.

MetaWidget [19] does not address context-awareness, although it utilizes another approach. It suggests applying code-inspection to data model, deriving various types of presentations. This reduces information restatement, on the other hand the approach has limited transformation rules, which makes the approach hard to use for UI variations or context-awareness.

From the perspective of development frameworks and technology, HTML5 and CSS3 suggest responsive web design. It allows the presentation to adjust to screen size and makes the UI presentation reflect the resolution. This can be adopted, for instance, for layouts. Notice that it considers only a subset of layouts. For example, it is non-trivial to make a custom order of fields displayed at the page, and it may require absolute positioning, which become impractical from the development and maintenance perspective.

GWT [18] is a web development framework that transforms the UI description defined in Java to JavaScript (JS) representations. The Java code can be seen as a model that through the transformation rules using introspection generates multiple JS outputs for different web-browsers. GWT applies SoC to the UI delivery although in a small scale. It separates application data values from the presentation. Data are requested through a separate stream, through web resources. This makes the data values separable and easy to machine process.

GWT usually generates the JS representation so that client requests all the applications states at once. This brings the support for partial offline interaction. GWT fits to a specific domain of systems such as interactive consoles, email clients and so on. Such approach is not a best fit to information systems that contain large amount of forms. Weakness of GWT can be seen in Java philosophy that is distant and does not correspond to web-development. The high abstraction brings difficulty for debugging low level interaction. Similar to MDD and GP, in GWT the Java code is compiled to produce JS code that is deployed to the server in the low-level format. The produced JS is not aimed for extensions and all changes need to be made at the Java level.

The alternative view to GWT brings AngularJS [17], which is similar to GWT in a way that it separates out data values through web-resources. On the other hand, it goes the direction of low-level abstraction using JS and HTML to design application UIs. Another difference is that AngularJS approach goes the direction of incremental state extension, which fits better to data oriented systems. Novel is a client-side template mechanism that greatly simplifies data

decoration. On the other hand when compared to JavaServer Faces templating [5] the template composition and decoration mechanism is limited. Similar to GWT, AngularJS is a UI framework not providing any correlation mechanism with coexisting data definitions, which exacerbate development and maintenance efforts and does not prevent typological errors.

The SoC-based UI approach introduced in [7] has the ability to reuse coexisting information similar to MetaWidget. Instead of providing multiple levels of abstraction it suggests to define concerns separately. For instance, JP representation of data definitions derives the core structure, and aspects define field presentation, layout, data-field to UI field mapping, input validation, security, etc. Although this approach reuses coexisting information from data definitions, addresses SoC, context-awareness, it does not bring platform-independence, nor does it apply SoC to the delivery.

The above approach is extended by our previous work applying SoC to the delivery [8–10]. Multiple concerns can be provided through separate channels, which brings more opportunities for client-side UI variations, concern reuse or caching as well as improvements to UI responsiveness. In this paper we research the platform-independence perspectives.

4 Separating out Platform-Independent UI Particles

In order to reduce restatements of recurring information across platform-specific UIs that present data, it is necessary to classify various types of information that influence the UI. Platform-independent, model-level description of UIs is researched in [11], and it suggests that the application data model is the main driver for data presentations, although basic data structural information is not sufficient to derive usable UI components.

The [11] suggest extending data descriptions with various information profiles. For instance, the data structural information is accompanied with constraints, input validation, and field semantics for the presentation or security. [7] then shows that such extensions that come from persistence [13] and input validation [3] standards are practical for use in code-based applications and can be considered by code-inspection. [7,9] suggests to treat the information accompanying the data model not only as data structure, but as AOP JP representation [20] that determines a particular field presentation. In addition, the application runtime context is considered together with the structural information as its extension. Thus together they determine the JP representation that is the source for context-aware UI derivation.

The advantage of [7] code-inspection applied to data definitions is that it automates the derivation of such information, avoiding manual work. Aspect weaver conducts the inspection. The JP representation is the subject of AOP-based transformation that uses the JP constellation when determining field presentation templates to use as well as their content.

The template content uses a DSL together with the target language extended with constructs interpreted by the aspect weaver to direct the transformation. These constructs reference data definitions on the meta-level not introducing coupling to a particular data type or its field. These constructs enable integration of other concern, using the same mechanism that is used for the field template selection, mentioned in Section 2 with AOP, etc. The interpreted content is further decorated through a selected layout template, which allows deriving variation matching the user device screen size.

The above approach can be considered from the perspective of platform-independence. As suggested in [8], the JP representation can be treated as platform-independent as it correlates to the MDD approach introduced in [11]. The [11] shows that similar elements can extend the platform-independent UML models through UML profiles and thus be considered in the UI derivation in MDA, which has known ability in support for platform-independence.

The JP representation is used for platform-independent UI derivation. Besides the above-mentioned information, it needs to represent the structure of particular data definition [7]. The application context also extends the JP representation with information that is not specific to a particular platform. Context can either extend or modify the JP representation. For instance, it may restrict certain properties for a particular use basing on the access rights. Similar to GWT or AngularJS, the logical separation of data values is a step towards reuse across different platforms.

Table 1 shows example elements of the JP representation. The example shows JP elements derived from data definitions considering Java standards for persistence [13], input validation [3] as well as presentations profile introduced in [11] or received from contextual information derivation. Each JP can have further properties. For instance, a numeric value for length. The first column of Table 1 give the JP element name, next column give a short description and meaning, the following column provides applicability to particular field type if applicable, and the last column gives an example source of the JP considering the Java platform. The JP elements given in Table 1 are not limited to Java and can be derived from UML [11] or other platforms [25].

The above elements, although determining the UI presentation of data do not say which UI components, widgets, validation listeners, etc. should be used. That this sort of information is platform-specific and involve presentation and layout templates that use the target UI language components and constructs. The integration of templates with the JP representation and data values must be considered by a pro-

Table 1 Example JP representation elements and corresponding source from Java

JP element	Description	Data type	Java source
Class-level			
entity	Class name	-	Class#name
fullClassName	Full class name	-	Class#package.name
Field-level			
field	Field name	-	Field#name
dataType	Field data type	-	Field#type
1. Persistence profile [13]			
notNull, required, maxLength, unique	DB table column props.	Any	Field@Column()
notNull, required, unique	DB table column props.	Any	Field@JoinColumn()
temporal	Date, Time, TimeStamp ..	Date	Field@Temporal()
2. Validation profile [3]			
minLength, maxLength	Value length in the range	String	Field@Length()
min, max	Value in the range	Number	Field@Min()/Max()
email	Match email pattern	String	Field@Email()
pattern	Matches the reg-exp	String	Field@Pattern()
future, past	Future/Past date	Date	Field@Past()/Future()
notNull, required	Not null value	Any	Field@NotNull()
required, notEmpty	Not empty value ..	Any	Field@NotEmpty()
3. Presentation profile [11]			
link	Web link expected	String	Field@UiLink()
text, cols, rows	Long text expected	String	Field@UiText()
param (name, value)	Any Param expected	Any	Field@UiParam()
html	Html expected	String	Field@UiHtml()
password	Secret text expected	String	Field@UiPassword()
type	Type of widget to use	Any	Field@UiType
order	Order in view	Any	Field@UiOrder()
ignore	Ignore field in UI	Any	Field@UiIgnore()
profile	To support logical fragments ..	Any	Field@Profile
4. Access control profile [11]			
restrict	Third parti restriction	Any	Field@Restrict()
roles	Values specifies user role ..	Any	Field@UiUserRoles()
5. Sample Context (passed by developer in a particular domain)			
screenSize	User screen	-	Calculated
origin	User origin locaiton	-	Calculated
errorRate	User error rate	-	Calculated
age	User age	-	Calculated
device	User device	-	Calculated
openingHours	Store opening hours ..	-	Calculated

cessor with the knowledge of a particular platform, which makes the processor platform-specific.

Considering the client-server interaction and support of platform-heterogeneous clients that aim to assemble the UI using native components, the aspect weaver introduced in Section 2 with AOP, must divide onto two parts. First, an aspect weaver relevant to the server-side and producing platform-independent output, such as JP representation and corresponding data values for different data definitions. Second, to an aspect weaver relevant to the client-side that interacts with the server provided information and uses the knowledge of the particular platform to interpret the JP representation using native components and constructs. The interaction between the weavers should involve a platform-independent format. Web-resources provide standard formats that are understood across platforms, and thus they naturally fit the goal.

Although it may seem that we solved the separation and can proceed to the next section, we should consider a practical impact. Providing the entire JP representation to clients can be impractical, mostly when considering that it can consist of internal information or information not relevant to the UI presentation. Thus application internal information should be filtered out. At the same time we can consider that all of the client-side weavers will make the "JP representation"-to-"UI presentation" transformation. It essentially becomes repetition or repeated decision across different platforms. Furthermore, it might not be known ahead of time, which join points are used at the client-side to determine the presentation. To address this, the AOP transformation can be partially performed at the server-side, considering the unfiltered JP representation and giving an advice on field template selection to the client-side weaver.

The question is how to connect the transformation result with templates that are at the client-side in a platform-specific format. It is possible to perform the transformation in a way that a standard set of templates is considered by the server-side, referencing each template by unique key that accompanies the provided JP representation of each data field. This makes the platform-specific client implementation easier as it is known ahead of time, which template should exist. There is no limitation that would prevent the client to make local decision or use multiple sets of components.

The filtered fraction of the JP representation is provided through a web-resource in a machine readable format, separating out information not relevant to given user's access rights, context, or information not relevant to UI, such as data identifier, version lock, business key, etc. [13]. Security is considered at the server-side for both the provided JP representation and values for the requested data instance in a given context (user, location, time-frame, etc.). Only data values that would be part of a conventional secured-view are provided to users, but in a machine-readable format.

The client-side is responsible to provide the expected set of presentation templates using platform-specific components. It determines layout and implements the platform-specific aspect weaver. The weaver interacts with the server-side to obtain data values for presented data and JP representation. It populates the local presentation templates with information given by JP and their attributes. Additionally, local context can select presentation templates from multiple template sets (read-only/editable fields, data table, report, list, etc.). Furthermore, it is possible to aggregate provided JP representation and data values to provide merged or specialized views, decorate the data presentation with wizard-like components or collapse the presentation into multiple panels.

5 Platform-Specific UI Clients

A conventional server-side application, usually available on web, can integrate the approach to open its UI for reuse on different platforms. Thus besides the web access it can provide mobile and standalone system access to improve the usability, efficiency or provide partial offline operation.

The server-side of the application should contain aspect weaver capable of JP representation derivation, template suggestion and JP/data transformation to a machine-readable format. As an example, it is possible to use AspectFaces weaver implementation capable of Java platform code-inspection considering existing Java standards [3, 13], as well as the extensions shown in Table 1. Custom application context can be passed to the weaver and influence weaver operation. The transformation rules use Java expression language for pointcuts. The weaver has the ability to use the Annotation Driver Participant Pattern (ADPP) to filter out given properties. The weaver-derived JP representation for requested data is generic and provided as a single web service responding in a JSON format. The weaver can also determine data values to the corresponding JP representation. For illustration, see the server component in Fig. 6.

Next, let us consider the mainstream web-based UI and platform-specific clients. We should aim to reuse the same UI mechanism provided by the server, although there is no limitation that would prevent us from using conventional web design approach for data presentations. Unfortunately, the conventional design would introduce restatements and inefficiency when dealing with context-aware UIs.

The web-based implementation that provides UIs basing on HTML is interpretable by web browsers. Integration of our approach would use conventional technology for page flow and page high-level definitions and component interaction; the difference comes for data presentations. Consider the bottom right node given by Fig. 6 that represents web-based client. The requested page is supplemented with JS-based client weaver and templates defining UI component for presentation and layout. The client interprets the page that does not contain any data presentation and instead indicates the intention to present given data within a local context, and passes this intention to the client weaver. The client weaver requests particular JP representation from the server concurrently with the request to obtain data values for particular data instance. In case multiple data show at the page, multiple requests can be issued or an aggregated request can be issued. Weaver then interprets the received information integrating the JP representation with templates extending it with data values. Finally, the received result embeds to the page. The data submission follows the conventional POST/GET mechanisms or uses JS submission.

Implementation of a platform-specific client does not need to apply any platform-specific extension to the server-side. It requires implementing a client weaver that requests and interprets the JP representation and data values received from the server-side. The platform-specific sets of presentation templates are defined, reflecting the expected set of template keys. These templates consider native components that provide expected functionality, capture constraints, input validation, etc., given by the JP representation model. The advantages of platform-specific features include increasing UI usability (e.g., touch-based element selection). The appropriate presentation template is selected basing of factors such as server-side suggestion or the local context. Local context may influence the set selection, presentation goal, etc. Templates for layouts are defined complying with the expected and supported screen-sizes. The client weaver further requests and embeds data values and has the ability to post them to the server-side. Fig. 6 captures the situation with Android and Desktop clients.

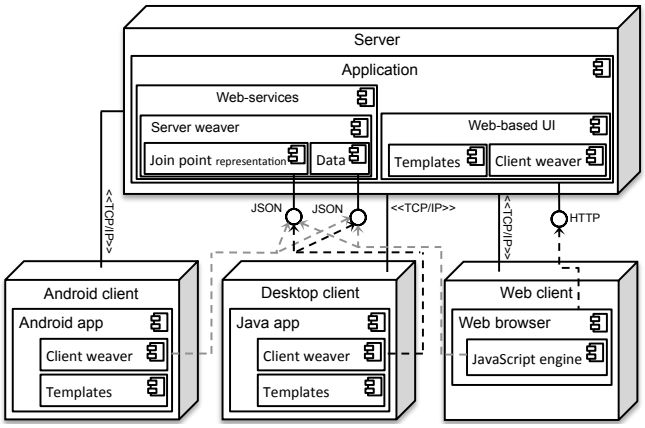


Fig. 6 Sample UML deployment diagram - three heterogeneous clients

Heterogeneous clients interpret the server-provided, platform-independent information. No matter a processed data element, the same presentation templates apply over and over again for the particular platform. The templates are reusable across applications. It is possible to consider generic layouts and reuse them across various data types. The size of the application data-model does not influence the size or complexity of clients and provided templates. The contra example is when data model extends with a novel, unusual field type not previously considered by templates, all clients need to introduce a novel presentation template.

The client weaver and templates are the same for an application providing a single data element or for an application with hundred of various data element types. This implies that changes to server-side data structures are automatically reflected by the weaver at the client-side, since the provided JP representation reflects it and existing templates are reused. Thus server-side evolution and changes directly apply by all clients, with no manual efforts, unless, the previously mentioned contra example occurs, which is expected to be a rare scenario.

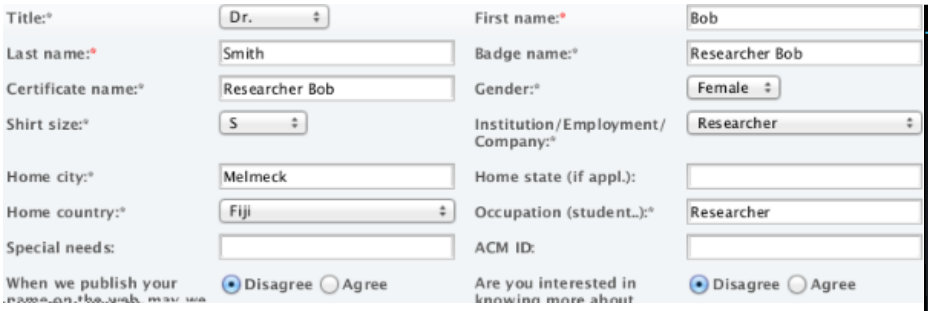


Fig. 7 Web-based UI demonstration

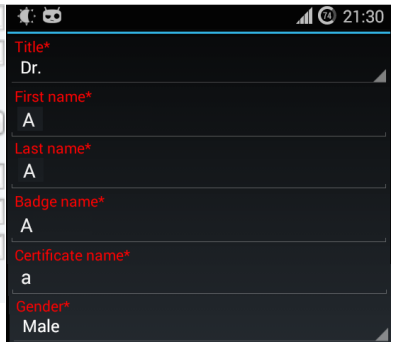


Fig. 8 Android-based UI demonstration

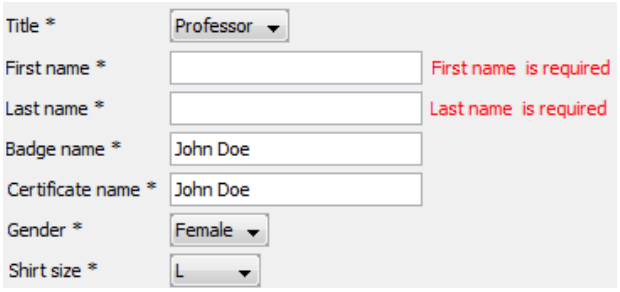


Fig. 9 Java Swing-based UI demonstration

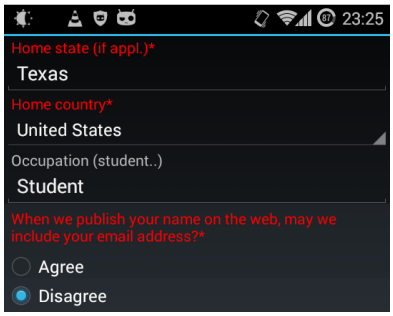


Fig. 10 Android-based UI demonstration

Regarding the development efforts, designing a platform-specific client becomes simplified since information is reused and restatements are reduced. For a demonstration, we implemented three clients of different platforms. Our implementation corresponds to the deployment diagram in Fig. 6. The web-based client UI is shown in Fig. 7, an Android mobile client UI is in Fig. 8 and 10 and a standalone, Java Swing client UI is in Fig. 9. They all use the same application server, base on the same data model, domain business rules and services to provide context-aware data presentation and data manipulation. The web-based client is different from other clients as it loads the weaver and templates from the server-side in the form of a JS library. Various sorts of data presentations can be derived ranging from forms, read-only forms, tables, lists, etc. The nature of the separation enables caching of the JP representation for a particular time span, although this determines the volume of context-awareness. The application page-flow is left for custom, non-automated implementation.

The benefits are that upon server-side change all prototypes update the presentation. Human-errors related to restating information mitigate, since the metaprogramming used by server weaver automates the information propagation. Novel concern can apply at the server weaver level, but it is not limited to it. For instance, similar to JP representation and data values the client weavers can consider another resource or channel. The separation of platform-specific templates supports reuse and caching. This is mostly evident at the web-based prototype that gives the opportunity to cache templates. When limited context-awareness in given session exists then it is possible to cache the JP representation. [9, 10] provide a study evaluating the web-based approach regarding the UI responsiveness, volume of transmitted information, server involvement and concurrency. The concern separation approach considered in the study shows noticeably better performance over the conventional approach that tangles all concerns together in UI descriptions.

The implemented prototypes request the JP representation and data values instantly upon navigation, this although does not comply offline operation. The mobile and desktop applications could consider pre-loading the JP representation and data for given user, session or scenario in given context and cache while offline, in local database. The online transition should consider current data versions and possibly evaluate changes in context before any server-side data modification applies. Offline operability and caching can be extended for web applications using local JS database storage preserving data values over page loads. Example store is HTML5 local storage or Minimongo of Meteor[1].

6 Web-based State Transmission Strategies

Section 3 mentions two web-based alternative approaches. One brought by GWT with full state transmission to the client and AngularJS with incremental state transmission. This is related to the ability to package the UI content and influence the client-server interaction. One approach pre-loads

[1] Meteor, JavaScript App Platform, 2015, http://www.meteor.com

multiple states that client can get into, and the other loads a small initial page, whose state increments on demand.

GWT builds on a higher abstraction designing the UI in Java that is type-safe. At the same time even Java does not have a mechanism to avoid restatements from data definition constraints [7] or to effectively handle duplication cause by inability to deal with cross-cutting concerns. The benefit of having the UI defined in Java brings the ability to use the introspection to derive the full state of the application that gets transformed to JS. GWT provides multiple transformation results to address web browser incompatibility. When we consider that the Java code defines model, then the underlying philosophy is not far from MDD. At the same time the high-level of abstraction make optimization and debugging hard, since the result of the transformation does not correspond to what is defined.

AngularJS goes the direction of low-level design using JS and HTML with extension in the form of templates and client-side "compiler" that interprets the templates towards server-provided data values. The templating brings good improvement for data iterations such as tables, on the other hand it is no help when dealing with data presentations, such as forms that integrate multiple concerns [7]. The templating does not easily support recursion and decoration. The advantage brought by AngularJS is a bi-directional data mapping, which maintains correspondence of the JS client-side model and server-side model and simplifies development. Such model is delivered to the client through JSON, which is the same as in GWT or in our approach.

Neither AngularJS nor GWT separates out other particles than data, even though as we show in [9,10] it provides benefits from the caching and responsiveness perspectives. At the same time we must be aware of the different approach suitability. GWT with the Java abstraction and the ability to generate all states in the JS description suits well to one-page applications, such as interactive consoles, while AngularJS with low-level design lacks such ability. On the other hand such incremental state on demand suit better to information systems or data-oriented application where full state transmission would not suit.

In related work we mention that future systems should address context-awareness at runtime [4]. Even though, GWT generates it at compile time. In case the context-awareness produces many states and variations, the GWT result grows in volume, which may be impractical for use at mobile devices with limited resources. On the other hand GWT brings broader offline operability over loaded data.

Naturally, particle separation suggested in this paper fits better to the incremental state approach. A particular page requests only resources relevant to the particular state and context. The granularity of context-awareness influences whether the presentation changes for particular data, or whether it stay the same in given session, based on that the separately provided JP representation can be cached and reused, which extends caching abilities.

Considering AngularJS, the proposed particle separation brings the ability to extend caching, support further reuse of presentation templates, support concurrency and mostly decrease efforts related to UI data presentation, which is in AngularJS done manually. Our approach automates the data UI presentation design avoiding repeated work and duplicated definitions across the application. At the same time, even though AngularJS separates out data values, these must be provided manually from the server-side. Our approach provides automated derivation of data values corresponding to the context-aware JP representation. The JP representation provided to clients does not necessarily need to correspond to the persistence data definitions structures [7], it can consider multiple data aggregation or even its subset corresponding a particular filter. Furthermore, the approach can involve data transfer objects [15] with corresponding fields.

GWT would benefit from our approach mostly regarding to reduced development and maintenance efforts due to code-inspection, on the other hand the particle separation does not fit the full state transmission as each presentation would request a novel JP representation reducing the offline operability. On the other hand this extension brings GWT the ability to deal with growing granularity of context-awareness processes at runtime [4].

Considering the prototypes from Section 5 we implement GWT and AngularJS web-clients. Next, we extend them with our approach and consider the impact and benefits. The total number of requests extends due to the JP representation request. Both approaches differ in the client-cached and uncached scenarios.

While the AngularJS initial uncached page transmission volume extends by 9% due to the extra request for the client-weaver and templates, the GWT page transmission volume stays unchanged as the client-weaver size amortizes over the reduction in the form description.

The cached scenario amortizes the AngularJS client weaver and templates as the platform-specific particles cache over the time. The overall transmitted volume in our prototype reduces by 12.5% when considering highly context-aware situation (updated JP representation issued). When we consider no context changes and reuse of JP representation, which we assume is often the case, the transmission volume reduces by 65%. The transmitted volume uses compression and thus we must consider that the server-side processes even more information volume. In the above cases our approach reduces the server-side information volume processing by 46.5% in the highly context-aware situation or by 85% with the JP representation-reusing version.

The GWT transmission and server involvement stays the same for the less context sensitive situation in caching scenario. The highly context-aware situation that re-issues new

JP representation grows by 63% for the transmitted volume and the server information processing volume grows by 107%. At the same time we must point out that the second considered scenario gives the GWT prototype much higher flexibility compared to the original version. Based on the JP representation, it renders the data presentation in changed field orders, has different requirements on input validations, rendering or even presentation. The original GWT prototype only considered single data presentation in fixed format.

To summarize this section, we can classify various web-based approaches to render UI:

Server-side rendering - the server determines the UI description and client only interprets it with limited client-side involvement. The server determines the state presented to the client.

Full state client-side rendering - corresponds to GWT abilities. The server-side provides clients on the initial load the entire state transitions. Client then interprets the descriptions and incrementally requests data values to display them in the UI. Context-aware UI impact the volume of the description, it may demand high resources requirements at clients.

Incremental client-side rendering - corresponds to AngularJS or our approach. The initial page load provides a particular state, while the presentation involve client-side rendering. State changes are requested from the server, avoiding the communication overhead of transmission that would provide unchanged concerns. For instance, it is possible to reuse templates, weaver, etc.

Different clients may benefit from different approaches. For instance, we must have a different expectation from a client that uses a laptop attached to electric socket, than from a client using a cellphone with discharged battery, or even a client using smart watch. Transmitting the entire page states may be beneficial knowing the client most likely visits all states. Consider the related work example regarding GP UI [29] that had 5×10^{17} UI prototypes, most likely we do not want all these to be transmitted to the client at once.

In future systems we can consider selection of an appropriate rendering strategy for a given situation as the above ones. At the same time developers should avoid design of distinct pages for the same purpose and different strategy as it leads to extended development and maintenance efforts. Basing on AOP capabilities applicable to UI [7–10, 25] we believe that it has the prerequisites to effectively address the strategy selection, while avoiding duplication, repeated decisions and not placing considerable development and maintenance efforts when compared to conventional context-less page design in traditional approaches.

7 Conclusion

This paper considers separation of UI particles. The motivation behind the separation are growing demands on UI abilities in support of multiple platform-specific UIs presenting the same server-side information. Traditional UI design approaches lead to restatements and repeated decisions exacerbating the development and maintenance efforts with the UI context-awareness.

This paper provides a survey on exiting development approaches that has the prerequisites and capabilities to effectively address separation of concerns or even cross-cutting concerns. Even through, there exist multiple different approaches; various similarities can be find among them.

Separation of platform-independent UI particles from the specific ones leads to extended capabilities of their reuse across different platforms. Such particles provided in machine-readable format loosen the coupling among UI elements and the particular application platform.

The reduction of development and maintenance efforts is further supported through code-inspection of existing data definitions and AOP-based transformation to deal with cross-cutting concerns occurring in UIs.

The inspection driven by a server-weaver considering the runtime context derives a JP representation. This representation is the subject an AOP-based transformation deriving suggestion on field presentations template for a particular client-request. The JP representation together with the suggested presentation is provided to clients as platform-independent particle in a machine readable format. Similarly data values for particular data are provided through a separate channel.

Platform-specific elements, components and widgets are part of the particular client not influencing the server output. The integration of platform-specific elements with the JP representation and data is the responsibility of client-weaver that uses defined set of presentations templates and layouts.

When it comes to context-awareness and UI variations reflecting the application context, the server-provided JP representation fully determines the resulting UI provided to client. Changes in server-side are reflected by updated JP representation derived by code-inspection and thus immediate reflection by all heterogeneous clients. This avoids manual work and decreases the error-rate caused by human factor.

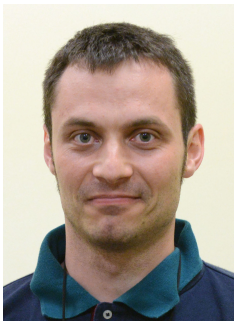
We evaluate the approach through three platform prototypes. Web-based, mobile and desktop clients are implemented. While the mobile and desktop version solely use the JP representation, data and server-side data manipulation services, the web-based version defines the client-weaver and templates at the server-side even though there is no limitation that would disallow to delegate the templates and weaver to any third party service.

Furthermore, we evaluate web-based approaches and classify the approaches to *server-side rendering*, *full state client-side rendering* and *incremental client-side rendering*. We believe that each such strategy can provide benefits to given situation mostly when dealing with varying client capabilities, resources or time-limited resources. Deeper evaluation and possible integration to server-side is left to future work. Although we believe that AOP-based approach has the prerequisites to deal with integration of such strategies not deteriorating the development and maintenance efforts.

The limitation of the approach is its applicability to data presentations and thus page-flow is left for custom, non-automated implementation. The page-flow provided in a platform-independent format is left for future work. Presentation components and layouts are considered as specific particles of heterogeneous clients, although layout platform-independent generalization is possible, but also left for future work. The future research will evaluate the approach applicability to service-oriented architecture, system integration, data exchange sharing validation or middleware interaction.

References

1. Meteor, javascript app platform (2015). Https://www.meteor.com
2. Ali, M., Prez-Quiones, M., Abrams, M., Shell, E.: Building multi-platform user interfaces with uiml. In: C. Kolski, J. Vanderdonckt (eds.) Computer-Aided Design of User Interfaces III, pp. 255–266. Springer Netherlands (2002)
3. Bernard, E.: JSR 303: Bean validation (2009). URL http://jcp.org/en/jsr/detail?id=303
4. Blumendorf, M., Lehmann, G., Albayrak, S.: Bridging models and systems at runtime to build adaptive user interfaces. In: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems, EICS '10, pp. 9–18. ACM, New York, NY, USA (2010)
5. Burns, E., Griffin, N.: JavaServer Faces 2.0, The Complete Reference, 1 edn. McGraw-Hill, Inc., New York, NY, USA (2010)
6. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. Interacting with Computers **15**(3), 289–308 (2003)
7. Cerny, T., Cemus, K., Donahoo, M.J., Song, E.: Aspect-driven, Data-reflective and Context-aware User Interfaces Design. SIGAPP Appl. Comput. Rev. **13**(4), 53–65 (2013)
8. 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)
9. Cerny, T., Macik, M., Donahoo, J., Janousek, J.: Efficient description and cache performance in aspect-oriented user interface design. In: Federated Conference on Coumputer Science and Information Systems (2014)
10. Cerny, T., Matl, L., Cemus, K., Donahoo, M.J.: Evaluation of separated concerns in web-based delivery of user interfaces. In: Information Science and Applications, pp. 933–940. Springer Berlin Heidelberg (2015)
11. Cerny, T., Song, E.: Model-driven rich form generation. Information-An International Interdisciplinary Journal **15**(7, SI), 2695–2714 (2012)
12. Czarnecki, K., Eisenecker, U.W.: Components and generative programming (invited paper). SIGSOFT Softw. Eng. Notes **24**(6), 2–19 (1999)
13. DeMichiel, L.: JSR 317: JavaTM persistence API, version 2.0 (2009). URL http://jcp.org/en/jsr/detail?id=317
14. DeMichiel, L., Keith, M.: JSR 220: Enterprise javabeans version 3.0. java persistence API (2006). URL http://jcp.org/en/jsr/detail?id=220
15. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
16. 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)
17. Green, B., Seshadri, S.: AngularJS, 1st edn. O'Reilly Media, Inc. (2013)
18. Hanson, R., Tacy, A.: GWT in Action: Easy Ajax with the Google Web Toolkit. Manning Publications Co., Greenwich, CT, USA (2007)
19. Kennard, R., Edmonds, E., Leaney, J.: Separation anxiety: stresses of developing a modern day separable user interface. Proceedings of the 2nd conference on Human System Interactions pp. 225–232 (2009)
20. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.M., Lopes, C.V., Maeda, C., Mendhekar, A.: Aspect-oriented programming. In: IECOOP'97-Object-Oriented Programming, 11th European Conference, vol. 1241, pp. 220–242 (1997)
21. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
22. Laddad, R.: AspectJ in Action: Enterprise AOP with Spring Applications, 2nd edn. Manning Publications Co., Greenwich, CT, USA (2009)
23. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (2001)
24. López-Jaquero, V., Montero, F., Real, F.: Designing user interface adaptation rules with t: Xml. In: Proceedings of the 14th international conference on Intelligent user interfaces, IUI '09, pp. 383–388. ACM, New York, NY, USA (2009)
25. Macik, M., Cerny, T., Slavik, P.: Context-sensitive, cross-platform user interface generation. Journal on Multimodal User Interfaces **8**(2), 217–229 (2014)
26. Mori, G., Paterno, F., Santoro, C.: Design and development of multidevice user interfaces through multiple logical descriptions. Software Engineering, IEEE Transactions on **30**(8), 507–520 (2004)
27. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. Computer **42**(10), 44–51 (2009)
28. Schattkowsky, T., Lohmann, M.: Uml model mappings for platform independent user interface design. In: J.M. Bruel (ed.) Satellite Events at the MoDELS 2005 Conference, *Lecture Notes in Computer Science*, vol. 3844, pp. 201–209. Springer Berlin Heidelberg (2006)
29. Schlee, M., Vanderdonckt, J.: Generative programming of graphical user interfaces. In: Proceedings of the working conference on Advanced visual interfaces, AVI '04, pp. 403–406. ACM, New York, NY, USA (2004)
30. 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)
31. 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 2006 conference) (2006)
32. Störzer, M., Hanenberg, S.: A classification of pointcut language constructs. In: Software-engineering Properties of Languages and Aspect Technologies (2005). Held in conjunction with AOSD'05



Tomas Cerny is a doctoral candidate in the Faculty of Electrical Engineering of Czech Technical University (FEE, CTU) in Prague, Czech Republic. He received his Bachelor's and Master's 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.

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.