Separating out Platform-independent Particles of User Interfaces

Tomas Cerny[1] and Michael J. Donahoo[2]

[1] Computer Science, FEE, Czech Technical University,
Charles Square 13, 12135 Prague 2, Czech Rep.,
tomas.cerny@fel.cvut.cz
[2] Computer Science, Baylor University, Waco, TX, 76798, US,
jeff_donahoo@baylor.edu

Abstract. User Interfaces (UIs) visualize a wide range of various underlying computer application concerns. Such orthogonal concerns present in even the simplest UIs. The expectation of support for users from various backgrounds, location, different technical skills, etc. serves to increase concern complexity. Nowadays users typically remotely access to applications from a variety of platforms including web, mobile or even standalone clients. Providing platform-specific support for multiple UIs further increases the concern complexity. Such a wide-range of concerns often results in a significant portion of the UI description being restated using platform-specific components, which brings extended development, and maintenance efforts. This paper aims to separate out the platform-independent particles of UI that could be reused across various platforms. Such separation supports reduction of information restatement, development and maintenance effort. The platform-independent particles are provided in a machine-readable format to support their reuse in platform-specific UIs.

Keywords: Separation of concerns, user interface, platform-independence

1 Introduction

Conventional UI designs describe all the displayed information and concerns [8] tangled together [9]. Usually the description can be found at a single location or logically divided to fragments. Although such monolithic design may be easier for the developer in the initial design, it does not provide enough flexibility to support UI variations for a particular context or situation. The tax for the "all at single-location" description is that each UI variation is treated as a different UI, restating information.

For example, consider a UI page that presents application data, while considering multiple concerns [1] [8]. It presents data fields in a given order, following a particular layout, where each field has 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 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 the system context changes, the UI must reflect the change. Consider the case when the user's access role changes; some fields should disappear and 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, and this may lead to the introduction of a new page, considering and restating the same data, constraints, validation rules, conditionals, etc. Furthermore, if the user decides to relocate from computer to cellphone to finish the work, the application may be accessed through web interfaces, or in order to support native UI widget features and offline mode, the application provides a native client for given mobile platform. Such a client application must restate all page concerns, data, conditionals, etc.

A concern separating UI design approach [1, 2] does not provide the ability to find all the UI-related information within a single description or location. Instead, it is divided into independent stripes. In order to build the whole picture, all stripes must be integrated. This seems worse for a simple UI, but the big advantage is the supported reuse, support for variability [9] and consequently the possibility to distribute the stripes separately within different delivery channels to clients. Furthermore, it is possible to divide these stripes onto platform-independent and platform-specific.

This paper addresses such difficulties when supporting various clients and particular contexts. 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.

This paper is organized as follows. Section 2 provides related work. The separation of the platform-independent particles from the UI is described in Section 3. Three platform prototypes are described in Section 4. Finally, Section 5 presents our conclusions.

2 Related Work

Existing research ranges from Model-Driven Development (MDD) [3], concerns separation through Aspect-Oriented Programming (AOP) [1], Generative-Programming (GP) [4], proprietary formats or basing on Domain-Specific Languages (DSL) [9]. Existing prototypes such as the User Interlace Protocol (UIP) [9] suggest basing on abstract UI descriptions and adding the native-client specifics to the description. Finally, a few contemporary UI development frame-

works such as Google Web Toolkit (GWT) [6] or AngularJS [5] suggest separating out data values to machine-readable formats.

The idea of MDD [3] suggests that the model is the central source of information, and the rest of the system is generated from the model. This usually works for application prototypes [1], although most production systems build on code-based approaches. MDD brings multiple benefits, such as supporting platform independence and transforming information captured at the model-level to multiple locations to reduce the information re-definitions and restatements. [3] aims to derive UIs from UML models and suggests industrial standards for persistence/constraints and input validation, as well as presentation specifics at the model-level. Such an approach allows deriving rich UIs for data presentations considering certain number of variations and conditionals; however, the approach is limited by the nature of MDD. It does not address cross-cutting concerns and does not apply to code-based design.

Separation of cross-cutting concerns is the domain of AOP [8] and GP [4]. These approaches suggest describing given concerns through independent components, often involving DSL descriptions. These concerns are woven to core components to extend their behavior. AOP has well-understood concern integration mechanisms and operates at runtime. An approach considering AOP for UI concern separation is provided by [1, 2]. Compared to MDD, it operates at the code-level and at runtime considering application context. Concerns such as presentation, layout, data binding, input validation, etc. are considered. The approach simplifies the design of context-aware applications [9].

A unified description of the UI is considered by UIP [9] in a DSL format called Abstract UI (AUI). This description is platform-independent and processed by a server-side application called the UIP server. The UIP server uses a core component called the Concrete UI (CUI) generator, which takes the AUI and context including the consideration of a target platform specified by the given client (C#, iOS, Web, etc.). Based on the selected target platform, the CUI generator interprets the AUI and context, and produces the platform-specific CUI that it streams to a particular client. These proprietary clients interpret the received CUI at the platform-specific environment using native components. The goal of UIP is to address context-aware UIs. From the perspective of [3], UIP reinvents the standards and from the perspective of [1], it uses custom DSL, which demands restating information from the application data model, increasing possible errors. A novel platform requires changes to the server-side, which does not naturally scale. Next, each data element must have its custom AUI, which must correlate with the actual application data structure. To avoid the necessity of manual definition of AUI, the approach of [1] can generate the AUI through code-inspection of the application data model.

MetaWidget [7], similar to the AOP approach [1], suggests applying code-inspection to data model, deriving various types of presentation. Unfortunately, this does not address cross-cutting concerns, changing context, and requires a one-to-one mapping preventing the use of templates.

From the perspective of development frameworks and technology, HTML5 and CSS3 suggest responsive web design (RWD). RWD 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 [6] is a web development framework that transforms the UI description to JavaScript representations and separates application data values from the presentation to a separate stream, requested through web resources. This makes the data values separable and easy to machine process.

AngularJS [5], which is similar to GWT, separates out data values through web-resources; next it defines a templating mechanism that allows decorating data. On the other hand the mechanism does not have natural support for recursive templating.

3 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. Platform-independent, model-level description of UIs is researched in [3], and it suggests that the application data model is the main driver for data presentations, although basic data structural information is not sufficient. The suggestion from [3] is to extend data descriptions with various types of profiles. For instance, the data structural information is accompanied with constraints, input validation, and field semantics for the presentation or security. [1] then shows that such extensions are practical for use in code-based applications and can be derived by code-inspection. [1, 2] suggests to treat the information accompanying the data model not only as data structure, but as AOP join points that are used to determine a particular field presentation. In addition the application runtime context is considered together with the structural information, which together produce a join point model from which can be derived the context-aware UI. The advantage of [1] is that code-inspection applied to data models derive such information, avoiding manual work. [1] uses join points in a way that applies AOP-based transformation querying field join points and determines field presentation through templates. Next, it decorates the result through a layout, determined by a template.

The above approach can be considered from the perspective of platform-independence. As suggested in [2, 3], the join point model can be treated as platform-independent. The application context extends the information received from the data model and is not specific to a particular platform. Similar to GWT or AngularJS, data values are not specific to a particular UI platform.

On the other hand, presentation and layout templates use the target UI language components and thus contain platform-specific elements. The integration of templates with the join point model and data values must be considered by

a processor that has knowledge of a particular platform and is platform-specific and located at the client-side.

In order to separate these particles, the platform-independent part should be provided in a platform-independent format. Web-resources provide standard formats that are understood across platforms, and thus they naturally fit the goal. Platform-specific clients then provide presentation and layout templates and the processor.

Providing the entire join point model to clients can be impractical, mostly when considering that it can consist of internal information or information not relevant to the UI. Thus some kind of filter should exist. At the same time, it might not be known ahead of time, which join points are used to determine the presentation at the client-side. When considering that all the different clients perform the same "join point model"-to-"presentation" transformation, it essentially becomes repetition. Thus the transformation can be partially performed at the server-side, considering the unfiltered join point model.

The question is how to connect the transformation with templates that are at the client-side in a platform-specific format. It is possible to perform the "join point model"-to-"presentation template" transformation in a way that it considers an expected set of templates with a unique identifier. The result of the transformation gives each data field a template identifier and accompanies it in the provided join point model. A filtered fraction of the join point model is provided through a Web-resource, separating out information not relevant to given user access role, context, or information not relevant to UI, such us data identifier, version lock, business key, etc.

Security is considered at the server-side for both the provided join point model and values for the requested data instance and in a given context (user, location, time-frame, etc.). Thus only data 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 by supplying platform-specific components. It determines the layout and implements the processor. The processor requests the data values and join point model, giving the data structure from the server-side. Next, it populates the local presentation templates with received information. Additionally it is possible for the client-side to use multiple sets of presentation templates that are changed based on local context (readable forms, editable form, data table, report, list, etc.). It is also possible to decorate the data presentation with wizard-like components that collapse the presentation into multiple panels.

4 Platform-Specific UI Clients

Implementation of a platform-specific client requires implementing a processor that requests and interprets the join point model and data values received from the server-side. The platform-specific sets of presentation templates are defined, reflecting the expected set of template identifiers. These templates consider native components that provide expected functionality, capture constraints, input

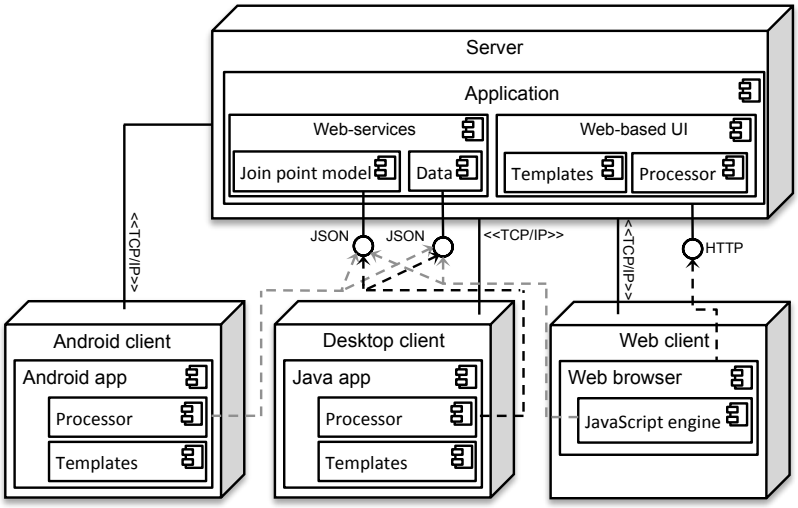

Fig. 1. Sample of deployment diagram considering three heterogeneous clients

validation, etc., given by the join point model. The advantages of platform-specific features include increasing UI usability (e.g., touch-based element selection). The appropriate presentation template is selected based on the suggested template identifier, provided by the join point model. Local context may influence the set selection. Templates for layouts are defined complying with the expected and supported screen-sizes. The processor further requests and embeds data values and has the ability to post them to the server-side.

Heterogeneous clients interpret the server-side, platform-independent information. No matter the particular processed data element, the same presentation templates apply for a particular platform. 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 client application processor 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 processor at the client-side, since the provided join point model reflects it and existing templates are reused.

On the other hand if the system extends to provide a novel, unexpected constraint or data type, such a change needs to be reflected across all platform-specific clients. Even though such an extension is rare since the join point model [3] based on the industrially-standardized set of constraints and validations, it should be considered ahead of time in the design. There is no prevention mechanism that disallows clients to choose custom presentation templates and not to follow the suggestion given by the server-side.

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. The deployment diagram is shown in Fig. 1. The web-based client UI is shown in Fig. 2, an Android mobile client UI is in Fig. 3

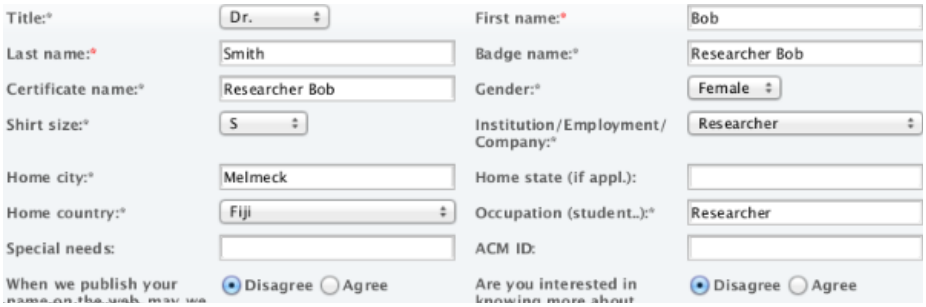

Fig. 2. Web-based UI

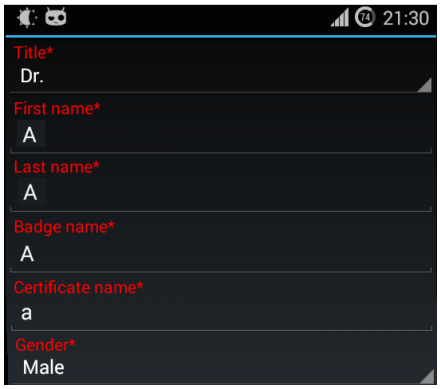

Fig. 3. Android-based UI

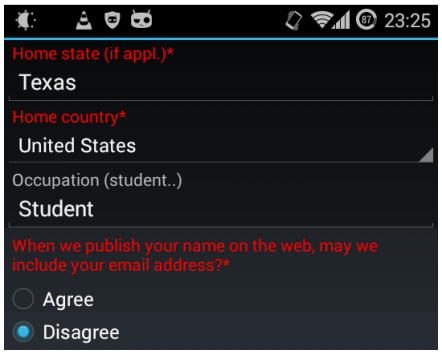

Fig. 4. Android-based UI

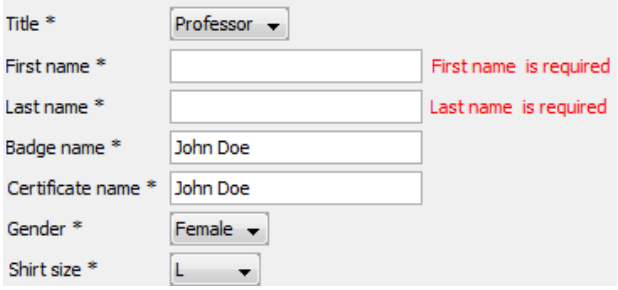

Fig. 5. Java Swing-based UI

and 4 and a standalone, Java Swing client UI is in Fig. 5. They all use the same application server, 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 JavaScript library. Various sorts of data presentations can be derived ranging from forms, read-only forms, tables, lists, etc. Templates designed for a particular platform can be reused across different applications on the same platform. The platform-independent data presentations adjust to the server-side application and its join point model. It allows caching the join point model for a particular time span. The application page-flow is left for custom, non-automated implementation.

5 Conclusion

This paper elaborates the idea of separation of UI data presentation particles into platform-independent and platform-specific parts. Platform-independent information is provided through a machine-readable format through web-resources at the server-side. Such separation simplifies implementation of heterogeneous clients, while reducing their development and maintenance efforts through supported information reuse. All clients immediately reflect changes of application data structures in the UI, since the provided join point model derives the actual structures though code-inspection. Our approach derives various types of presentations using native components, thereby increasing usability.

The limitation of the approach is that it applies to data presentations and thus page-flow is left for custom, non-automated implementation, although the page-flow provided in a platform-independent format is left for future work. Layouts are considered as a specific particle of heterogeneous clients, although platform-independent generalization is possible but also left for future work. Future work will also consider integration with AngularJS and evaluation in a large application. The approach will be considered for Service-oriented architecture and middleware interaction.

6 Acknowledgments

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

References

1. T. Cerny, K. Cemus, M. J. Donahoo, and E. Song. Aspect-driven, Data-reflective and Context-aware User Interfaces Design. *SIGAPP Appl. Comput. Rev.*, 13(4):53–65, 2013.
2. T. Cerny, M. Macik, J. Donahoo, and J. Janousek. Efficient description and cache performance in aspect-oriented user interface design. In *Federated Conference on Coumputer Science and Information Systems*, 2014.
3. T. Cerny and E. Song. Model-driven rich form generation. *Information-An International Interdisciplinary Journal*, 15(7, SI):2695–2714, July 2012.
4. K. Czarnecki and U. W. Eisenecker. Components and generative programming (invited paper). *SIGSOFT Softw. Eng. Notes*, 24(6):2–19, Oct. 1999.
5. B. Green and S. Seshadri. *AngularJS*. O'Reilly Media, Inc., 1st edition, 2013.
6. R. Hanson and A. Tacy. *GWT in Action: Easy Ajax with the Google Web Toolkit*. Manning Publications Co., Greenwich, CT, USA, 2007.
7. R. Kennard, E. Edmonds, and J. Leaney. Separation anxiety: stresses of developing a modern day separable user interface. *Proceedings of the 2nd conference on Human System Interactions*, pages 225–232, 2009.
8. G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *IECOOP'97-Object-Oriented Programming, 11th European Conference*, volume 1241, pages 220–242, 1997.
9. M. Macik, T. Cerny, and P. Slavik. Context-sensitive, cross-platform user interface generation. *Journal on Multimodal User Interfaces*, 8(2):217–229, 2014.