Evaluation of Separated Concerns in Web-based Delivery of User Interfaces

Tomas Cerny[1], Lubos Matl[1], Karel Cemus[1], and Michael J. Donahoo[2]

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

Abstract. User Interfaces (UI) play a significant role in contemporary web applications. Responsiveness and performance are influenced by the UI design, complexity of its features, the amount of transmitted information, as well as by network conditions. While traditional web delivery approaches separate out presentation of UI in the form of Cascading Style Sheets (CSS), a large number of presentation concerns are left tangled together in the structural description used for data presentations. Such tangling impedes concern reuse, which impacts the description size as well as caching options. This paper evaluates separation of UI concerns from the perspective of UI delivery. Concerns are distributed to clients through various resources/channels, which impacts the UI composition at the client-side. This decreases the volume of transmitted information and extends caching options. The efficacy is demonstrated through experiments.

Keywords: Separation of concerns, user interface, networking

1 Introduction

A User Interface (UI) defines the visual part of a computer application. Its design must not only consider usability and development efforts, but the UI must be performant and highly responsive to address growing demands in Rich Internet Applications (RIAs). The responsiveness to user requests has many influencing factors. Besides the UI design, it is influenced by page feature complexity, page size, and mostly by network conditions and HTTP delivery.

Traditional UI design approaches describe a particular UI page combining various concerns [1, 5] together. For instance, data presentation descriptions consider concerns, such as data structure, individual field presentation, user input validation, field layout, data values binding, security, etc. Although single-location, multi-concerns descriptions are easy to read to see the global picture, such an approach brings multiple disadvantages. When a single concern changes, other "tangled" concerns [1] distract designers from the modification since no explicit boundary exists in the description. The tangling further limits concern reuse [5]. Consider a situation when a page varies based on context. Certain variations can be solved through page conditionals, although many of the variations require designing a new, similar page. For instance a novel page description might be needed when field presentation changes with the context, the user input validation is user-sensitive, layout changes with the screen resolution, etc.

Separating UI concerns make it possible to describe each concern individually and reuse them across different data types presented in the UI. Such reuse could reduce the size of UI descriptions. Consequently, from the perspective of web delivery, providing clients UI concerns separately supports reuse and thus reduces the amount of transmitted information. Furthermore, since each concern is provided to clients separately, the client may cache a given concern for a particular time span, which is not possible in the conventional approach that delivers tangled concerns to clients.

This paper evaluates the idea of separated concern delivery in UIs for data presentations in Section 2. UI concerns are provided through multiple channels and combined at the client-side. The impact on transmission size, UI responsiveness and performance is considered and compared to traditional UI delivery in Section 3. Next, we evaluate extended UI caching options from the perspective of client-side caching and content-delivery networks (CDN). We also evaluate the impact on the server-side considering the server CPU usage. Related works are presented in Section 4. Finally, our conclusion is given in Section 5.

2 Related Work

Many approaches [1] consider Model-Driven Development (MDD) [3] with the expectation that a model captures all sorts of information at a single location. In MDD the UI is generated from the model [1]. It reduces design efforts for usual UIs, but it has limitations. When dealing with context-aware UIs, the runtime generation might be performance inefficient. MDD does not effectively address cross-cutting concerns [7] since no generalized mechanism to address multi-model integration exists [2]. Similar to MDD, Domain-specific languages (DSLs) describe UIs [1], although they fail to address cross-cutting concerns [1].

Approaches addressing cross-cutting concerns are Generative Programming (GP) [7] and Aspect-Oriented Programming (AOP) [5]. GP suggests describing concerns separately in a DSL and combining them together through a configuration to offer multiple result variants. Its operates at compile-time, which might be inefficient for context-aware UIs. The AOP operates at runtime. An example AOP adaption to UI is given in [1]. The target UI presentation is described through templates; concerns integrate through an AOP-based transformation of the audited data considering the context.

The Google Web Toolkit (GWT)[3] partially addresses UI responsiveness. It separates the UI presentation and data. It is similar to our approach, but with limited separation of concerns. It uses Java for the UI description, compiling into JS. The resulting presentation has cacheable and uncacheable JS parts and calls

[3] GWT - http://gwtproject.org, AngularJS - http://angularjs.org; November 2014

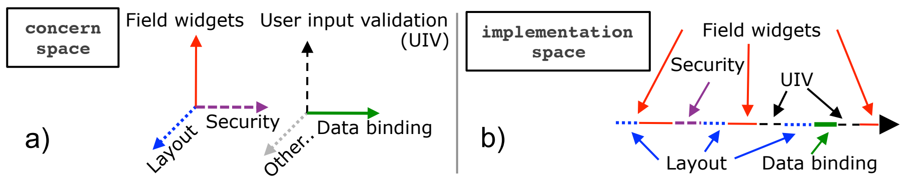

Fig. 1. Demonstration of UI cross-cutting concerns collapsing into a single dimension

Web-resource for data. The UI delivery is similar to conventional approaches. Similar to MDD, GWT faces issues with cross-cutting concerns and restated information. For instance, all the UI states are combined together at compile-time [2], which may result in a bloated UI description size.

AngularJS web framework[4], similarly to GWT, suggests data separation into another channel, although the presentation is left tangled together. In addition to GWT, it provides templating mechanism for content resolution, but it does not separate out the structure, context, or the presentation and layout templates.

3 Separation of Concerns in UI Data Delivery

Conventional programming languages, including object-oriented, provide various decomposition mechanisms. Such decomposition aims to separate concerns - information that impact the code of a particular program - into logical sections. Based on Kiczales et. al. [5], these languages lack the ability to effectively address cross-cutting concerns, which are aspects of a program that affect other concerns. One of the approaches that effectively address cross-cutting concerns is Aspect-Oriented Programming (AOP).

Recent work on separation of concerns in UI [1] considers the AOP-based UI design (AUI). The goal of such an approach is to reduce UI development and maintenance efforts. Since AOP allows describing various concerns separately, it supports concern reuse across various data type presentations. The demonstration in Fig. 1*a*) shows possible cross-cutting concerns in the UI. Each of these concerns impacts the UI data presentation and can be reasoned independently, in separate dimensions. When capturing these concerns in the implementation space, they collapse and tangle together into a single dimension [6], Fig. 1*b*).

The AUI design considers the application data model to be the main source of *join points* [8] that influence data presentation and UI concern integration [3] showing that it is possible to derive from such *join points* a variety of data presentations, and it is not specific to a particular platform. The data model is the subject of code-inspection [1, 4] for this reason. The obtained *join point* structure represents structural properties of system data, structured by fields. The *join point structure* can further be extended by runtime context (e.g., logged user, geo-location, rights, etc.).

To derive the UI for data elements, the *join point* structure is queried for its properties. A set of rules that perform the selection of integration templates uses an AOP-based mechanism consisting of *pointcuts* and *advices*. The *pointcut* specifies a query to *join points* of a particular field and context. When a *pointcut* finds a match, the associated *advice* suggests an integration template for the field presentation. Such an integration template uses the target UI language to describe the field presentation. References to the data structure and its constraints are resolved. Furthermore, it defines integration rules to integrate other concerns. An integration rule uses the same principle as transformation rules. Its *pointcut* resolves whether to apply the concern defined in an *advice*. A target layout is integrated through a layout template.

Through the separation of concerns, the AUI supports reuse. [1] shows reduction of 32% of the UI code in a large application. The UI code reductions are considerable from the development perspective, although the UI description that is delivered to clients is being generated and is equivalent to the conventional approaches. Although AUI supports concern reuse, it does not take into account the perspective of UI delivery. The separation of concerns collapses into a single, tangled description before it is delivered to the client-side, and thus the separation becomes lost.

An extension to the AUI approach that maintains the concern separation at the client-side is suggested by [2]. The perspective of UI delivery may be considered independent of AUI. The main difference for the web-delivery is that various concerns are delivered through multiple resources/channels.

The client requests an HTML page that should display particular data, e.g., multiple forms. The delivered page HTML description consists of page elements, but the forms are not described. Instead instructions on how to assemble and embed them are provided, and the required concerns are requested from other channels. The HTML description is accompanied with a JS library that contains the weaver. It takes the instructions and requests other concerns for the particular data elements to derive the aimed presentation for particular context. The design sketch for the description is provided by Fig. 2.

The particular application data definition from the data model determines the presentation structure as well as its constraints and validation rules [3]. This defines the structural concern, although the information must be provided to clients in a readable format. To avoid manual restatement of the structure information in a particular output format, the content can be derived through code-inspection [4, 3].

Besides the data structure, the application runtime context should be taken into account. For instance, guest users can see less fields that registered users, the input validation differ in given contexts; field presentation may be influenced by user location; etc. The application context may restrict existing structure or override its properties using the Annotation Driver Participant Pattern [6]. The client receives the Context-aware Structural Information (CSI) from a server in a machine readable-format (JSON, XML).

Different data and contexts give different CSIs, although for given context and data type the CSI does not change. The CSI determines the structure of the data presentation. It is structured by data fields and for each field gives detailed information of its properties, constraints, validation rules, etc. The CSI is used by the weaver to determine presentation for each particular field.

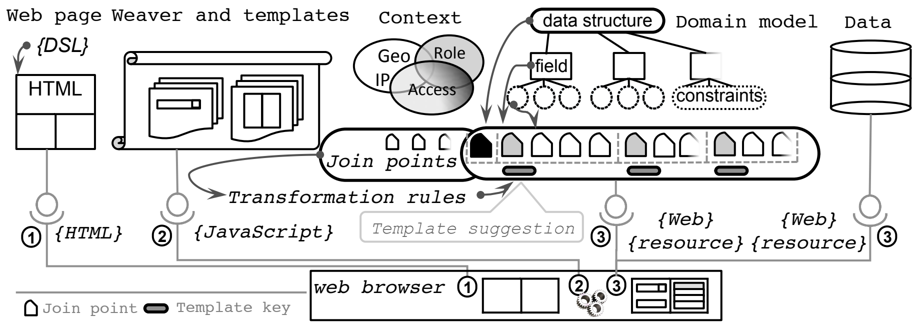

Fig. 2. Architecture of the concern-separating approach of UI data delivery. The web browser requests an HTML page (1) that is supplemented with a JavaScript library (2) responsible for the data presentation composition. The HTML page indicates data to show and the library (3) requests the CSI and data values and produces its presentation.

Similar to AUI integration templates, a set (or sets) of templates is delivered through to the clients as a JS library with templates. Template content can be resolved in the context of CSI, although the question to answer is how to select a particular template for given field. AUI [1] suggest defining small a number of generic rules that perform the selection. These rules could be part of the client-side weaver and work directly with the CSI, although it would raise its complexity at the client-side and at the same time the rule could not be based on information that was not provided from the server in the CSI. Instead the template selection executes at the server-side, and the suggested template identifier accompanies the field in CSI.

Layouts are defined through templates and delivered as a JS library. The weaver processes the CSI to determine the presentation structure and the content of field templates. Next, the weaver determines layout template and decorates the result. The data values are requested in parallel to CSI, integrated to the assembled component, and embedded at the instructed location.

The separated concerns are maintained at the client-side, and all JS sources can be cached. The weaver can cache in HTML5 LocalStorage the CSI for a particular time span and context. The weaver can further consider local context. When multiple sets of templates are provided, switching the page from read/write modes, resulting in changes to layout or presentation template set, does not require reloading other concerns than data values. Requesting the same page for a different data instance in the same context requires only loading new data values, although this is influenced by the context-awareness. The data and CSI can be even machine processed and reused by native clients, such as Android.

4 Experiments

Out experiment evaluates the Concern-Separating Approach (CSA) from the perspective of page load time, specifically caching and volume of transmitted information related to data presentations. A fragment of an existing production-level application using the conventional approach is compared with the CSA. The person profile pages from ACM-ICPC[4] contest managements system is considered. First, a subset of the page with 21 form fields is evaluated; next a 42-form field version is considered. The application is built on Java EE 6 (JDK 7) using JSF 2.1 and the PrimeFaces 3.4 library. The same environment, UI resources, and logical structure of UI elements for the data presentation is used. The evaluated application is deployed on a server in Waco, Texas with 8 cores of 2.4 GHz, 16 GB RAM, and network access of 645/185 Mbits/s download/upload (D/U). The client is in Prague, Czech Rep. with 4 cores of 2.3 GHz, 16 GB RAM and 10/6 Mbits/s (D/U). For the content-delivery network (CDN), there is a server in Nuremberg, Germany with 2 cores of 3.4 GHZ and 3 GB RAM with 200 Mbits/s (D/U). The round-trip time (RTT) between client and server is 150 ms and client and CDN is 20 ms. 50 measurements are made, and the average with the standard deviation are provided in Tables 1-3.

The first evaluation with the shorter conventional page (Table 1, Row *a*) has a main document size of 74.4 KB, which compresses to 9.2 KB through gzip. In total the page calls 10 requests (including static resources), the transmission has 218 KB, and no caching is involved. Tables 1-3 show load times for Chrome$^{37.0.2062.122}$/Firefox$^{32.0.2}$/Opera$^{24.0.1558.53}$ web browsers (including JS processing). The CSA (Table 1, Row *b*) makes 15 requests. The main document size reduces to 3.3 KB (1.3 KB compressed), although it additionally loads a JS library (3.3 KB compressed), and data (1 KB compressed) with CSI (4 KB compressed) from web-resource. The transmission has 218 KB. Notice that additional resources load in parallel. The processed size of UI is considerably smaller (Table 1, last column). Row *c* gives the percentage improvement of page load times at around 10%. The data presentation description volume reduces by 61%.

The longer, 42-field conventional page has a main document size of 98.9 KB (compressed 11.1 KB), as shown in Table 1 *d*. The CSA in Table 1 *e* requires 23 requests since it requests information from 6 data instances. The main document size is 5.3 KB (compressed 1.6 KB), the CSI 6.5 KB, and data 2.7 KB. Row *f* shows around 10% improvement in page load time. The data presentation information volume improves by 62%.

The Table 1 *a*-*f* shows improvement in the page load times and reduction in the processed UI description volume. The compressed transmission is equivalent, although both sides work with the original size. The number of requests in CSA grows, because multiple data elements are involved for CSI and value requests.

When we consider browser cache, the load times drop even further. The CSA has the advantage of concern separation, making it possible to cache the templates and CSI. The conventional approach delivers the "tangled" main document, which is 9.2/11.1 KB for the shorter/longer version (Table 2, Rows *a*,*d*). The volume of data presentation description is the same as the uncached version.

The cached CSA transmits the main document and data values. It gives 2.4/4.2 KB in 3/7 requests for the shorter/longer version (Table 2 *b*,*e*). There is a 15-20% improvement in page load time. The RTT is the dominant factor. The UI data presentation volume improves by 93% and 62-74% in the transmission.

[4] ACM-ICPC Contest management system, http://icpc.baylor.edu, December 2014

Table 1. Page load measurements

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

Table 2. Page load measurements

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

Table 3. Simulation download evaluation

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

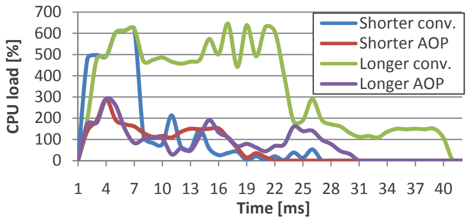

Fig. 3. Server stress-test CPU (no cache)

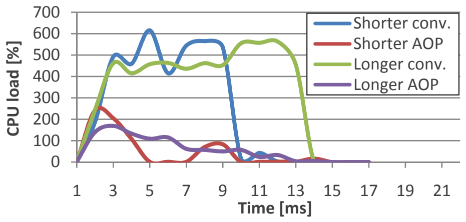

Fig. 4. Server stress-test CPU (cache)

A simulation involving browser request traces considers the network impact. This does not consider UI construction nor resource decompression. Results in Table 3 show simulation with no cache (*a*), cache (*b*) and CDN (*c*). Although *a* and *c* show improvements comparable with web browser results, *b* is worse. The explanation is the dominant RTT, which in the conventional approach occurs once but in CSA multiple times and multiple data element requests are made. Notice this experiment only involves network communication.

The server-side impact evaluation uses also the browser request traces simulation and stresses the server with 100 clients loading a particular page simultaneously. The server CPU load is measured for all four cases, not cached and cached. An Unix tool “sysstat” samples the CPU every second during the stress test. The results in Fig. 3 and 4 show that CPU load is higher and longer for the conventional approach, which match with the higher information volume transmission for the approach and delegated UI composition to clients.

5 Conclusion

This paper provides evaluation of separated concern, web-based delivery in UIs. Data presentations are provided to clients through multiple resources/channels. This approach brings clients the ability to reuse particular concerns and thus reduce the amount of delivered information. Clients gain the ability to cache given concerns for a certain amount of time and request only the concern(s) that changes. The server-side has to process less data, and its concern weaving responsibility is delegated to clients. The evaluation results confirm the expectations based on the approach.

The limitations for our approach include that it does not address page-flow, applied solely to data presentations, and thus it must build on the top of another UI framework. To improve the resource delivery, it is possible to aggregate CSI-related resources into a single request for multiple data elements and deliver it at once. Separation of platform-specific and platform independent channels is considered for future work.

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. A profile approach to using uml models for rich form generation. In *Information Science and Applications (ICISA), 2010 International Conference on*, pages 1–8, 2010.
4. 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.
5. 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.
6. R. Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications.* Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.
7. 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.
8. M. Stoerzer and S. Hanenberg. A Classification of Pointcut Language Constructs. In *SPLAT'05: Workshop on Software-Engineering Properties of Languages and Aspect Technologies*, 2005.