A Conceptual Framework for User Input Evaluation in Rich Internet Applications

Matthias Book, Tobias Brückmann, Volker Gruhn, and Malte Hülder

Applied Telematics/e-Business Group, University of Leipzig
Klostergasse 3, 04109 Leipzig, Germany
{book,brueckmann,gruhn,huelder}@ebus.informatik.uni-leipzig.de

Abstract. The more complex an application's user interface is, the more important is the need to guide users filling out the forms—typically by highlighting invalid input, showing/hiding or enabling/disabling particular fields according to business rules. In Rich Internet Applications, these reactions are expected to occur virtually immediately. We discuss aspects to be considered for consistent reactions to user input, and describe how evaluation rules can be formulated for model-driven development.[1]

1 Introduction

The user interfaces (UIs) of web-based information systems tend to mirror the complexity of their underlying business processes: In areas as diverse as e.g. market research, insurance claims or reinsurance underwriting, users need to enter a lot of structured data that must obey a variety of domain-specific constraints.

To support users in working efficiently with complex forms, UIs typically react to input with local changes in individual UI widgets (text fields, list boxes etc.) such as making the user aware of invalid input by highlighting affected widgets, decreasing visual complexity by hiding unnecessary widgets, or guiding users by enabling or disabling input in particular widgets. Rich Internet Applications (RIAs) enable instantaneous input evaluation and interface updates, and can thus provide immediate feedback and guidance to users.

In this paper, we discuss the aspects that influence a UI's reaction to user input (Sect. 2), and present a behavior model that includes dependencies between UI reactions such as handling incomplete input, prioritizing validation issues, and considering visibility in validation (Sect. 3). For use in practice, we briefly describe our Cepheus framework that automatically generates evaluation logic following this model based on rules specified by domain experts, eliminating the need for manual implementation (Sect. 4). We conclude with an overview of related work (Sect. 5) and a summary of our contributions (Sect. 6).

[1] This work was supported by a technology support grant from the European Regional Development Fund (ERDF) and funds of the Free State of Saxony. The Applied Telematics/e-Business Group is endowed by Deutsche Telekom AG.

M. Gaedke, M. Grossniklaus, and O. Díaz (Eds.): ICWE 2009, LNCS 5648, pp. 275–282, 2009.
© Springer-Verlag Berlin Heidelberg 2009

2 Specification of Input Evaluation

2.1 Interface and Data Model

A web application's interface model is characterized foremost by the **UI widgets** displayed on its pages. Often, multiple widgets will jointly describe a particular semantic entity from the business domain (e.g. a group of radio buttons for 1-of-n selection, or a group of text fields for entering elements of a postal address). To model such relationships, we allow widgets to be contained in hierarchically nested **containers** that also govern the layout of the interface's **pages**.

To store the entered content, all widgets must be bound to **variables** in the application's data model. While widgets can only produce string input (as this is the serialized format universally used to exchange data between web application components), the data model's variables have certain **types** (e.g. Boolean, integer, floating-point, text, date etc.).

2.2 Evaluation Aspects and Rules

To formulate rules governing the evaluation of the information in the interface and data model, several orthogonal aspects have to be considered: Evaluation rules can serve different **purposes**—in this paper, we will focus on deciding *validity*, *visibility*, and *availability* of widgets, which are usually closely tied to UI **reactions** such as *highlighting* violating widgets, *hiding* invisible widgets, and *disabling* (e.g. "graying out") unavailable widgets, respectively.[2]

At the core of each evaluation rule must be an **expression** that describes the actual evaluation of certain values in order to arrive at a decision for one of the above purposes. While such an expression may consist of nested terms performing *comparisons*, *arithmetic*, *boolean* or *string operations* on literals or variables from the data model, it must ultimately resolve to a boolean value indicating the outcome of the decision.

Regardless of its purpose, any evaluation rule must relate to certain **subjects** on which the respective reaction shall be effected. For increased flexibility, we allow that subjects can not only be individual *widgets*, but also groups of widgets contained directly or transitively in a particular *container*. Note that the subject widgets do not necessarily need to correspond to the expression's input variables.

For the purpose of input validation, we must consider several additional characteristics. First, we can distinguish several **levels** of validation that depend on each other: The most basic level is checking for the *existence* of any input in a required field. Next, the *technical* check concerns whether a particular input string can be converted to the associated variable's type. Finally, performing any *domain-specific* validation of the input is only sensible if the previous two validation levels were satisfied.

[2] We can also conceive other purposes of user input evaluation, such as deciding on navigation options. However, we will focus on the above-mentioned purposes here since their reactions are more interrelated with each other, and they pose more interesting challenges in RIAs as they may impact a page's Document Object Model immediately, as opposed to navigation choices.

Our experience shows that in practice, it may be inconvenient or even impossible for the user to satisfy all validation rules immediately—rather, we identified four common **triggers** upon which different sets of validation rules can be sensibly checked and enforced: Validation may occur upon a widget's *"blurring"* (i.e. losing focus) when the cursor is moved to another widget; upon *leaving* a page in order to jump to the next or previous page in the dialog; upon *saving* the data entered so far as a draft version, in order to prevent data loss or continue working on the dialog at a later time; and finally upon *committing* all entered data in order to complete a task in a business process. By staging the validation through associating rules with appropriate triggers, developers can strike a balance between business requirements and usability considerations, ensuring data integrity while maintaining users' flexibility in working with the application.

In a similar vein, experience shows that typically not all rule violations are equally serious: Depending on the business semantics of a rule, developers may choose to associate a certain **severity** to it. We distinguish informative, warning and error rules in our evaluation specification, in order to tailor the interface's reactions to different severities, as we will see in the following section.

When formulating input evaluation rules, developers need to specify all of the above aspects (expression, subjects, level, trigger and severity) for the purpose of validation. In visibility and availability rules, only the expression and subjects must be specified, as their evaluation is always triggered immediately upon a widget's blurring, and we cannot distinguish different levels and severities.

3 Behavior of Input Evaluation

Having introduced the elements of input evaluation rules that developers need to specify at design-time, we will now discuss how these static specifications govern the dynamic behavior of an application at run-time, and how different rules affect each other. Anytime an evaluation is triggered, we need to (1) update the data model with the contents of those widgets that are technically valid; (2) validate the data model according to domain rules, and update the list of known issues; and (3) update the UI to reflect visibility, availability and issues of widgets.

In the following subsections, we will describe these steps in more detail. In this process, three data structures will be dynamically updated at run-time: The **contents** currently entered into the widgets of the interface model, the **values** currently stored in the variables of the data model, and the identified **issues**, i.e. the subset of all validation rules that are currently violated by any given input.

3.1 Data Model Update

Any time an evaluation is triggered (i.e. upon leaving a field or a page, or before saving or committing the dialog's data), we first need to update the data model according to the contents entered into the widgets affected by the trigger. The evaluation logic needs to implement the following algorithm for this purpose:

IF a widget is visible AND contains input THEN
  IF the input has the expected type THEN
    store the input in the variable associated with the widget
  ELSE leave the associated variable's current value unchanged
ELSE render the associated variable undefined

This way, we ensure that input is only included in the data model if its type is actually suitable for storage there; that incorrect input cannot overwrite previously stored data; and that any absence of input is reflected in the data model.

3.2 Data Model Validation

In the previous step, we have ensured that only technically sound input (i.e. input of the proper type) is accepted into the application's data model. Now, we still need to check if that data complies with the existence and domain-specific rules, and potentially signal any validation issues.

Existence Validation. When checking existence validation rules, we must not just check for the presence of content in a widget, but also take into account whether that widget is actually visible: We define that an existence rule is satisfied iff the respective widget contains input or is invisible. By taking the visibility into account when checking required fields, we eliminate the need for the developer to explicitly specify this connection in every rule, as it would be nonsensical to require input in a field we have hidden.

Domain-Specific Validation. When checking domain-specific properties, we need to arrive at a validation result in a way that takes both business rules and usability factors into account: In complex forms, subjects to which the validation pertains may be invisible, or variables on which the validation depends may still be undefined as the user makes his way through the form. We therefore define that a domain-specific validation rule is satisfied iff its expression evaluates to *true* or all its subjects are invisible. In evaluating the rule's expression, we should strive to arrive at a meaningful result even if some of the input variables are still undefined. In our model, any non-Boolean term that encounters an undefined parameter will therefore return an "undefined" result. In a Boolean OR term, meanwhile, we consider undefined parameters as $false$ values, and in a Boolean AND term, as *true* values, in order to let the result depend only on the other operand, thereby neutralizing the undefined part of the expression. This way, a term that returns an undefined result due to missing input parameters has no effect, so any empty widgets are not validated until they are filled—a behavior that we would intuitively expect from a dialog that is not yet filled completely. (Of course, an empty widget declared as required input would already be reported as invalid by the existence rules discussed before.)

Issue Tracking. To react to all validation issues consistently, regardless of when they occurred, we keep track of all rule violations in a central set. Anytime an evaluation is triggered, we perform the following two updates on this set:

1. add rules just found to be violated upon this triggering occasion
2. remove rules found to satisfied

By considering the triggering occasion when adding, but not when removing issues, we ensure that rule violations are not admonished until the time deemed appropriate by the developer, but that they are removed immediately when the violation is remedied. We found this behavior more intuitive for users than maintaining an old error message until the next trigger occasion, even when the user had already fixed the problem.

3.3 User Interface Reaction

Finally, our behavior model must define how the UI reacts to the various conditions that arise from validation results, visibility and availability of widgets:

Issue Notifications. We found it intuitive to signal validation issues in two ways: At the top of each page, the UI displays a concise list of human-readable explanations for all violations that were identified on the current and other pages. In case a particular set of subjects violates several rules, we display only the most severe issue to reduce clutter. To further aid the user in identifying invalid input, we highlight the respective widgets in a color corresponding to the severity (e.g. red for errors, orange for warnings, blue for information). Two relationships influence this coloring scheme: Firstly, if the subject of a rule is not an individual widget, but a container, the issue is assumed to apply to all directly and transitively contained widgets, which are all colored accordingly. Secondly, if a subject is affected by several issues (through several rules or inclusion in an affected container), it will be colored according to the most severe issue applying to it.

Visibility. In formulating the evaluation algorithms earlier, we have already relied on an indication of whether a particular subject is currently visible, but still need to define precisely how that decision is made: For any given subject (i.e. widget or container), we define that it is visible iff all visibility rules applying directly to it are satisfied, and if the container that it is contained in is visible. Note that this condition implies transitive dependency on the visibility of all containers in a subject's nesting hierarchy, allowing developers to conveniently hide or show whole groups of semantically related widgets if necessary.

Availability. Whether a widget is "grayed out" or editable is determined by availability rules that are specified and evaluated analogously to visibility rules. While visibility affects the data model, availability is a pure interface reaction that does not affect how data is validated or stored. The navigation buttons found on each page (typically, for navigating forward and backward in a dialog wizard, saving a draft of the current data, or committing it for further processing) are a special case insofar as they are implicitly associated with availability rules that do not need to be specified by developers: While a page contains validation errors triggered by leaving a field, the "previous page" and "next page" buttons are unavailable; while errors triggered by leaving a page are present, the "save

draft" button is unavailable; and during the presence of errors triggered by trying to save a draft, the "commit" button is unavailable, to prevent entering, storing or committing invalid input.

4 Implementation

The input evaluation logic described in the previous section was implemented in our Cepheus framework that generates presentation, validation and persistence logic from models created by domain experts in a visual editor. As Fig. 1 shows, views (based on the ICEfaces framework [1]) and evaluation rules are derived from the specifications at deploy-time. At run-time, the views will trigger validation, visibility and availability checks based on the specified rules and the values entered into the data model, and update the presentation accordingly by highlighting, showing, hiding or graying out the GUI widgets.

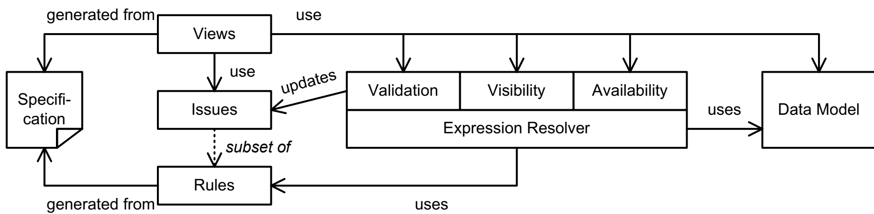

Fig. 1. Architectural overview of input evaluation in Cepheus

The screenshot in Fig. 2 illustrates an example system's behavior, where the visibility of the last three questions depends on the answer to the first question, and the allowed expenses have been limited to a certain amount that is exceeded here. Any changes in the input fields that affect the visibility or validation rules are immediately reflected in the user interface.

At this time, we have only anecdotal data about the time and effort savings that can be gained when using the Cepheus editor and framework instead of manual implementation. Initial experiences from employing a Cepheus prototype in an industry project from the market research sector (which is characterized by the need for frequent roll-outs of new web-based questionnaires) back our expectations that enabling domain experts to specify user interface layout and validation directly can reduce the time to market, since no transfer of domain knowledge to application programmers is required. Regarding performance implications, we do not expect a significant impact since all necessary rule-checking code is generated at deploy-time, so no expensive parsing occurs at run-time.

5 Related Work

Virtually all approaches for modeling and developing web applications provide means for realizing some form of input validation. To name just a few, in

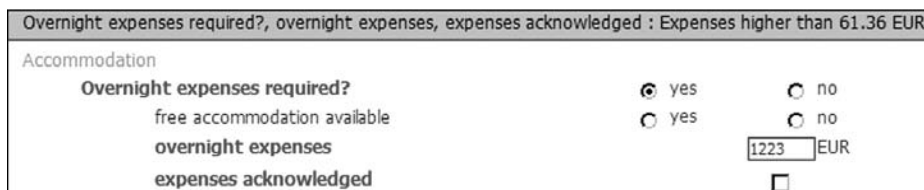

Fig. 2. Screenshot excerpt from a Cepheus-based application

WebML [2], validity predicates are properties of entry units in the hypertext model, using an expression language that supports comparisons of input fields with constants or other field contents. UWE [3] enables designers to specify when and where fields should be validated in its process flow and process structure models, but leaves the actual implementation of these rules to the developer. OO-H [4] provides means for validating the type of input fields; its "visible" and "hidden" attributes however have slightly different semantics from our approach.

Recently, these approaches have also been extended to support the modeling of RIAs: Comai and Toffetti Carughi [5] discussed how to extend WebML to capture more fine-grained user interaction with page elements; Meliá et al. [6] introduced corresponding structural and behavioral models in OOH4RIA; and Preciado et al. [7] combined UWE with the RUX-Method, an approach that focuses especially on the spatial, temporal and interaction aspects of Rich Internet Applications [8]. However, the validation models do not seem to have changed in the extension of these approaches' scopes.

Looking at representatives of popular web application frameworks, Struts provides a number of built-in validators for simple type and range checking, and allows the formulation of more flexible expressions in the `validator.xml` file [9]. In Spring, developers can provide custom validation classes implementing the `Validator` interface [10]. The Seam framework [11] relies on constraints defined in the data model using the Hibernate Validator. Common AJAX frameworks such as ICEfaces [12] or RichFaces [13] typically provide means for adding validation rules at different points in the request lifecycle as well.

While all approaches provide hooks for validation rules, their actual formulation is typically so technical that it requires a developer's rather than a domain expert's skills. In particular, any interdependencies between rules (e.g. visibility vs. validation, or the handling of different issue severities), are not supported by these models and frameworks themselves, but must be implemented explicitly.

6 Conclusion

In this paper, we identified several aspects that have to be considered in the evaluation of user input in RIAs for the purpose of technical and domain-specific input validation, widget visibility and availability. We have shown how these aspects are entwined with each other, and how they are incorporated in the Cepheus framework that generates user input evaluation logic automatically,

based on specifications that can be visually modeled by domain experts without the need for programmer assistance. We expect this approach to reduce implementation and maintenance efforts for RIAs considerably, and are striving to obtain more practical evidence to support this hypothesis.

References

1. ICEsoft Technologies, Inc.: ICEFaces, http://www.icefaces.org
2. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a modeling language for designing Web sites. Computer Networks 33, 137–157 (2000)
3. Koch, N., Kraus, A.: The expressive power of UML-based web engineering. In: IWWOST 2002: Proc. 2nd Intl. Workshop on Web-oriented Software Technology, pp. 105–119 (2002), http://www.dsic.upv.es/~west/iwwost02/papers/koch.pdf
4. Gómez, J., Cachero, C., Pastor, O.: Conceptual modeling of device-independent web applications. IEEE Multimedia 8(2), 26–39 (2001)
5. Comai, S., Carughi, G.T.: A behavioral model for Rich Internet Applications. In: Baresi, L., Fraternali, P., Houben, G.-J. (eds.) ICWE 2007. LNCS, vol. 4607, pp. 364–369. Springer, Heidelberg (2007)
6. Meliá, S., Gómez, J., Pérez, S., Díaz, O.: A model-driven development for GWT-based Rich Internet Applications with OOH4RIA. In: ICWE 2008: Proc. 8th Intl. Conf. on Web Engineering, pp. 13–23. IEEE Computer Society Press, Los Alamitos (2008)
7. Preciado, J.C., Linaje, M., Morales-Chaparro, R., et al.: Designing Rich Internet Applications combining UWE and RUX-Method. In: ICWE 2008: Proc. 8th Intl. Conf. on Web Engineering, pp. 148–154. IEEE Computer Society Press, Los Alamitos (2008)
8. Linaje, M., Preciado, J.C., Sánchez-Figueroa, F.: A method for model based design of Rich Internet Application interactive user interfaces. In: Baresi, L., Fraternali, P., Houben, G.-J. (eds.) ICWE 2007. LNCS, vol. 4607, pp. 226–241. Springer, Heidelberg (2007)
9. Apache Software Foundation: Struts Validator Guide, http://struts.apache.org/1.2.4/userGuide/dev_validator.html
10. SpringSource: Validation, Data-binding, the BeanWrapper, and PropertyEditors, http://static.springframework.org/spring/docs/2.0.x/reference/validation.html
11. Red Hat Middleware, LLC: JSF form validation in Seam, http://docs.jboss.org/seam/1.1GA/reference/en/html/validation.html
12. ICEsoft Technologies, Inc.: How to Use Validators, http://facestutorials.icefaces.org/tutorial/validators-tutorial.html
13. Red Hat Middleware, LLC: rich:ajaxValidator, http://www.jboss.org/file-access/default/members/jbossrichfaces/freezone/docs/devguide/en/html/ajaxValidator.html