Context-aware Input Validation in Information Systems

Karel CEMUS

Dept. of Computer Science, Czech Technical University, Technická 2, 166 27 Praha, Czech Republic

cemuskar@fel.cvut.cz

Abstract. *Enterprise Information Systems (EISs) maintain corporation data with the respect to business processes. These business processes constraint operations by preconditions and post-conditions. Unfortunately, contemporary approaches fail to effectively express and reuse these conditions and tend towards high restatements throughout the entire system.*

Alternative Aspect-Driven Design Approach (ADDA) focuses on advanced system decomposition and encapsulation. Its automated transformation simplifies information distribution into various parts of the system. This paper elaborates the transformation of the above-mentioned conditions into distributed context-aware user interfaces. This results with significant reductions to the error-prone process, and leveraged development and maintenance efforts.

Keywords

Enterprise Information Systems, Aspect-Oriented Programming, Input Validation, Business Rules.

1. Introduction

Every Enterprise Information System facilitates two major features. First, it maintains large amount of data with the respect to the business domain constraints, and second, it exposes API and a User Interface (UI) to access and modify them. Both these are heavily constrained by the business domain, i.e., the processes, their preconditions, post-conditions and model constraints. Furthermore, all conditions may vary in time, differ per user's role and context, or simply change during the system evolution.

Unfortunately, contemporary design approaches lack the support of business rules[1] transformation and reuse [4]. Instead, they repeat conditions throughout the whole system, from the UI through business services to the persistence layer [4]. Consequently, a single condition gets repeated many times in the system, which makes its maintenance very difficult. All changes are very error-prone as we have to locate all occurrences and update all places [6]. Things get more complicated when we consider the possible complexity of business rules and their possible dependency on time, user role, IP address, etc.

Alternative Aspect-driven design approach [2] focuses on extraction of business rules tangled throughout the whole system and their isolation in the single focal point called *knowledge base*. This registry aggregates all conditions in the system and groups them up into *business contexts*[2] [3]. Then the approach proposes the automated rules transformation into various target components such as the persistence layer and business services to deliver easily maintainable EIS without repetitions of the business rules. Unfortunately, the ADDA concept has not been fully demonstrated yet, there are many possible targets of the transformation [2] but exist only several proofs of the concept.

In this paper, we elaborate the ability of ADDA to transform business rules represented in the platform-independent knowledge base into the context-aware input validation in forms in the UI to avoid manual rules repetition and simplify the maintenance. The paper is structured as follows. The next chapter, Section 2, presents the issue of repetition of validation rules in contemporary approaches, while Section 3 browse through the related work. Section 4 discusses the ADDA concept. In Section 5, we introduce our implementation of the concept in the UI form validation. Section 6 demonstrates the simple example. Finally, we conclude the paper in Section 7.

2. Business Rules in EIS

EISs usually separate the logic into three layers [8]: 1) the persistence layer facilitating access into the persistent storage, 2) the application layer implementing the business logic, and finally 3) the presentation layer exposing API and UI enabling users and other systems accessing the data.

This architecture organizes the business logic into business operations. Each operation is a method in the middle layer and represents a step in a business process, i.e., such an operation has some business significance [4, 8]. We know

[1]By business rules we understand all preconditions and post-conditions of all processes and their operations identified in the business domain as well as all model constraints [3].

[2]By the business context we understand a set of preconditions and post-conditions defining the business operation, i.e., the step in the business process with some business value.

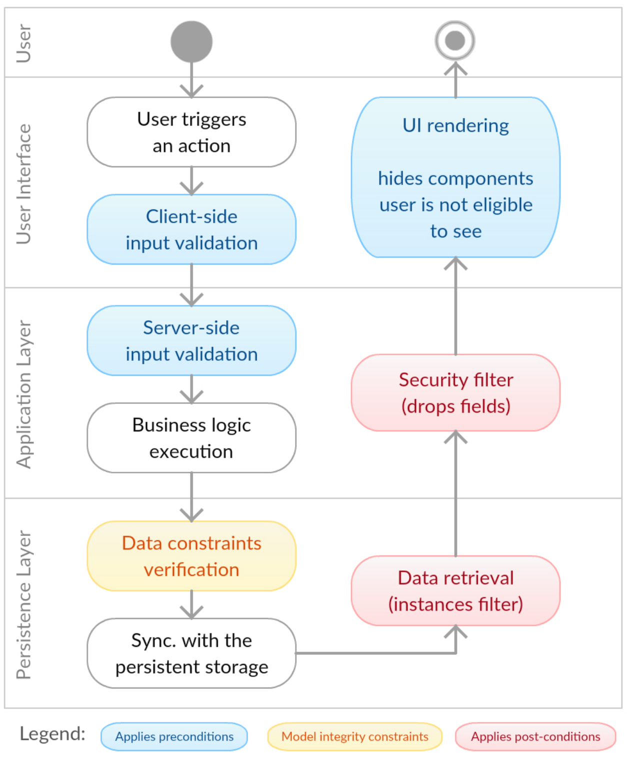

User
User Interface
Application Layer
Persistence Layer
User triggers an action
Client-side input validation
Server-side input validation
Business logic execution
Data constraints verification
Sync. with the persistent storage
UI rendering
hides components user is not eligible to see
Security filter (drops fields)
Data retrieval (instances filter)
Legend:
Applies preconditions
Model integrity constraints
Applies post-conditions

Fig. 1. Business operation execution

that every business operation, i.e., a step in a process, is restricted by some preconditions such as user's role, IP address, state of the application, and some model constraints. It also defines some post-conditions, i.e., what must be true when the operation finishes successfully. For example, consider the retrieval of *my opened issues* in an issue tracking system. This operation defines post-conditions such as returned issues are *assigned to me* and *opened*.

Every EIS is usually business-operation centric, i.e., every request invokes one business operation. The execution of the request is shown in Fig. 1. As we see, first, user initiates some action and operation's preconditions apply to validate input data before they are submitted to the server. Then, they apply again to verify data at the entrance to the application layer to ensure data and context's validity in case of some inconsistencies in UI. After the business logic executes, we verify model constraints again before saving the data to avoid their corruption in the persistent storage. Next, post-conditions apply. First, we restrict returned data to drop instances not matching the post-condition criteria, and then drop instances and fields the user is not eligible to see. Finally, when we render UI, we apply preconditions again to hide components and action buttons for which the user does not fulfill security requirements.

Business rules are used in many places throughout the whole system, which makes them difficult to encapsulate in a single place [11]. Furthermore, especially preconditions are used multiple times, and usually in multiple technologies[3], which makes their maintenance even harder. Unfortunately, neither contemporary technologies nor design focus on business rules reuse and force developers to repeat the rules manually instead. Keeping all places synchronized is very error-prone and requires significant efforts [9].

3. Related Work

The authors in [11] discuss concerns tangling in enterprise systems. Among concerns they identify widgets, localization, and business rules as an independent concerns difficult to describe next to each other. Instead, they observe that contemporary systems tend to tangle concerns together, which enforces their repetition and complicates the maintenance. As solution, in [12] they propose an alternative approach decomposing the concerns and automatically statically mixing them into resulting UI. This approach suffers from its inability to efficiently address context-dependent changes in business rules.

Difficulty of business rules representation in EISs is evaluated in [6]. It compares several techniques to represent the rules to ease the maintenance and concludes with the Visitor design pattern as the most efficient solution for the application layer. The proposed approach reminds the object-based approach considered in [4]. However, this solution is not more efficient than simple JSR 303: Bean Validation [1] with RichFaces[4] framework [4] transforming business rules annotating the model into UI. Either way, none of these covers context-aware rules, and the object-based approach does not allow rules transformation into UI. Finally, both these tend to be outdated as the modern UIs are more self-standing instead of being strongly dependent on the server-side.

The complexity of UIs is discussed in [5]. The authors also describe various independent but tangled concerns in UI including business rules but their proposed approach does not efficiently consider them. They fallback to JSR 303 with all their benefits and limitations. However, in [7] they extend their proposal to support distributed concerns delivery, which enables the modern self-standing UIs with reuse of server-side concerns description.

Alternatively, the authors in [15, 16] propose focusing on business rules, and maintain them on various levels of abstraction to have them maintainable by both developers and domain experts. They suggest maintaining them using CASE Tools and then referencing them from the code to allow their automated transformation.

Finally, Model-driven development represents another approach avoiding business rules repetition [14]. It suggests maintaining various models on a few levels of abstraction and having them automatically transformed into more spe-

[3]We validate user's input in the client-side technology, then verify it in the server-side technology, and finally we might declare database integrity constraints to protect our data storage.

[4]http://richfaces.jboss.org/

cific levels, which facilitates the information reuse. Unfortunately, this approach has difficult maintenance as it is tailored for the forward transformation. Any change performed to the code is difficult to propagate back into the model. Nevertheless, this approach does not conflict with the previous approaches, thus it is possible to use them together to minimize manual code duplication and information restatement.

4. Aspect-driven Design Approach

All approaches discussed in the previous chapter suffer from some limitations such as inability to express context-dependent business rules. Alternative ADDA introduced in [2] focuses on overcoming this issue. This approach generalizes several cross-cutting concerns such as those identified in [11], and applies Aspect-Oriented Programming (AOP) [13] to consider them as *aspects*.

Such thinking enables us describing all those concerns including business rules independently in the most efficient way, and storing them in a single focal point. Then we use techniques for code generation and transformation rules to weave all isolated aspects into proper *join points* to deliver the resulting system. Such decomposition and aspects definition significantly simplifies the system development and maintenance efforts as every piece of information is captured only once and is automatically propagated throughout the whole system at runtime.

The concept considers domain-specific languages (DSLs) [10] to be the most efficient method for aspects description. While it enables very efficient syntax and comprehensibility, it also introduces the significantly steep learning curve. Nevertheless, the concern representation is platform independent and requires various aspect weaver implementations to produce resulting components.

Implementation of aspect weavers enables both richer pointcuts and richer aspects. Both may consider contextual information such as user's identity, privileges, IP address, current application state, server load, etc., and use them to select proper join points to weave in. The only assumption is runtime weaving and efficient concerns transformation.

Although, ADDA introduces significant project overhead as uses multiple DSLs and requires various aspect weaver implementations, it shows very promising results [4]. Use of DSLs enables responsibility delegation to the domain experts, and a single focal point ease development and maintenance. Finally, simple description of concerns in isolation reduces their error-proneness and simplifies their testability.

5. Context-aware Input Validation

Business rules participate in various aspect weavers. For example, one weaver produces a context-aware data retrieval service [3], and the other produces validation logic in the application layer [2]. This chapter introduces the aspect weaver transforming the platform-independent business rules into the context aware input validation in forms in UI.

Modern EIS uses distributed UI with the single-page client-side application exposing the system to end users. Such a client is backed by the server-side application providing required data [7]. The client application let end users to browse and fill data in forms consisting of inputs mapped into the model fields. Every field and the model itself is constrained by several business rules depending on the current business context, i.e., set of rules attached to the current business operation. Moreover, business rules may vary based on time, user's privileges, and the application state. In consequence, validation rules are complex, and it is error-prone to maintain them in many places, especially when a single field may occur in multiple views, forms, layouts, etc. [5].

ADDA concept uses AOP to decompose the concerns and weave them back together. To follow this paradigm, we identify AOP components (aspects, join points, pointcuts and advices) in the UI input validation are reuse the business rules concern from the server-side.

By an aspect we consider the business rules. This cross-cutting concern is tangled in the system and often repeated. Our previous work shows the ability to extract it and organize the rules into business contexts efficiently described in some DSL such as JBoss Drools[5]. For input validation, we consider only preconditions from the business context. Postconditions are ignored because they apply in the different part of the process[6], as we show in Fig. 1.

Join points are places where to possibly weave in the concern. In this case, we identify several places:

① during form preparation before it is rendered to check security preconditions on fields (read/write access)
② on change and blur input events to validate input data
③ on changing event to provide hints, e.g., how many characters left when maximal length is defined
④ on form submit event to validate the form including form-wide rules

Advise is functionality to weave in. In case of validation, it is verification of a condition. For every form to render, developers address the business context, and we extract preconditions from it. The advise is a transformed single precondition, i.e., verification, if the condition is satisfied.

Pointcuts select subsets of join points, where to apply the particular advise. We use pointcuts based on the type of a business rule. For example, min and max value restrictions we bind to ②, while *OR* conditions to ④. Furthermore, when you apply the length restriction into the long text area,

[5]http://www.drools.org/

[6]Note: Model constraints are supposed to be independent on a particular operation and the current context as they define data integrity constraints. In consequence, they are not part of the context.

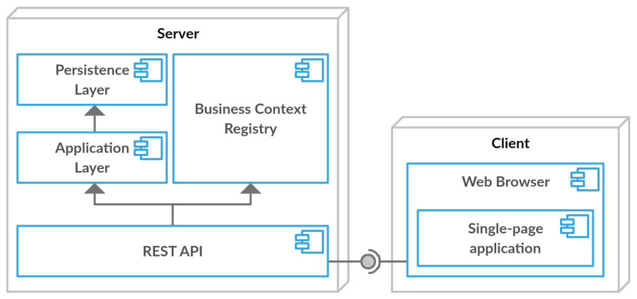


Fig. 2. Application architecture in the deployment diagram

we bind the validation to both ② and ③ to hint the number of remaining characters.

Weaving is being perform when the form is being rendered. First, we fetch the provided business context from the server, and then extract preconditions from it. Finally, we transform each precondition into an executable advice, assign the proper pointcut based on the condition type, and weave it into selected join points. In results, the form behaves exactly as if the developer would create it manually, although the business rules are automatically reused.

Context-awareness of this ADDA implementation is provided in two ways. First, declared business contexts may accept any contextual information the server provides [2] such as user's identity, which is reflected in the transformed validation rules. Contextual data are fetched from the server. Second, our pointcuts consider the current UI context they weave into. For example, when the precondition restricts the length and the UI widget is a *textarea*, then it weaves also into ③ to inform a user about characters left.

The ADDA implementation assumes a bit specific server-side architecture [2], as we show in Fig. 2. The persistence and the application layers are hidden behind the REST API exposing data and operations to clients. The presentation layer is divided into two components: 1) the REST API and 2) the single-page application, which is the contemporary approach to modern UIs. Besides this common layout, there is the business contexts registry providing business rules organized into contexts around business operations. It exposes them in the platform-independent format, and lets clients to transform them.

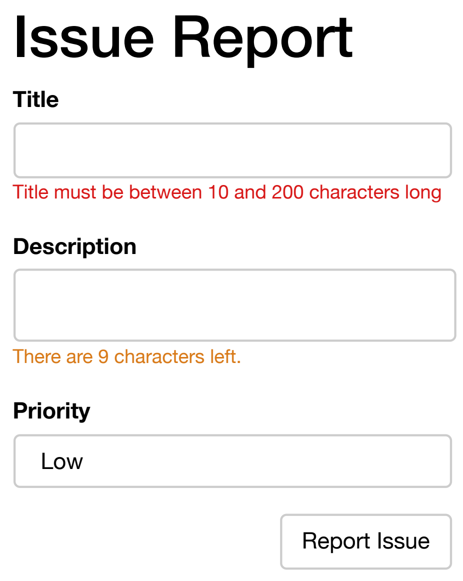
6. Input Validation Example

We demonstrate our weaver implementation on a small issue tracking system. The system maintains *Projects* consisting of *Issues*. Each issue maintains its *Comment*s. Each *User* stands in one of three roles: a user, a developer, or an administrator, and is assigned to zero or more projects and zero or more issues. Users report, resolve, and comment issues, create and archive projects.

In this application, we consider various types of preconditions. For example: min/max value, min/max length

Lis. 1. Business context: Report an Issue

```
rule "Report Issue" when Issue(
  # non-empty description, but min length
  title != null && title.length > 10,
  # max length
  title == null || title.length < 200,
  # pattern check on title
  title matches "^[a-zA-Z0-9 ]*$"
  # max length with long texts
  description == null || description.length < 1000,
  # min max value
  priority >= 1 && priority <= 3,
  # required
  type != null
) end
```


Issue Report

Title

Title must be between 10 and 200 characters long

Description

There are 9 characters left.

Priority

Low

Report Issue

Fig. 3. Preview of the Issue tracking application

restriction, required, and value pattern. We also consider conjunctions and disjunctions of preconditions. To support security checks, we compare user's roles against list of permitted roles. In practice, the business context may look like in Lis. 1. This example uses JBoss Drools framework and shows preconditions of *Report Issue* operation. It considers length of the title and the description, checks priority range and verifies the issue defines its type.

Having the business context, we annotate the HTML form to let the weaver know, where the model fields are. Lis. 2 shows issue reporting form. Notice the context name in the form tag; otherwise it is just regular form. Filed names are extracted from input names and aspect weaving is triggered on page load. The resulting UI is shown in Fig. 3.

There are no validation rules in HTML or in Javascript. They are fully fetched from the server and transformed from the platform-independent format into Javascript and bind to the events, which successfully shows another target for business rules transformation in terms of ADDA. However, there still remain some challenges. First, although we are able to reuse business rules, alternative approaches [5, 12] show, it is also possible to generate forms themselves from independently described concerns such as a model, localization, widgets, and a layout. Improvement of the weaver by sup-

Lis. 2. Form with meta-data to report an issue

```
<form data-business-context="Report Issue">

  <label for="title">Title:</label>
  <input type="text" name="title" id="title"/>

  <label for="description">Title:</label>
  <textarea name="description" id="description">
  </textarea>

  <label for="priority">Title:</label>
  <select name="priority" id="priority">
    <option value="1">Low</option>
    <option value="2">Medium</option>
    <option value="3">High</option>
  </select>

  <button type="submit">Report the issue</button>
</form>
```

port of more concerns would be significant benefit to both development and maintenance efforts as well as to testability. Second, example in Lis. 1 uses JBoss Drools DSL, which is not much convenient for this particular use [2]. Instead, we suggest tailoring own DSL better fitting the needs. That would allow simpler expressing of business contexts in more declarative and readable syntax, with indirect impact on transformation simplification.

7. Conclusion

Every EIS maintains data with the respect to the processes in his business domain. These processes define many business rules consisting of preconditions and postconditions of every operation. Unfortunately, neither contemporary design approaches nor technologies focus on business rules, their efficient maintenance and reuse.

In this paper, we introduce another component for business rules reuse using ADDA focusing on concerns separation including business rules and their automatic runtime weaving. In terms of AOP and this approach, we define context-aware input validation in forms in distributed user interface.

Our example shows a proof of the concept and demonstrates the ability to fully reuse validation constraint provided by a server as business context in the platform-independent format. This enables us using efficient DSL and delegate responsibilities to domain experts, while it automatically propagates the rules throughout the whole system. Together with the previous work, we are able to reuse the rules in the application and the presentation layer for the input validation, and in the persistence layer for output restriction, without their no manual repetition. Furthermore, the approach weaves them in runtime, which enables considering current user's and application's context. Contrary, this design approach introduces significant initial project overhead because it uses several domain-specific languages and requires complex aspect weavers.

In future work, we will extend this implementation by support of additional concerns such as model, widgets, and layouts to deliver comprehensive example of multi-platform context-aware forms in distributed user interface.

Acknowledgments

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

References

[1] BERNARD, E., AND PETERSON, S. Jsr 303: Bean validation. *Bean Validation Expert Group, March* (2009).

[2] CEMUS, K., AND CERNY, T. Aspect-driven design of information systems. In *SOFSEM 2014: Theory and Practice of Computer Science, LNCS 8327*. Springer International Publishing Switzerland, 2014, pp. 174–186.

[3] CEMUS, K., CERNY, T., AND DONAHOO, M. J. Automated business rules transformation into a persistence layer. *Procedia Computer Science 62* (2015), 312–318.

[4] CEMUS, K., CERNY, T., AND DONAHOO, M. J. Evaluation of approaches to business rules maintenance in enterprise information systems. In *Proceedings of the 2015 Conference on research in adaptive and convergent systems* (2015), ACM, pp. 324–329.

[5] CERNY, T., CEMUS, K., DONAHOO, M. J., AND SONG, E. Aspect-driven, data-reflective and context-aware user interfaces design. *ACM SIGAPP Applied Computing Review 13*, 4 (2013), 53–66.

[6] CERNY, T., AND DONAHOO, M. J. How to reduce costs of business logic maintenance. In *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on* (2011), vol. 1, IEEE, pp. 77–82.

[7] CERNY, T., MACIK, M., DONAHOO, M. J., AND JANOUSEK, J. On distributed concern delivery in user interface design. *Computer Science and Information Systems 12*, 2 (2015), 655–681.

[8] FOWLER, M. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[9] FOWLER, M. *Refactoring: improving the design of existing code*. Pearson Education India, 2002.

[10] FOWLER, M. *Domain-specific languages*. Pearson Education, 2010.

[11] KENNARD, R., EDMONDS, E., AND LEANEY, J. Separation anxiety: stresses of developing a modern day separable user interface. In *Human System Interactions, 2009. HSI'09. 2nd Conference on* (2009), IEEE, pp. 228–235.

[12] KENNARD, R., AND STEELE, R. Application of software mining to automatic user interface generation, 2008.

[13] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. *Aspect-oriented programming*. Springer, 1997.

[14] KLEPPE, A. G., WARMER, J. B., AND BAST, W. *MDA explained, the model driven architecture: Practice and promise*. Addison-Wesley Professional, 2003.

[15] MORGAN, T. *Business rules and information systems: aligning IT with business goals*. Addison-Wesley Professional, 2002.

[16] THEODOULIDIS, B., AND YOUDEOWEI, A. Business rules: Towards effective information systems development. *Business Information Systems–uncertain futures* (2000), 313–321.