Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2015) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

The 2015 International Conference on Soft Computing and Software Engineering (SCSE 2015)

Automated Business Rules Transformation into a Persistence Layer

Karel Cemus[a,]*, Tomas Cerny[a], Michael J. Donahoo[b]

[a]*Dept. of Computer Science, FEE, Czech Technical University, Technicka 2, Praha 6, 166 27, CZ*
[b]*Dept. of Computer Science, Baylor University, One Bear Place #97356, Waco, TX, 76798, US*

Abstract

Enterprise Information Systems maintain data with respect to various business processes. These processes consist of business operations restricted by business rules expressed as preconditions and post-conditions. Each rule must be considered and enforced throughout the system, from user interface to persistence storage. Such rule evaluation in multiple contexts results in both significant rule restatement and high maintenance complexity, as there is no single focal point for capturing and reusing these rules. In this paper, we apply the Aspect-Oriented Design Approach to the persistence layer to simplify business rules management, enforce business rules throughout the system and consequently decrease development and maintenance efforts. Our preliminary results show that it is possible to define business rules in a single place and then apply them automatically in a persistence layer. We retrieve data sets restricted by given operation post-conditions with respect to current execution context without any manual rule restatement. This paper provides a small case study emphasizing the benefits and future challenges.

Peer-review under responsibility of organizing committee of The 2015 International Conference on Soft Computing and Software Engineering (SCSE 2015).

Keywords: Enterprise Information Systems; Business Rules; Model-Driven Development; Aspect-Oriented Programming

* Corresponding author
E-mail address: cemuskar@fel.cvut.cz (Karel Cemus)

Peer-review under responsibility of organizing committee of The 2015 International Conference on Soft Computing and Software Engineering (SCSE 2015).

1. Introduction

Every Enterprise Information System (EIS) maintains a significant amount of data, interacts with many users, and implements various business processes with business rules. Although there are many possible system architectures, there are always rules to be considered in multiple places through a system. Consider domain model restrictions (rules)[a]. They restate in two dimensions in the three-layered architecture[1]: horizontal and vertical. They horizontally restate among operations of a single layer, e.g., in the service layer, all operations changing a domain object must consider its restrictions. Also, they repeat in all vertical layers of a system:

- User Interface (UI) as client-side input validation
- Service layer as server-side input validation
- Persistence layer to store only valid instances

Furthermore, this is only a fragment of possible repetition of business rules. Another example involves data retrieval. Every business operation returning data from storage (e.g., a database) declares rules restricting the returned subset. For instance, to enable recovery, data are never actually deleted in an EIS; instead they are flagged as deleted. The filter excluding all flagged entries is a typical example of a highly repeated restriction. These rules restricting a returned set of data might be seen as operation post-conditions because they apply on operation output. Usually in conventional approaches, developers repeat these rules manually without any support of automated reuse of rules shared across multiple operations or layers[2]. We discuss rules reuse in a persistence layer in more detail and illustrate it with examples in Section 2.

As we show, there is significant business rule restatement in development of an EIS. Unfortunately, common approaches often lead to manual repetition, which significantly impacts development and maintenance efforts. To minimize manual restatement, previous work proposes the Aspect-Oriented Design Approach[3] (AODA) refining the three-layered architecture[1] to ease reuse of captured information. This novel approach decomposes a system to enable reuse of shared aspects and allow their automated runtime integration. Section 3 presents a brief description of this approach. Unfortunately, the authors neither specify how to implement the approach nor how to apply it to existing systems[3]. It only introduces the fundamental idea and leaves the rest for future work.

In this paper, we begin the exploration of the actual application of AODA by automating business rule reuse in a persistence layer when an EIS integrates with a database. In Section 4, we refine the terminology used in AODA and introduce the idea of a Business Context. As previously shown, there are plenty of scenarios when a single rule affects multiple retrieval operations. In Section 5, we present our preliminary results, showing that it is possible to efficiently reuse independently declared business rules in a persistence layer without their manual restatement. Such enhancement significantly improves system development and maintenance efforts and simplifies application testing.

2. Background

We begin by motivating the need to reuse business rules. We discuss common EIS design with emphasis on business operation preconditions and post-conditions. To illustrate the challenge we introduce a simple example.

Business rules represent a fundamental part of an EIS. They affect all components of a system, from the database through domain model and business operations to the UI. Since by nature such rules cross-cut the system, it is difficult to capture and integrate them into a system as well as maintain them. Common implementation platforms for EISs such as Java Enterprise Edition and Microsoft .NET provide little support business rules representation and reuse except for a few exceptions (Section 3).

During the design phase, we decompose an EIS into use cases and then into business operations. For each operation, we define a set of preconditions and post-conditions as a specification of expected behavior[4]. Unfortunately, these conditions have to be considered throughout the system to deliver reliable and intended behavior. To illustrate, consider a small Issue Tracking System. Fig. 1 shows its domain model, consisting of four entities and three enumerations. Let us have two following use cases (UCs):

---

[a] We use Martin Fowler's terminology[1]. Similar used terms to *domain model* are *data model* and *business objects*.

UC 1: Assign user to an issue	UC 2: List user's assigned issues
Preconditions: - a current user is a project leader - a user is logged in Post-conditions: - developer of the project was assigned	Preconditions: - a user is logged in Post-conditions: - returned issues are opened - returned issues are not deleted - returned issues are assigned to the user

Now let us look at an implementation. We must check preconditions for both UCs in multiple places such as the UI and service layer; however, such vertical application is beyond the scope of this paper. Here let us focus on post-conditions, which affect a persistence layer more significantly.

To deliver UC 1 with autocomplete function, we require an operation returning all developers assigned to the project. The post-condition of the operation restricts the collection of returned users to those with a developer role and assigned to the project. Although the rule is not restated in a system, it has to be transformed into a persistence layer query language (e.g., SQL) to load only users meeting the condition. The other UC is more complex. Declared post-conditions might be often restated in a system, as they may occur also in other use cases. Such information, like what it means that an issue is opened or assigned to a particular user, then repeats in a system. It is much more difficult to maintain and keep consistent when rules are manually transformed into a query language.

AODA introduces an independent knowledge base (see Section 3) as a single focal point of business rules declaration. It uses Aspect-Oriented Programming (AOP) to integrate them into a system to prevent manual information restatement. In the following text, we apply this idea to automatically transform post-conditions to increase EIS development and maintenance efficiency.

3. Related Work

Leading, contemporary platforms for development of EISs include Microsoft .NET and Java Enterprise Edition. Both platforms use Object Relational Mapping (ORM)[5,6] frameworks to map a domain model into a relational database. This mapping simplifies use of persistent storage (a database) by making an abstraction level above it. For example, Hibernate[7] is an evolved ORM framework extending the Java Persistence API standard[6] by additional restriction declaration. It transforms all these restrictions, such as `@Email` or `@NotBlank`, into database integrity constraints.

Unfortunately, there is no existing framework efficiently dealing with business rules reuse. The Java Bean Validation standard[8] supports a small part of business rules, and there are frameworks reusing these restrictions in various layers. For example, RichFaces[b] transforms these restrictions into a presentation layer as client-side validation and Hibernate into database integrity constraints in a persistence layer.

Business rules reuse is introduced in[3] using an Aspect-Oriented Design Approach. Authors suggest applying AOP to decompose the system into isolated, independent aspects and describe them in their own convenient languages. Among various identified aspects such as UI layouts, UI widgets and internationalization, they suggest to isolate business rules in an independent knowledge base, capturing all business rules in a single place. Then the rules as well as other aspects might be addressed by the rest of the system and at runtime integrated in by an aspect weaver.

There are various options for representing business rules. It is possible to use: a) an application programming language, b) language meta-instructions, e.g., annotations, and an expression language, c) a functional programming language or d) a custom Domain-Specific Language (DSL)[9]. Although a) is very powerful and easy to learn, it lacks support of inspection, which makes us unable to transform rules into different target domains as is assumed by AODA. The second option is easy to inspect but not type safe. Also there has to be something to annotate, which might be an issue for cross-cutting rules. Functional languages such as Scala, Python or Groovy enable loosened syntax, which would make rules more convenient to read/write. DSL designed as a part of this language is powerful

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

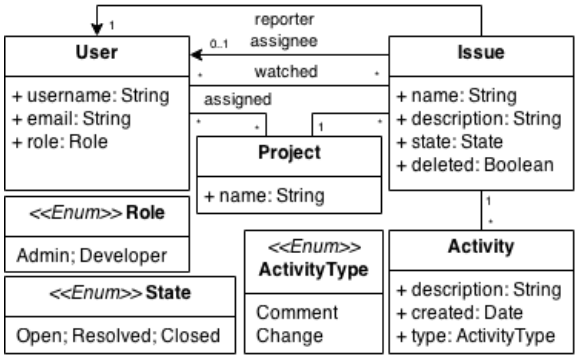

Fig. 1: Data model of the Issue Tracking application

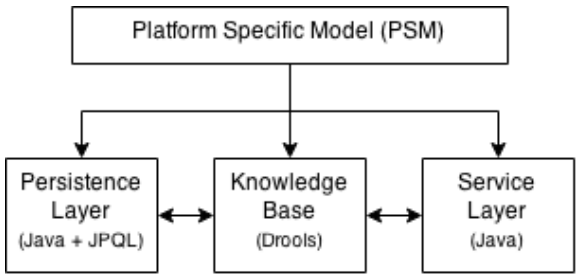

Fig. 2: Model transformation in Model-Driven Development

and without need of inspection, but it requires its compilation/interpretation and execution. That binds its evaluation to the specific environment that limits its reuse. Finally, it is possible to design a new custom DSL, but it is difficult, as it involves many challenges and design decisions[9].

For a convenient aspect description, AODA recommends using a custom tailored DSL. Although DSLs are a fundamental assumption for AODA deployment as they allow convenient and fast aspect development, they also introduce multiple challenges. First, the more languages a project uses, the more complicated it gets. The learning curve gets steep, and there is major overhead before the first deployment. On the other hand, the particular aspect development can be delegated to domain experts to ease it.

Often the best choice for a language to express business rules is to utilize some already existing implementation. There are a few convenient languages. One of formal languages often used for application modeling and Model-Driven Development[10] (MDD) is the Object Constraint Language[11] (OCL). It has strict syntax and is often used in research papers. Similar but more friendly to use is Drools DSL designed as a part of JBoss Drools framework[12] for rule-based systems. This Java-based, powerful language carries its own compiler and parser transforming source into meta-model. Although the intended purpose of this language differs from the scope of this paper, its expressiveness and supporting tools fits our use.

4. Business context

An EIS implements many business processes[4], e.g., the process of user registration or issue reporting. Each process consists of several activities mapped to UCs. A UC is decomposed into steps, e.g., user provides his registration data, system creates a user's account or system sends a notification email. These steps map into business operations, i.e., activities with some business value such as send an email or save a user. Every process is restricted by certain business rules applied also on its parts - UCs and business operations. Therefore it is possible to put down a set of rules, i.e., preconditions and post-conditions, for every UC and operation.

In an EIS, we differentiate four types of contexts: a) application, b) user, c) execution, and d) *business*. Application context represents the current application state, i.e., server IP address, state of global variables, etc. User context binds user-specific variables, i.e., origin, security roles and username. By execution context, we understand a whole context used for execution of a business operation that includes both application and user contexts and operation parameters.

Definition: A business context is set of parameterized preconditions and post-conditions with a business value.

As we state above, every business operation has its own set of preconditions and post-conditions. We capture these parameterized rules as a business context. Before execution of each operation, we verify its preconditions and later possibly apply its post-conditions. To do that, we create an instance of a business context and bind variables with values from an execution context. Then we can evaluate and apply rules.

Listing 1: Business context description in the OCL

```
ctx:  IssueService::listIssues(user: User): Set(Issue)
pre:  user <> undefined
post: result->forAll(
        issue|issue.assignee = user and issue.state = State::Open and not(issue.deleted)
      )
```

Listing 2: Business context in Java and JPQL

```
public Set<Issue> listIssues(User user) {
  if (user == null) throw new Exception("Not logged in");
  else return load("SELECT i FROM Issue i WHERE i.assignee = :user AND state = 'Open' AND NOT i.deleted", user );
}
```

Listing 3: Business context declared in Drools DSL

```
global User $user

rule "Pre: Issues assigned to the given user" when eval( $user != null ) then end

rule "Post: Issues assigned to the given user" when
 Issue( assignee == $user, state == State.Open, not(deleted) )
then end
```

Listing 4: Business context addressed from Java service

```
@PreConditions( "Pre: Issues assigned to the given user")
@PostConditions("Post: Issues assigned to the given user")
public Set<Issue> listIssues(@Param("$user") User user) { /* not invoked */ }
```

A business operation can be viewed at various levels of abstraction, determined by the number and detail of operation-specific rules. For example, at a higher level of abstraction, such a general precondition is that a user is logged in or that he is a developer. Similarly, a general post-condition is that entries of returned data are not flagged as deleted. Such rules might be highly repeated among operations. At a lower level of abstraction, rules are more specific for a particular operation, e.g., user is assigned to given project or given issue is closed. The specificity of rules determines specificity of a context.

To make a business context an efficient mechanism and allow rules reuse, it is crucial to support a hierarchy. Rules might be shared across multiple contexts so; to follow the *single place of declaration* principle, there has to be only one context declaring them. Such a context might be included by other contexts. Unfortunately, it carries typical disadvantages of multiple-inheritance, which we leave for future work.

We can capture a business context differently in various phases of the development life-cycle. For instance, consider again UC 2 from Section 2, *List user's assigned issues*, and the standard development approach[4]. It is possible to express the rules in plain text (Section 2) during application analysis. In the design phase, it is possible to use a formal language such as the OCL (Listing 1) and then manually transform it into an implementation platform, e.g., the Java language for precondition verification and JPQL[c] statement for data retrieval (Listing 2). As we can see, even this simple business context is repeated many times, and it is difficult to manually keep consistency among occurrences.

As AODA proposes, we can declare the rules in a single, independent knowledge base and then automatically transform them into the rest of a system. For example, we can use the OCL (Listing 1) or the Drools DSL representation (Listing 3) already in a model phase and then automatically transform it into implementation platform (Listing 4). In terms of MDD, we can perform vertical transformation from a platform-specific design model into an implementation platform without need of any change, as the knowledge base uses the same DSL. Then, as AODA

[c] Java Persistence Query Language, abstraction over SQL[6]

suggests, we perform only horizontal transformation to apply business context on an operation. The schema of the transformation is suggested in Fig. 2.

Deployment of independent business context description and then its automated transformation can significantly simplify project development and maintenance efforts because we no longer have to maintain the consistency of manually restated rules. Furthermore, we are not limited to the transformation of business context into only the persistence layer; such transformations apply many other places in the system (e.g., UI) as suggested in[3].

5. Preliminary results

Independent description of a business context enables us to develop a meta-model of business rules, which is easy to further transform. The Drools framework contains such a meta-model, although it does not smoothly fit the purpose; the original intention is different. Our framework builds above the Drools DSL, its compiler and a meta-model, and implements horizontal transformation from the knowledge base into a persistence layer (Fig. 2). The use of our framework comes out of the business context description and its addressing. Listing 3 provides an example of a business context declaration while Listing 4 shows how to address it in a persistence layer. As we can see, the body of the method in the example is not important, because it is replaced using AOP by the advice performing rule inspection, its transformation into a data loading query, and finally its execution. A similar approach is possible for verification of preconditions, as shown in[3].

We deployed the framework into the application introduced in Section 2. Although it is a small project, including our test cases we successfully transformed 25 different types of business rules from very simple conditions to parameterized restrictions over collections, multiple entities and with aggregation functions without any manual transformation or rules restatement. Furthermore, we joined our approach with the work of AODA and managed to develop an application automatically evaluating operation preconditions as well as transforming post-conditions into JPQL SELECT queries.

Despite this accomplishment, there remain several challenges. First of all, only a certain type of post-conditions is supposed to be transformed into a SELECT query: post-conditions restricting a returned value. The others just define method behavior, such as performed changes to a database. By now, we are unable to determine which part of a rule we should propagate into a database and what we can ignore. Furthermore, some rules are so simple that they might be evaluated already in memory to increase performance.

The second challenge is rule composition and context inheritance. As we stated previously, one of the most important qualities of business contexts is their inheritance. To deliver efficient rule reuse, the contexts are supposed to be connected into a graph. This assumption requires composition of rules over the same context into a single rule.

The third challenge lies in context chaining. Although every operation has a defined business context, we cannot use that operation in preconditions of another business context. For example a UI component is conditionally rendered based on the return value of an operation, i.e., the operation result affects preconditions of a UI component rendering. Right now, there is no solution for chaining contexts as these methods might take parameters. It is not possible to inject them from a current execution context and verify an assumption based on the returned value as there is no defined user's context to be used as one of parameter providers. For instance, the list of user's issues should be rendered only if the collection of them returned by a business operation is not empty.

6. Conclusion

Enterprise Information Systems facilitate data maintenance for many processes restricted by rules. Each business process consists of multiple business operations constrained by preconditions and post-conditions. Unfortunately, these conditions have to be considered in multiple places through a system, which usually leads to their manual restatement and difficult maintenance.

Authors of AODA propose the EIS decomposition approach to allow easy and efficient information description and maintenance through the definition of a single focal point and automated information transformation. We extend this idea by formalizing a business context encapsulating preconditions and post-conditions of each business operation. We then develop a framework applying AODA to deliver reuse and automated transformation of result restricting post-conditions into a persistence layer. Through this behavior, we deliver automated data filtering

without any need of manual rules restatement, which significantly simplifies application maintenance. Although there are several challenges left to deal with, such as context inheritance, our preliminary results suggest possibility to deploy the approach into a large production-level system to prove its feasibility. For future work, we leave several remaining challenges and major case study to demonstrate how much it simplifies project maintenance and speeds up development.

Acknowledgements

This research was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS14/198/OHK3/3T/13. We would like also to thank the Baylor University in Waco, Texas for the support during the research.

References

1. Fowler M., Patterns of Enterprise Application Architecture, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
2. Cerny T., Donahoo M. J., How to reduce costs of business logic maintenance, in: Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on, Vol. 1, IEEE, 2011, pp. 77–82.
3. Cemus K., 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.
4. Larman C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Pearson Education India, 2012.
5. Ambler S. W., Mapping objects to relational databases: What you need to know and why (2000).
6. DeMichiel L., JSR 317: Java Persistence 2.0 (2009).
7. Bauer C., King G., Hibernate in action, Manning Greenwich CT, 2005.
8. Bernard E., Peterson S., JSR 303: Bean validation (2009).
9. Mernik M., Heering J., Sloane A. M., When and how to develop domain-specific languages, ACM, New York, NY, USA, 2005, pp. 316–344.
10. Kleppe A. G., Warmer J. B., Bast W., MDA explained, the model driven architecture: Practice and promise, Addison-Wesley, 2003.
11. Demuth B., Hußmann H., Loecher S., OCL as a specification language for business rules in database applications, in: UML 2001-The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Springer, 2001, pp. 104–117.
12. Browne P., JBoss Drools business rules, Packt Publishing Ltd, 2009.