# Evaluation of Approaches to Business Rules Maintenance in Enterprise Information Systems

Karel Cemus, Tomas Cerny
Dept. of Computer Science
Czech Technical University
Technická 2, 166 27 Praha, Czech Republic
{cemuskar, tomas.cerny}@fel.cvut.cz

Michael J. Donahoo
Dept. of Computer Science
Baylor University
One Bear Place #97356, Waco, TX, USA
jeff_donahoo@baylor.edu

## ABSTRACT

Enterprise information systems maintain a significant number of business rules defining complex business processes and data constraints. Conventional frameworks distribute such rules over multiple layers, making consistent rule location discovery difficult. This lack of rule centrality results in complicated and even error-prone development and maintenance as future application changes may lead to inconsistencies.

In this paper, we consider these aspects of business rule definitions in various software design approaches from the software metrics perspective. We research several design approaches in a case study, compare their qualities regarding business rule definitions/maintenance, and demonstrate the difficulties with these approaches. The results of the study indicate that conventional approaches are insufficient in this area. Future design approaches that involve separation of concerns provide mechanisms to address the inefficiency.

## CCS Concepts

**•Software and its engineering → Software design engineering; Maintaining software;**

## Keywords

Enterprise information systems; aspect-oriented programming; design patterns; business rules; three-layered architecture; metrics; maintenance efforts

## 1. INTRODUCTION

Every Enterprise Information System (EIS) deals with various business processes and maintains corporate data. Consequently, it consists of a large stack of business rules determined by business processes and data constraints [17]. We consider all rules used in the system as business rules. Usually, they are derived from business processes, their constraints, and data flows. They affect everything from the User Interface (UI) through application logic to underlying persistence.



*RACS '15, October 09-12, 2015, Prague, Czech Republic*



DOI: http://dx.doi.org/10.1145/2811411.2811476

Many of the business rules repeat throughout the system, as they are considered in all layers – vertical repetition. Furthermore, many rules also repeat inside a single layer – horizontal repetition. For example, consider the rule that only administrators are authorized to execute certain methods. This rule repeats both horizontally and vertically.

There is also another type of repetition. First, we restate definitions of rules, a problem called semantics repetition (e.g., how to interpret that the user's name must be at least 4 characters long). Second, we repeat references to rules from related contexts [11], called reference repetition. For example, the same stack of rules may be considered in client-side validation, server-side controller validation, and validation of the input in the application layer. Contemporary technologies require repetition of the rules in all these places. In consequence, the business rule definition repeats many times in a code base. Maintenance of such code is very difficult, tedious, and error-prone as there is no single place for rule update [6].

Although there are some approaches to minimize business rules repetition, all are insufficient and fail to fully avoid repetition. They are usually verbose and introduce significant overhead. Furthermore, all of them consider only a subset of rule types. In this paper, we compare three approaches for dealing with business rules repetition to an purposely unoptimized implementation. We focus on identification of benefits of considered approaches and outline recommendations for future approaches to reduce maintenance efforts.

We structure the paper as follows. We discuss repetition types in Section 2. In Section 3, we present related concepts and metrics. Next, we discuss evaluated approaches in Section 4. To measure efficiency of the approaches, we conduct the case study in Section 5 and measure rule repetition and maintenance difficulty. Finally, we conclude in Section 6.

## 2. BACKGROUND

Business rules represent a fundamental part of every EIS. They consist of various types of conditions, from data constraints through access policy to operation preconditions. To illustrate their importance, consider a small Issue Tracking System with the domain model in Figure 1. The system maintains *Project*s consisting of *Issue*s. For each issue, there are submitted activities, such as comments and commits. Each *User* has one of three roles: user, developer, or administrator, and is assigned to zero or more projects and zero or more issues.

A common EIS uses the three-layered architecture [4, 5], distributing logic into three components:

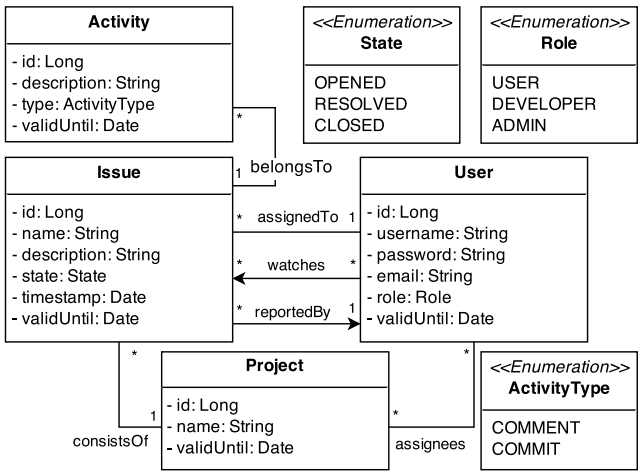


**Figure 1: Data model of the Issue Tracking system**

- persistence layer maintaining resources
- application layer implementing business logic
- presentation layer presenting data and operations

Business rules spread over all of these layers, each using rules for different purposes. The presentation layer partially validates the input as client-side form validation and conditionally rendered components, e.g., only some users are eligible to perform some operations or view the data. The application layer performs full input validation and verifies preconditions of executed operations so as not to violate business process flows. Finally, the persistence layer facilitates basic constraint validation and restricts returned collections of data according to given constraints, e.g., return all active projects led by John.

In this example, the persistence layer facilitates access to resources, e.g., in a database. It provides a storage-independent API to maintain resources with respect to data constraints. For example, a project's name can never be empty, a user must have a role, etc. Furthermore, it returns collections of instances satisfying given conditions, e.g., it returns issues assigned to the user or all opened issues belonging to the project. The application layer ensures data validation on its input plus it verifies preconditions of each operation. The modeled processes determine these preconditions. For example, a User instance must have unique username with length between 6 and 15 characters and a strong password. Furthermore, only unauthenticated users are permitted to sign up. Finally, the presentation layer delivers data to users and facilitates the access to business operations. Web services, input forms, and UI buttons must respect business rules. For example, an inserted issue must have a name longer than 5 characters and only the project developers can mark the issue as resolved, i.e., hit the button "resolve".

Unfortunately, a single rule may cross-cut both horizontally and vertically throughout the system. Consider the operation *Resolve an issue*. In the presentation layer, the "resolve" button is visible only when 1) *The user is a project developer*[1] and 2) *The issue state is open*. These preconditions are verified also in the application layer, in case the presentation layer fails to enforce the constraints (inadvertently or maliciously). This is vertical repetition. Consider the first rule. Such a precondition must be verified before execution of all operations and before rendering all UI buttons restricted to project developers. We call such repetition inside of the single layer, horizontal repetition.

Horizontal and vertical repetitions are consequences of business rules semantics and references repetition. Imagine the rule *The user is a project developer*. First, we must define its semantics, i.e., how to interpret it. In this case, we check the user's role and assignment to the project. Then we reference it everywhere we want to consider it, i.e., from every restricted UI button, every restricted business operation, etc. Common approaches often lead to repetition of both semantics and references throughout a code base. The cause of this lies in different technologies used for each layer[2] and lack of support of sharing business rules among them. Thus it usually results in high repetition of both kinds as there are no efficient ways to avoid it [15].

## 3. RELATED WORK

[14] identifies multiple, tangled UI concerns as causing error-prone and expensive maintenance. They suggest concern decomposition and automated integration of concerns by a build tool. As concerns, they address widgets, localization, navigation flow, and data validation, i.e., business rules. [12] details similar issues in the application layer, explores a few approaches to business logic maintenance in the Java EE platform, and concludes by suggesting using object-oriented design patterns to reduce maintenance efforts.

The authors in [22, 23] propose focusing on business rules and maintain them on various levels of abstraction so as to be 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. Either way, the authors in [24] recommend the *Don't Repeat Yourself* principle suggesting capturing every piece of knowledge just once in code.

A similar approach emphasizing the business rules as an important concern is suggested in [10]. The core idea lies in concerns decomposition and their runtime weaving through aspect-oriented programming. We provide more details of this approach in the following section. This approach, as well as the previous two, relies on power of domain-specific languages [7]. These languages, tailored for the particular domain, benefit from comprehensibility and the possibility to delegate the responsibility to domain experts. Unfortunately, they introduce significant overhead as they have to be designed, parsed, and compiled/interpreted. Furthermore, for their efficient use we must consider some development tools for syntax highlighting and code completion [20]. The efficiency of domain-specific modeling the authors also confirm in [13]. They claim the domain-specific languages can increase the overall productivity by 500-1000% just by using domain description and specific generators.

Concepts of business rules maintenance are well explored in context of rule-based systems [9]. They deal with a significant number of preconditions and action including their description and flow. Contemporary rule-based engines build on the top of domain-specific languages as they deliver an easily maintainable code base through support in development environments [3].

[1]A user with a role "developer" and assigned to the project.

[2]At least for the presentation layer and the rest of the system. Between these two components repetition is the most significant.

Model-driven development represents another approach to avoid code duplication [16]. It suggests maintaining various models on a few levels of abstraction and having them automatically transformed into more specific levels, which facilitates knowledge reuse. Unfortunately, this approach often has difficult maintenance as it usually supports only forward transformation. When we perform any change to the code, it is difficult to propagate it back into the model. Nevertheless, this approach does not conflict with the considered approaches, so they are usable together to minimize manual code duplication and information restatement.

Code cohesion [17] represents one of several code metrics often used for quality assurance. In [1] authors summarize existing cohesion approaches and proposes their own metrics. Their work focuses on early-stage model evaluation-based UML diagrams. Like many others, they consider class attributes, methods, and their relations. Such a metric is not suitable for business rules efficacy evaluation as they are captured inside methods. Code coupling [17], another metric, measures how many classes relates to each class [18]. Although this is valuable for overall system design, in context business rules there are barely any differences among approaches. The best fitting metric for this purpose is a cyclomatic method complexity [19]. It determines the average number of independent paths through methods. In each method of the application layer, i.e., in every business operation, we consider all business rules bound to it [11]. That means that for every rule, we are branching the code, which increases the complexity. By simplification of business rules definition and reuse we are able to lower 1) complexity and 2) repetition.

# 4. COMPARED APPROACHES

The paper considers four approaches for design and implementation of an EIS. These approaches significantly differ in the resulting qualities of the code and development and maintenance efforts. In this chapter, we present the approaches and highlight their fundamental ideas. In the next chapter, we compare them in a case study.

### *Unoptimized solution.*

For the reference solution, we consider a completely unoptimized solution. No frameworks or tools for business rules management or their representation are used and no efforts are made to optimize them. It captures them in a programming language directly in places where they must be considered. Unfortunately, this is a common practice in real-world applications. In consequence, the business rules semantics and definitions repeat both horizontally and vertically in the application and the presentation layer.

### *Java EE framework stack.*

Contemporary EISs often build on the top of the Java EE or .NET platforms, both with similar underlying principles. On the Java platform, the basic idea of dealing with business rules lies in *JSR 303: Bean Validation* [2]. This technique uses meta-programming (annotations) as declarative programming to annotate the model fields by additional constraints such as `@Email` for strings matching the email pattern or `@NotBlank` for non-empty strings. In addition, it allows annotating business operations to restrict the access by security policy (*JSR 205*) [21]. For example, `@RolesAllowed(DEVELOPER)` makes the operation available only to developers. In addition, a framework for the UI development called RichFaces[3] reads a subset of these annotations and automatically adds constraints into input fields in UI forms. For example, when we create an input box for an email field, it automatically adds its client-side constraints.

Unfortunately, this type of business rules representation suffers from serious limitations. These constraints have to be valid all the time, which fits only for a few rules. Usually, a significant number of constraints are defined only in the context a business operation and relates to user's context. Unfortunately, the Java EE stack is unable to deal with such contextual rules. It forces developers to tangle these rules into the code as we mention in the first case.

Although the concept is more generic, the implementation is very limited. To overcome that, for purpose of this case study, we implemented the Java EE extension enabling us to also annotate methods. It allows expressing constraints declaratively as in the model but restricts given method parameters and context. Method execution is triggered from the *Proxy* design pattern, which is natively supported by the Java EE platform. In conclusion, we may restrict both model and operations to express constraints. Consequently, in the application layer, we encapsulate the constraints semantics in a single place, but we repeat their declaration as there is no mechanism to reuse declared annotations among methods or classes. This extension does not affect the presentation layer; it remains limited to RichFaces behavior.

### *Object-based solution.*

The authors in [12] suggest respecting object-oriented design principles, such as encapsulation and polymorphism, to minimize repetition. They propose use of validator objects encapsulating rules as self-standing objects and their connection to code through visitor pattern and Java EE interceptors, i.e., proxy design pattern. The concept is similar to declarative programming from the previous paragraph, but this benefits from its easier extendibility. It is simple to implement custom validators or group up some common validators into single object. On the other hand, it is less declarative, and it cannot be inspected at runtime through reflection.

For example consider the `Issue` from the previous example. Its fields, such as `name` or `description`, are restricted by constraints about their length. In every place validating the Issue object, we invoke these validators to verify these constraints without actually duplicating validation code, although it restates the set and configuration of used validators. To avoid this, we create an `IssueValidator` encapsulating all validator invocations related to the Issue validation. Then we just reference the `IssueValidator` without any semantics restatement. We do the same for all business operations if necessary.

The downside of this approach lies in interpretation of business rules in multiple environments. While in the application layer we use them to validate the input, in the presentation layer we just translate them into a client-side language to perform client-side validation. In the persistence layer, we just use them for integrity constraint declaration. All these different interpretations cannot be covered

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

by simple validator objects. This approach works only in the application layer as we are not able to transform the validators into different technologies and the technology for UI usually does not support this kind of validators.

### *Aspect-oriented design.*

The authors in [10] consider business rules as aspects [15]. They perceive business rules as independent, cross-cutting concerns that are easily self-describable. Consequently, they suggest capturing them in a convenient, domain-specific language and encapsulating into business operation contexts (business contexts), i.e., a stack of rules applied on the particular operation [11]. Then we reference these contexts through declarative programming, e.g., annotations, and let rules injection be performed in runtime. This approach decouples context descriptions and methods to prevent definition repetition. The repetition of semantics is avoided through a single place of knowledge, i.e., the knowledge base with all rules and contexts. Empirical studies show that aspect-oriented designs tend to higher stability and lower impact of change requests [8].

As the rules are described in a domain-specific language, they are easily transformable into various environments and technologies. Consequently, it prevents the vertical restatement, as we just reference the context, and lets the transformation be done automatically. In consequence, the business rules restatement is significantly reduced as they are only in the context definitions. On the other hand, this approach suffers from significant overhead at the project beginning, as it uses complex weaving and transformation processes. It results in its difficult first deployment as it recognizes many aspects: business rules, UI layouts, widgets, etc. For all of these aspects and technologies, we must implement a proper, complex aspect weaver, and possibly transformation mechanism.

## 5. CASE STUDY

To compare the considered approaches, we conduct a small case study. We implement the Issue Tracking system from Section 2 and measure its business rules repetition, maintenance efforts, and SLOC[4]. For all four approaches, we use Java EE 6 platform with JavaServer Faces 2.1 for the UI. An aspect-oriented design approach also uses implementation of the concept provided by the Rule Guvnor project currently using JBoss Drools 6 as a domain-specific language for business rules description.

In the project, we identified 33 business operations restricted by 116 constraints including basic model constraints. 14 different views are implemented providing all 33 business operations. As shown in Table 1, the SLOC metric shows some differences among approaches. The aspect-oriented design has an undefined presentation layer SLOC. Currently, there is no implementation transforming the business rules into the presentation layer so we can neither implement nor measure it. The current implementation of the approach for the presentation layer, Aspect Faces[5], supports only layouts, widgets, and model structure but not business rules.

We can see that, based on the SLOC analysis, the Object-based solution is almost identical to unoptimized solution. On the other hand, the Java EE approach reduces the SLOC in the application layer by 14,5% in trade off to increased SLOC in the data model by 13,5%. The most significant difference is among the aspect-oriented design and the rest of the solutions. Regarding the model and the application layer, it is always the lowest SLOC value. Consequently, it reduces the SLOC of the data model and the application layer together by 17,5%, 14,6%, and 13,9% relative to the unoptimized, Java EE, and OBS versions, respectively. The absolute values here are small as we use only four model classes, but when we adjust this value to much bigger models, the SLOC reduction is significant. This results from business rules extraction into separate files. In the table, we see significant SLOC dedicated to business rules, but it is a consequence of using Drools language, which is not designed for this purpose. There is significant overhead and repetition inside of this file, which can be removed by use of custom and better-suited languages.

**Table 1: SLOC efficiency**

| Component | Unopt. | Java EE | OBS | AOD |
|---|---|---|---|---|
| Data Model[A] | 198 | 225 | 198 | 198 |
| Application Layer | 304 | 260 | 283 | 216 |
| Presentation Layer[B] | 207 | 224 | 207 | ? |
| Presentation Layer[C] | 794 | 771 | 794 | ? |
| Other | 0 | 89[D] | 209[E] | 273[F] |
| Reusable code[G] | 0 | 89[D] | 209[E] | 0 |

[A] Including annotations.
[B] Only in Java files, i.e., backing beans.
[C] Only in XML files, i.e., UI description.
[D] Custom annotations.
[E] Validators implementing business rules.
[F] Rules declaration in the domain-specific language.
[G] These SLOC are also included in upper lines.

Table 2 shows business rules restatement. The unoptimized solution is the worst case; there are no efforts devoted to avoiding duplication. The results show inconvenient constraint distribution throughout the all layers for all but the aspect-oriented design. Although the standard Java EE solution with our extension partially deals with the business rules, it still leaves many constraints uncovered and tangled into the code base, especially in UI. The Object-based approach slightly improves the unoptimized solution as we do not restate the constraint semantics, i.e., how to validate it, anymore. However, we repeat invocation of validators, i.e., constraint references, which is still tedious and error-prone as we might easily overlook some occurrences even when we build custom aggregated validators. Furthermore, we are unable to deploy this approach into the presentation layer, as the technology does not allow it. The aspect-oriented design neither duplicates the invocations nor restates the constraints definitions; it only addresses decoupled business contexts to define what rules apply. Unfortunately, in the business rules definition in the domain-specific language, there are a few repetitions, as the language does not provide rule-sharing mechanism among context. This issue can be overcome by the design of a custom language. In conclusion, AOD looks like the best solution, however, right now there is no implementation transforming business rules into the presentation layer, which falls back to the worst case.

[4]Source Lines of Code
[5]http://aspectfaces.com

**Table 2: Constraint restatement**

| Constraints | Unopt. | Java EE | OBS | AOD |
|---|---|---|---|---|
| In model[A] | 0 | 27 | 0 | 0 |
| In SQL[B] | 38 | 38 | 38 | 0 |
| In business ops.[C] | 78 | 3/56[D] | 62 | 0/33[E] |
| In UI (Java)[F] | 0 | 17 | 0 | 0/?[E] |
| In UI (XML)[G] | 183 | 160 | 183 | 0/?[E] |
| Other | 0 | 12[H] | 15[I] | 116/33[J] |

[A] Annotations above fields and referenced validators.
[B] Constraints in SQL to retrieve data from the database.
[C] Constraints in bodies of business operations.
[D] If conditions/annotations over methods and parameters
[E] Constraints/annotations or tags referencing the context.
[F] Only in Java files, i.e., backing beans.
[G] Only in XML files, i.e., UI description.
[H] Annotations representing constraints, reusable.
[I] Number of validators, reusable.
[J] Number of rules/contexts in domain-specific language.

Out of the scope of this paper, we conduct another research implementing AOD in the UI to deliver the first reference implementation considering also business rules. Nevertheless it is supposed to directly use no rules in the presentation layer and be able to deliver the functionality only through context references.

Considering maintenance efforts, the important criteria are 1) high business rules cohesion, 2) low or no business rule restatement, 3) the fewest possible places to edit, and 4) the simplest business methods. Looking into the results in Table 2, we see that the unoptimized solution is very difficult to maintain. There is high restatement as the code contains 299 constraints while the application domain declares only 116, i.e., increasing to 258%. Furthermore, they are tangled throughout multiple different technologies. This correlates with the method complexity measurement reported in Table 3. Every method contains a sequence of if-statements.

The Java EE solution slightly reduces restatement, but the number of technologies is increased by the Java annotations representing another place to consider. A strong aspect of this approach lies in delegation of the rules semantics to the validation processor scanning the annotations. Unfortunately it is quite difficult to implement a custom annotation, which may result in partial distribution of business rules among annotations and method bodies. This is supported in the Table 3, where we see there are some conditions captured inside of methods.

Although the restatement measurement of Object-based solution suggests no improvement, as we can see in Table 3, encapsulation of constraints into singleton validators significantly reduces its error-proneness. We move all if-statements into custom and easily testable validators objects. Basically, the idea is similar to the Java EE annotations, but this approach is less declarative and simpler to implement. To deliver better results, we can combine the Java EE approach with object-oriented principles and use validators where annotations would be too difficult.

The best solution seems to be the aspect-oriented design, as it reduces all restatement, complexity, and the number of technologies with constraint distribution. It tangles

**Table 3: McCabe Cyclomatic Complexity**

| Metric | Unopt. | Java EE | OBS | AOD |
|---|---|---|---|---|
| Average complexity[A] | 3.18 | 1.09 | 1 | 1 |
| Highest complexity[A] | 12 | 2 | 1 | 1 |
| Total complexity[A] | 105 | 36 | 33 | 33 |

[A] Method complexity in the application layer

throughout the code only names of business contexts, i.e., addresses to the knowledge base, decoupling sets of rules to apply from the places of application.

Finally, almost all these approaches introduce some overhead. In case of the extended Java EE stack, we must design custom annotations and particular executors, for the Object-based solution we design custom validators objects, and for AOD there is whole platform with the domain-specific language performing its inspection and automated transformation into all layers and technologies. However, although these components are more or less complex, they are the completely reusable among projects. In conclusion, although they possibly introduce significant overhead for a single project, when we use them with many projects, the overhead is low as we have stable custom platform.

# 6. CONCLUSION

Business rules represent a crucial part of an EIS; they define processes and data constraints. EIS developers face the challenge of business rules maintenance due to their amount, scope of systems and tangling throughout the whole system. In this paper, we explore four approaches to the EIS design and conduct a case study to compare them.

The results show the strengths of all approaches and suggest the direction to focus the efforts to design a novel, more efficient approach to business rules maintenance or which approaches should be supported by frameworks to deliver better results.

First, our investigation shows that limited implementation of Java EE stack makes it nearly as inefficient as the unoptimized solution is as there is high business rules semantics repetition throughout multiple technologies. Consequently, the maintenance efforts are high because we must consider many places to perform even the minor changes. For the purpose of this case study, we extended the implementation by annotations applying business rules to particular methods. With this enhancement, we show that it significantly reduces cyclomatic complexity of business methods in the application layer especially when we combine this approach with the simple validator objects for more complex rules. Furthermore, use of declarative programming makes the code more readable. Unfortunately, the lack of model and method inspection by UI frameworks forces us to restate the rules in the UI. Advanced UI framework considering the annotations significantly lowers maintenance efforts whilst it preserved already standardized approach.

However, as the best choice of considered approaches, we recommend AOD, avoiding the rules restatement and duplication at all. It also minimizes the number of technologies defining the rules semantics to one, the domain-specific language. Furthermore, use of business contexts decouples the sets of rules to apply from their places of applications. The

approach benefits from high cohesion of the rules and possibility to engage domain experts into development. Unfortunately, this novel approach is not yet fully implemented, the current domain-specific language suffers from significant overhead and semantics repetition. Furthermore, although the approach delivers high efficiency, its initial deployment and development of supporting tools bears very high initial overhead into the project, which disqualifies it from efficient use for small and medium-sized systems.

## 7. ACKNOWLEDGMENTS

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

## 8. REFERENCES

[1] J. Al Dallal and L. C. Briand. An object-oriented high-level design-based class cohesion metric. *Information and software technology*, 52(12):1346–1361, 2010.

[2] E. Bernard and S. Peterson. Jsr 303: Bean validation. *Bean Validation Expert Group, March*, 2009.

[3] P. Browne. *JBoss Drools Business Rules*. Packt Publishing Ltd, 2009.

[4] R. Chinnici and B. Shannon. JSR 316: Java™ Platform, Enterprise Edition (Java EE) Specification, v6, 2009.

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

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

[7] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[8] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. SantAnna, S. Soares, P. Borba, U. Kulesza, et al. On the impact of aspectual decompositions on design stability: An empirical study. In *ECOOP 2007–Object-Oriented Programming*, pages 176–200. Springer, 2007.

[9] F. Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9):921–932, 1985.

[10] Cemus, Karel and Cerny, Tomas. Aspect-Driven Design of Information Systems. In *SOFSEM 2014: Theory and Practice of Computer Science, LNCS 8327*, pages 174–186. Springer International Publishing Switzerland, 2014.

[11] Cemus, Karel and Cerny, Tomas and Donahoo, Michael J. Automated Business Rules Transformation into a Persistence Layer. *Procedia Computer Science Journal*, 2015.

[12] Cerny, Tomas and Donahoo, Michael J. How to reduce costs of business logic maintenance. In *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, volume 1, pages 77–82. IEEE, 2011.

[13] S. Kelly and J.-P. Tolvanen. *Domain-specific modeling: enabling full code generation*. Wiley. com, 2008.

[14] R. Kennard and R. Steele. Application of software mining to automatic user interface generation. *SoMeT*, 8:244–254, 2008.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. Springer, 1997.

[16] A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA explained, the model driven architecture: Practice and promise*. Addison-Wesley Professional, 2003.

[17] C. Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, 3/e*. Pearson Education India, 2005.

[18] M. Lorenz and J. Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., 1994.

[19] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, 1976.

[20] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[21] R. Mordani. Jsr 250: Common annotations for the java platform. *Sun Mircrosystems Inc*, 2009.

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

[23] B. Theodoulidis and A. Youdeowei. Business rules: Towards effective information systems development. *Business Information Systems–uncertain futures*, pages 313–321, 2000.

[24] D. Thomas and A. Hunt. The pragmatic programmer: From journeyman to master, 1999.