Enterprise Information Systems: Comparison of Aspect-driven and MVC-like Approaches

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

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

ABSTRACT

Design of an enterprise information system significantly impacts its development and maintenance efforts. The research shows that maintenance consumes about 65-75% of the software development time and about 40-60% of maintenance efforts are devoted to software understanding [2, 9]. This paper compares the Aspect-driven design approach as applied to the three-layered architecture to the MVC-like design approach implemented by many conventional web frameworks. While both approaches strive to avoid information restatement, they differ greatly in the underlying idea; thus, this work compares based on development efficacy and ease of maintenance. We highlight their differences and qualities, such as information cohesion, coupling and restatement, and discuss their maintenance efforts. We also investigate their ease of use, deployments, and provide recommendations on when to use each approach.

CCS Concepts

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

Keywords

Model-View-Controller; three-layered architecture; enterprise information systems; design approach comparison; aspect-oriented programming

1. INTRODUCTION

Rapidly growing complexity and requirements of Enterprise Information Systems (EISs) together with the need for fast delivery place enormous demands on developers' abilities, available approaches, techniques, tools, and frameworks. In an early phase of a project, architects make critical design decisions to determine the system architecture. This decision impacts a project's future success with respect to development and maintenance efforts. As we noted, maintenance consumes 65-75% of total software development time [9]. Furthermore, [2] indicates that 40-60% of maintenance efforts are devoted to understanding the software.

An EIS reflects various information regarding a system to fulfill requirements, e.g., a domain model, business rules, validation rules, and presentation. A design approach significantly impacts development and maintenance, because the resulting architecture influences information cohesion, coupling, and encapsulation [13]. Specifically, it determines system's learning curve and affects the difficulty in making changes.

An EIS maintains its data with respect to a large number of business constraints [3] through complex user interfaces [8]. Despite the existence of many approaches suggesting division of high-level responsibilities into layers or dedicated components, a certain group of design concerns is usually hard to separate from others. In [12] the authors describe *cross-cutting concerns* that tend to negatively impact component maintenance and reuse. These concerns are responsible for code tangling because standard object-oriented approaches fail to address them. This results in information restatement, low reuse, duplication [8, 11], and concern distribution throughout an application's implementation, leading to difficult, tedious, and error-prone maintenance [5]. An efficient design approach must provide mechanisms to separate all sorts of concerns, avoid cross-cuts, and provide the ability to centralize logic.

Contemporary systems typically use as their master architecture[1] either the *three-layered* or *MVC-like architecture* [3]. The three-layered architecture is well-known as Java EE and Microsoft .NET platforms use this as their master architecture. For MVC-like, we mean any architecture using an MVC pattern where the core element driving design and development is a rich domain model [3].. Many conventional web frameworks belong to this category, such as Nette for PHP[2], Django for Python[3], Rails for Ruby[4], and Play for Java/Scala[5]. In such systems, design radiates

RACS '15, October 09-12, 2015, Prague, Czech Republic
© 2015 ACM. ISBN 978-1-4503-3738-0/15/10...$15.00
DOI: http://dx.doi.org/10.1145/2811411.2811477

[1]By master architecture, we mean the core architectural element, which we describe. These architectures certainly have other, important elements and some of these ideas can be combined; nonetheless, a core element drives development and maintenance.
[2]http://nette.org/
[3]http://djangoproject.com/
[4]http://rubyonrails.org/
[5]http://playframework.com/

(even autogenerates) from the data model, made rich with semantics augmentation.

The three-layered architecture significantly suffers from the inability to address the cross-cutting concerns. Contrary, the MVC-like approach partially deals with them in its rich domain model. Furthermore, there are some other approaches reducing restatements and improving reuse [1, 9], but none of them presents a generic solution. The only exception is the Aspect-driven approach [7], which proposes improvement to the three-layered architecture to deal with various types of cross-cutting concerns without any restatement or duplication.

In this paper, we describe differences among the three-layered architecture, Aspect-driven approach, and MVC-like approach with emphasis on the ability to deal with cross-cuts. In Section 2, we provide explanation of related challenges in EISs in more detail; while in Section 3, we discuss the approaches and explain their underlying concepts. Specifically, we identify differences in areas of information cohesion, coupling, encapsulation, and restatement. For better illustration and understanding of the differences, Section 4 shows a small case study which gives a good opportunity to highlight strengths and weaknesses of all approaches. Section 5 provides an overview of related work, and we conclude the paper in Section 6.

2. BACKGROUND

An EIS usually optimizes business processes and maintains large amounts of data with a respect to a given business domain [3, 13]. The data management in an EIS usually involves a User Interface (UI) and/or a web service component [8, 11]. To satisfy all given requirements, a system has to cover various types of information, such as a domain model, presentation widgets, page layouts, business (domain) rules, and text localization. Unfortunately, most of these concerns apply to multiple locations throughout the both horizontal and vertical dimensions of the system [8].

Typical representatives of cross-cutting concerns in terms of Aspect-Oriented Programming (AOP) [12] are business rules. For example, they have to be considered in all layers of the three-layered architecture:

- input validation in the presentation layer
- business operations in the application layer
- constraint verification in the persistence layer

Furthermore, the architecture supports component fragmentation inside of each layer. For example, a system can have three presentation layer components: a UI, a Web Service and a Console. The business rules description entangles all components as it determines validation and access restrictions.

Business rules are not the only cross-cutting concern in an EIS. The challenge lies in their addressing and reuse. Captured information is considered in various places throughout the system. When we consider that the different technologies and possibly programming languages are used for the implementation of different parts of a system [8], concern reuse becomes even more difficult. However, there are some attempts to overcome this gap [1, 8]. Unfortunately, so far there are no architectures, design approaches, or frameworks providing a generic solution. Consequently, in the most cases, developers are unable to capture a concern on a single focal place and then reuse it anywhere it is needed [11]; thus it usually results in high information restatement and code duplication [8, 12]. Unaddressed, tangled concerns are responsible for low cohesion of components [11], which deteriorates readability. The maintenance of such code is highly error-prone and inefficient [5]. Unfortunately, this challenge and technological limit must be considered in the design phase to develop a feasible system architecture.

3. DESIGN APPROACHES

Despite the absence of a standard solution, there are approaches addressing the challenge and minimizing information restatement. The key lies in the custom, often domain-specific, concern representation allowing its convenient reuse. This section compares and discusses two design approaches. It emphasizes their fundamental ideas and presents deductions of their benefits and limitations in the context of cross-cutting concern encapsulation, cohesion, coupling, and reuse.

3.1 The three-layered architecture

The three-layered architecture [3] splits up the system into three different layers: persistence, application and presentation. System functionality is thereby distributed with each layer owning a subset of the responsibilities. The key concepts of the three-layered architecture involve the *Anemic Domain Model* (ADM) [4, 10] and a *Transaction Script* [3] design patterns. They both determine the structure and qualities of a system as they directly define responsibility and information distribution. The point of the ADM is to capture only data in a domain model with neither additional functionality nor dependencies. The behavior is pushed into the application layer, including business rules and logic.

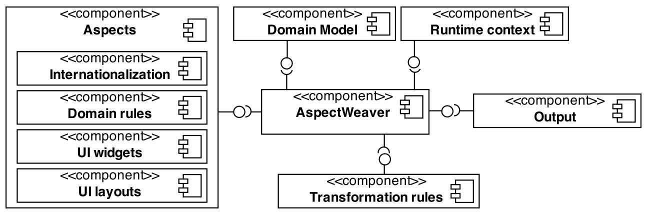
Migration of business rules from a domain model to the application layer causes inconsistencies in the rules as they must be restated in all operations (transactions) performed over the model [9]. Furthermore, there are other cross-cutting concerns, which apply to more than one location, e.g., security policy, presentation widgets, or localization, but in this architecture, there is no single location to capture them. The approach strictly limits capabilities of the model as it does not capture anything but data. Consequently, all these additions have to be mixed in upper layers, which tangles them throughout the code at the cost of low information encapsulation and high restatement [11].

The motivation for the use of the transaction script pattern lies in business operations (transactions), which reflect user's intentions. Each operation defines its own assumptions (preconditions) about the application state and a user's context. In this paper, we refer to any preconditions as *business rules* and the operation-specific set of preconditions as a *business context*. For each business context, we are able to put down assumptions and attach them to an operation (e.g., in a use case scenario); however there is no easy way to reuse a context among multiple operations [9].

3.2 MVC-like Approach

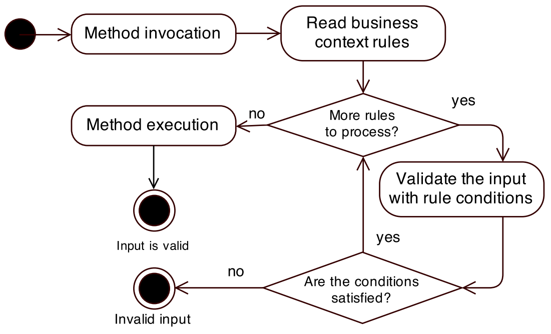
The other approach we call *MVC-like* because even though many conventional web frameworks use it, as we mention in Section 1, there exist many variations; e.g., Model-View-Presenter, Model-Template-View, etc. Nevertheless the fundamental idea remains same. The reason we choose this approach is its efficient development, maintenance, and ease of use for small and medium sized EISs.

The key idea behind the MVC-like approach is a *Rich Domain Model* (RDM) [4, 10]. Contrary to the Aspect-driven



<<component>> Aspects
<<component>> Internationalization
<<component>> Domain rules
<<component>> UI widgets
<<component>> UI layouts
<<component>> Domain Model
<<component>> Runtime context
<<component>> AspectWeaver
<<component>> Output
<<component>> Transformation rules

Figure 1: Aspect-driven decomposition of a system



Method invocation
Read business context rules
More rules to process?
no
yes
Method execution
Validate the input with rule conditions
Input is valid
yes
no
Are the conditions satisfied?
Invalid input

Figure 2: Input validation with an aspect interception

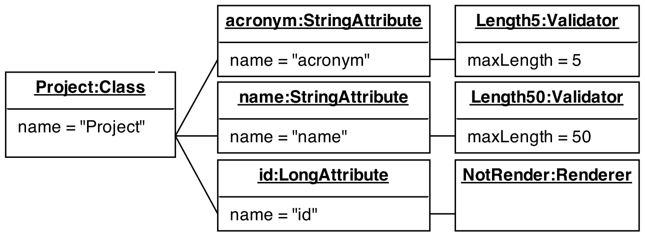


Figure 3: A typical MVC-like domain model

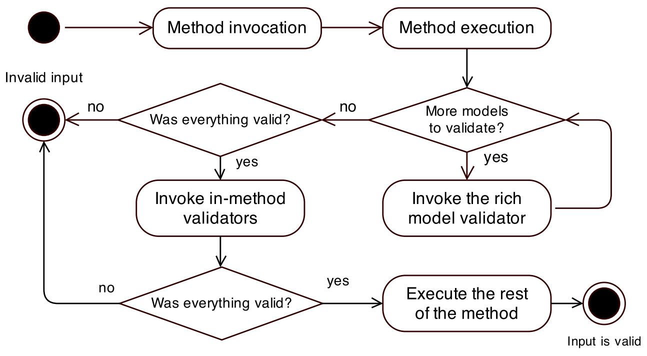


Figure 4: Input validation with operation validators

approach, it suggests capturing all concerns in a model or in its dependencies through highly-decorated classes and fields; e.g., domain logic, database access, or field presentation widgets. Basically, each class carries information on how to persist, validate, and render it.

Many fields and classes share a lot of configuration and mechanisms. The approach uses inheritance and complex field data types to avoid code duplication and information restatement. This means that for class validation, persistence, or presentation, there usually exist super-classes already doing that; for fields, there are predefined complex data types. For example, there is an EmailAttribute, a StringAttribute or a NumberAttribute. Each class carries its own default validation rules, database constraints, presentation renderer, etc. An example of such a data model is shown in Figure 3. We can see the class does not consist of basic data types, such as a String or an Integer, but of complex types with additional behavior such as validation or UI presentation.

Let us recall two things mentioned earlier in this paper: a) the MVC-like approach emphasizes a model and tries to capture there as much information as possible and b) cross-cutting concerns tangle through some other concerns, usually the model. Considering these two facts together, we reach the conclusion that the MVC-like approach tries to capture cross-cutting concerns in the model, more specifically in classes and fields cross-cut by those concerns. For example, attribute-specific widget renderers represent the presentation concern, and business rules are partially verified by model validators.

Unfortunately, there is a boundary because some concerns may cross-cut multiple classes or business operations. This approach is not able to effectively deal with these cross-cuts and falls back to information restatement and its object-oriented improvements described in [9]. For example consider business operation specific preconditions, which apply to multiple but not all operations. An invocation of validation of such business operation preconditions looks as is shown in Figure 4. The figure shows regular invocation of standard argument-specific validators at first followed by the invocation of operation-specific rules defined in the body of the operation.

Let us summarize the qualities of this approach. There is a strong emphasis on a domain model, and it aims to capture as much information as possible. This results with high encapsulation of most information because they are captured very close to their source. On the other hand, all various concerns are tangled into the model, which results in low cohesion and high coupling. Unfortunately, this approach is strongly class and attribute oriented, which limits the support of cross-cutting concerns. When the concern cross-cuts something else, the approach tangles it throughout a system instead of its clear separation.

3.3 The Aspect-driven Approach

The idea of the AOP-driven approach lies in decomposition of cross-cutting concerns and their isolated description (aspects) in convenient, domain-specific languages (DSL). These descriptions are then integrated (woven) into the system at runtime. Depending on transformation rules, a weaver produces multiple, distinct output components, such as a modified persistent layer or a modified application layer. Figure 1 illustrates an example of a decomposed system.

In consequence, we design and develop a system without any cross-cutting concerns involvement. For example, let us imagine input validation in an application layer. The conventional three-layered architecture tangles business rules into an operation, but the Aspect-driven approach [7] designs an operation itself and business rules captures in the aspect. Then, it addresses the rules from the operation through a business context. Figure 2 shows an example of such a method invocation with an aspect interception. The approach results in less information restatement, because the aspect weaver performs information reuses automatically at runtime. The isolated description of concerns in a single place highly increases cohesion and reduces coupling. On the other hand, concerns such as business rules are described separately from a domain model, thus their encapsulation is reduced.

3.4 Approach Comparison

This section sums up qualities of both approaches together, and concludes the results, which we demonstrate in the case study in Section 4.

3.4.1 *Coupling, cohesion and encapsulation*

These are crucial qualities [13] as they strongly determine the efficiency of development. While high encapsulation is needed when adding new features, low coupling and high cohesion strongly simplify system modification. Otherwise these operations are highly tedious and error-prone. The Aspect-driven approach relies on description of each concern isolated from the rest of the system. This results in high information cohesion and low coupling; the information encapsulation is low because each domain object is described in multiple different places, always in a different point of view (its aspect). Such qualities determine easy system modification as it is needed to change only a single place, but to add a new domain attribute or a class is complicated since it requires changes in many places.

In contrast, the MVC-like approach prefers all information captured in a single place where concerns cross-cut a domain model, even though they get tangled together. This results in high information encapsulation but also high coupling and low cohesion. The situation here is reversed: to add a new attribute or a class is easy, as we have to define it only in a single place. Modifications are more complex because the concerns are not isolated but tangled through a domain model. Difficulties come when we consider that we are unable to capture all concerns in the model, so we have to tangle them through the rest of the application. It badly impacts the application design as the model cannot be considered as the single place where the concerns are captured. These concerns then cross-cut the rest of the system instead. While the coupling remains high, the encapsulation suffers and decreases due to higher information restatement.

3.4.2 *Usage Efforts*

In order to use the Aspect-driven approach, it is necessary to apply multiple supportive tools and frameworks. As the key idea relies on multiple domain-specific languages, it needs compilers/interprets, and complex aspect weaver to describe and integrate all captured concerns together. Furthermore, there are various transformation rules to produce the whole application. Such overhead and technological dependency indicate a significant barrier impeding the smooth concept adaptation. Another issue is the steep learning curve for development. The more languages in use, the more complicated the development becomes. The strong advantage of DSLs lies in the possibility to delegate the work to domain experts with limited programming knowledge [14].

The MVC-like approach relies only on the domain model that is the central source of information for most of systems concerns. Such an approach does not require complex tools or frameworks since it uses a single (programming) language. Also the learning curve is much less. On the other hand, every change in the system must be done in the domain model, which can be done only by developers. Then it requires recompilation and new deployment of an application.

4. CASE STUDY

In order to show the advantages and disadvantages of considered approaches, we conduct a case study demonstrating the efficacy of both approaches and the pure three-layered architecture. There are significant differences between them in design so the study focuses on system modeling rather than implementation as implementation differences are consequences of design. We use platform-specific models [13] because there the differences are most visible.

We model an issue tracker supporting multiple projects composed of issues. Each issue has a set of work logs and comments. The system also maintains a catalog of users where each user might have a different role in each project. The expected system behavior is as usual. Users browse, report, edit, or resolve issues based on their role with a particular project, and optionally they make comments and work logs. The administrators also maintain catalogs of users and projects. To complete the requirements, we consider the list of business rules, which apply as model constraints or business operation preconditions. In total we identify 92 business rules, 63 as model constraints, and 29 behavioral constraints, which stand for operation preconditions. Considering nonfunctional requirements, the application has three different layouts (desktops, tablets, cell phones), supports two languages (English and French), and provides two different sets of UI widgets (one for touch devices and one for the rest).

The model with the Aspect-driven approach uses the Java Enterprise Edition (Java EE) platform for its future implementation. To define business rules, we use JBoss Drools Expert[6], and for the UI, we use the Aspect Faces[7] framework implementing the concept proposed in [8]. We use our own implementation of the aspect weaving core based on AspectJ[8] as there is no other public implementation. Similar to Java EE, the Aspect-driven approach uses the three-layered architecture and an anemic domain model so the model shown in Figure 5 is same for the pure three-layered architecture and the Aspect-driven design. Furthermore, the overall system architecture follows exactly the same concept because all the enhancements (business rules, localization, layouts, widgets) are described separately as standalone aspects, and they are weaved in at runtime. The overall design efficiency is captured in Table 1.

For the target platform of the MVC-like approach, we choose the Django 1.7 web framework built on the Python language. While the basic domain model structure remains the same, i.e., classes are identical, there are differences reflecting the rich domain model character of the approach. The detailed model diagram is very complicated because fields of classes are complex data types and each of them has a relation to its own widgets and validators. We publish there only a small part. The UML object diagram of the most simple class, the Comment, is shown in Figure 6. The

[6]http://www.drools.org/
[7]http://www.aspectfaces.com/
[8]http://www.eclipse.org/aspectj/

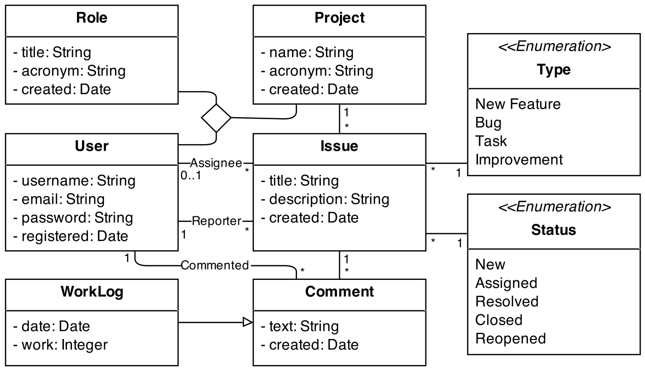


Figure 5: The model for the Aspect-driven design

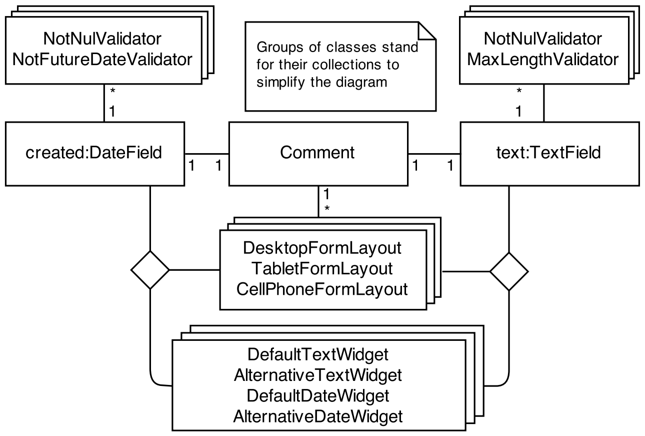


Figure 6: Object model of MVC-like Comment class

rest of the diagram would reflect the domain structure in Figure 5 and the object model in Figure 6.

The MVC-like approach considers almost all requirements on the model level, including UI widgets and business rules, though there is no single place to capture them. It results in their distribution into various parameterized validators, which makes rules and validators configuration tangled through the entire model. On the other hand, there are no rules captured outside the model, which is better than what the pure three-layered architecture provides. The other minor disadvantage of this approach lies in the validation invocation. As we already suggested in Figure 4, this approach invokes the validators manually in each domain operation; thus there is minor code duplication as this dependency cannot be omitted. Truth be told, a similar dependency exists in the Aspect-driven approach, but there are references to business contexts, not to the specific validators. They are attached automatically at runtime based on context descriptions.

Both approaches handle UI production in a different way but with similar results. While the Aspect Faces weaves all aspects at runtime to produce desired a UI, the MVC-like approach uses form templates and widget renderers to reach a similar effect. The significant difference is met when we look at business rules in the UI. While the Aspect-driven approach can inspect declared rules and transform them into the proper UI technology, the MVC-like approach cannot do it. Nevertheless, neither of these approaches significantly restates information, which would be apparent in the conventional three-layered architecture.

The pure three-layered architecture does not provide any mechanism to reuse business rules, widgets or layouts so it results in high information restatement. Although there is a technique [1] to reuse most of the model constraints, it does not cover all business rules, especially not preconditions.

As you can see in Table 1, all compared approaches deliver the application with the similar qualities. While the Aspect-driven approach has no information restatement, it is much more difficult to use and establish it. Such investment might be worth for huge expensive projects, but for small and medium sized, EISs might be too complex. The MVC-like approach is based on good object-oriented thinking and capturing almost everything in the domain model (in this case study, it is everything). Although there are situations where this approach leads to higher information restatement, even outside of the model, the total quantity of such business rules constitutes a small share, specifically, only those rules that affect multiple classes at once. On the other hand despite this limitation, the overall approach is very efficient and easy to deploy; thus it can be suggested for small and medium sized EISs. For large systems with many classes and business operations, it would result in a complex model that is hard to maintain as there is no single place where to capture the information.

5. RELATED WORK

We inspect the approaches from their ability to encapsulate the information and preserve high cohesion and low coupling to deliver the easiest possible maintenance. These challenges are already known for many years, and authors discuss them e.g., in [15] where state that about 65-75% of total project life-time is consumed by maintenance.

The difficulty of business logic description and maintenance is discussed in [9]. It states that efficient isolation and application of business logic is very complicated using pure object-oriented techniques, even when we restrict our focus only to the application layer. It proposes a new concept to transform business logic into a presentation layer in [8]. The method relies on a domain model decorated by additional information, such as simple business rules as is introduced in [1]. It transforms a decorated model according to a given dynamic user's context into a user interface at runtime. It shows significant source code reduction in the UI (up to 32%) and easy information/template maintenance. Later, the concept has been generalized into one of the compared approaches and introduced in [7].

The focus on maintainable, object-oriented software belongs among best-practices, which are covered by architectural and design patterns explained in [3, 6]. This technique is driven by the idea that the convenient code structure may significantly improve code maintainability. Such an approach is tightly bound to testability, reliability, and extendability. The premise is correct for many kinds of use cases, but there are cases where it fails. Furthermore, design patterns are meant for lower level of abstraction as they represent a component rather than a system architecture. The other architectural patterns would better fit this paper purpose but they are not used much in the domain of EISs.

Cross-cutting concerns impact design and code because they represent functionality and features, which affect multiple classes, components, and layers at once. Among these

Table 1: The overall design efficiency

Criterion / Approach	Three-layered[A]	MVC-like	Aspect-driven
Model constraints	63		
Op. preconditions	29		
Restated rules	29	20[B,C]	0
Rules locations	83	38[B]	1
Model classes	8	8	8
Business services	5	unsupported	5
Validators	0	11	0[D]
UI widgets	unsupported	12	12
UI layouts	unsupported	3	3
UI views	30	10	10
Client-side valid.	limited	supported[E]	supported

[A] Standard, not improved, without aspects.
[B] Only in the domain model.
[C] Restated only references to validators.
[D] It uses specific rules. There are more rules than parametrized MVC-like validators, but they do not restate their configuration.
[E] Requires mapping of validators to client-side technology.

belong, for example, logging of service results[9] because a logger must be attached into each business method to perform the action. Common object-oriented techniques fail here because they are unable to clearly assign the responsibility to a single object [12]. *Aspect-Oriented Programming* is an alternative concept, where the fundamental idea lies in isolated description of all cross-cutting concerns and then their automated integration into the rest of an application. This technique lies in the core of the Aspect-driven approach to deal with cross-cutting concerns. The value of AOP in the three-layered architecture is also shown by the Java EE Spring framework[10], which enhances the Java EE platform to support requirements as mentioned above.

Seemingly, the *Model-driven development (MDD)* [16] is another major approach to consider. This concept uses application modeling in an independent, possibly graphical, language and semi-automated transformations into source code in a target platform. The approach name indicates that this concept is on a different layer of abstraction. While this paper focuses on approaches to design a system architecture, the MDD is aimed to simplify the development process with respect to a design approach and a target architecture. Thus it can be used with both discussed approaches as long as we are able to transform the model into the chosen platform.

6. CONCLUSION

Development of enterprise information systems faces the pressure of fast-growing complexity, scope, and requirements. Selection of an approach is important as it significantly impacts system development and maintenance efforts. In this paper, we discuss two major approaches suggesting how to design a system architecture and distribute responsibilities to deliver a system with minimum efforts. Although we show that both approaches deliver equal results without any significant, unnecessary information restatement, their design qualities differ.

The Aspect-driven approach improves flaws of the three-layered architecture often used by large robust systems. It brings significant difficulties to establish it, get insight into an existing project, and learn how to use it as it suggests to use multiple, domain-specific languages. However, the solution is generic so it is able to capture all kinds of rules and cross-cutting concerns. It preserves high cohesion, low coupling, and no restatement at the cost of low encapsulation, because the concerns are described separately from the rest of a system. Furthermore, the vast use of DSLs brings the possibility to use domain experts to better distribute the work in a development team.

The MVC-like approach uses strict object-oriented thinking, leading to a rich domain model covering everything, from a domain structure through business rules to UI widgets in a single programming language. The only limitation of this approach lies in rules and concerns cross-cutting more than one class at once. Such cases lead to vast information restatement. Otherwise, information is captured with high encapsulation, although it is tangled through the model, which highly lowers its cohesion and increases coupling. Consequently, maintenance gets more complicated and expensive as there is no single place to update.

Such characteristics of the MVC-like approach make it easy to learn and establish and together with a compact domain model. It represents the best solution for small and medium sized systems. On the contrary, the generality and robustness of the three-layered architecture and its improvement makes the Aspect-driven approach the best fit for large and complex systems because the resulting architecture significantly decreases the maintenance and development efforts through its improved inner structure and organization.

7. ACKNOWLEDGEMENTS

We would like to thank the Baylor University in Waco, Texas for the support during the research, and especially to professor Michael J. Donahoo, Ph.D. for his supervision to this paper. This research was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS14/198/OHK3/3T/13.

8. REFERENCES

[9]In the application layer of the three-layered architecture.
[10]http://projects.spring.io/spring-framework/

[1] E. Bernard. JSR 303: Bean validation, November 2009.

[2] P. Bourque and R. Dupuis. Guide to the software engineering body of knowledge 2004 version. *Guide to the Software Engineering Body of Knowledge, 2004. SWEBOK*, 2004.

[3] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] M. Fowler. Anemic domain model. Availible at http://martinfowler.com/bliki/AnemicDomainModel.html, November 2003.

[5] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

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

[8] Cerny, Tomas and Cemus, Karel and Donahoo, Michael J and Song, Eunjee. Aspect-driven, Data-reflective and Context-aware User Interfaces Design. *Applied Computing Review*, 2013.

[9] 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*, June 2011.

[10] C. A. Iglesias, J. I. Fernández-Villamor, D. Del Pozo, L. Garulli, and B. García. *Combining domain-driven design and mashups for service development*. Springer, 2011.

[11] R. Kennard, E. Edmonds, and J. Leaney. Separation anxiety: stresses of developing a modern day separable user interface. In *Human System Interactions, 2009. HSI'09. 2nd Conference on*, pages 228–235. IEEE, 2009.

[12] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. Aspect-oriented programming. In *In ECOOP'97-Object-Oriented Programming, 11th European Conference*, volume 1241, pages 220–242. Springer, June 1997.

[13] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[14] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

[15] S. Muthanna, K. Ponnambalam, K. Kontogiannis, and B. Stacey. A maintainability model for industrial software systems using design level metrics. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 248–, Washington, DC, USA, 2000. IEEE Computer Society.

[16] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE software*, 20(5):42–45, 2003.