Aspect, Rich and Anemic Domain models in Enterprise Information Systems

Karel Cemus[1], Tomas Cerny[1], Lubos Matl[1], and Michael J. Donahoo[2]

[1] Dept. of Comp. Science, Czech Technical University, Technická 2, 166 27 Praha, CZ
{cemuskar, tomas.cerny, matllubo}@fel.cvut.cz,
[2] Dept. of Comp. Sc., Baylor University, One Bear Place #97356, Waco, TX, USA
jeff_donahoo@baylor.edu

Abstract. The research shows that maintenance of enterprise information systems consumes about 65-75% of the software development time and about 40-60% of maintenance efforts are devoted to software understanding. This paper compares the Anemic Domain Model used by the three-layered architecture followed by Java EE and .NET platforms and the Rich Domain Model often deployed into many conventional MVC-like web frameworks to a novel Aspect Domain Model followed by the Aspect-driven design. While all these models strive to avoid information restatement, they greatly differ in the underlying idea and resulting efficiency. This research compares considered models based on development efficacy, maintainability and their impact on the rest of the system. We evaluate qualities such as information cohesion, coupling and restatement, and discuss related maintenance efforts of the novel approach in the context of existing approaches.

Keywords: Model-View-Controller, three-layered architecture, enterprise information systems, design approach comparison, aspect-oriented programming, rich domain model, anemic domain model

1 Introduction

Enterprise Information Systems (EISs) reflect various information to fulfill rapidly growing requirements, e.g., a domain model, selected business rules, validation rules, and presentation widgets. A design approach significantly impacts development and maintenance, because the resulting architecture influences information cohesion, coupling, and encapsulation [13]. Specifically, it determines readability, system's learning curve and affects the difficulty in making changes. In an early phase of a project, architects make critical design decisions to determine system architecture. This decision impacts project's future success as consequence of development and maintenance efforts.

Despite the existence of many approaches suggesting division of high-level responsibilities into layers or dedicated components, *cross-cutting concerns* are usually hard to separate from others. They tend to negatively impact component maintenance and reuse because standard object-oriented approaches fail to selectively address them [12]. In consequence, this results in emerging information restatement, low reuse, high duplication [7, 10], and manual concern distribution throughout the whole system. Such design leads to difficult, tedious, and error-prone maintenance [2]. Therefore, an efficient design approach must provide mechanisms to separate all sorts of concerns, avoid cross-cuts, and provide the ability to centralize concerns to a single location.

Contemporary systems usually follow either the *three-layered architecture* with a *Anemic Domain Model* (ADM) or the *MVC-like architecture* with a *Rich Domain Model* (RDM) [3] with their *primary* system design. While the former is well-known[3], by the other we mean any *primary* system architecture derived from the *Model-View-Controller (MVC)* architectural pattern with RDM. For illustration, many conventional web frameworks belong to this category, such as Nette for PHP, Django for Python, Rails for Ruby, and Play for Java/Scala[4].

Unfortunately, the three-layered architecture with ADM often fails to address cross-cutting concerns. However, there exists its aspect-oriented extension, i.e., *Aspect-driven design approach* [6] with *Aspect Domain Model* (AsDM), tailored to deal with cross-cutting concerns. In this paper, we evaluate qualities and the impact of AsDM by its comparison to ADM and RDM used by common approaches. In Section 2, we discuss challenges in EIS design; while in Section 3, we elaborate the models and their underlying concepts. For better illustration of the differences, Section 4 shows a small case study highlighting qualities of all models and discusses the results. Section 5 provides an overview of related work, and we conclude the paper in Section 6.

2 Cross-cutting concerns

An EIS 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 often involves a User Interface (UI) and/or a web service component [7, 10]. To satisfy all given requirements, a system covers 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 [7].

Typical representatives of cross-cutting concerns in terms of Aspect-Oriented Programming (AOP) [12] are business rules. For example, we consider them 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

[3] Java EE and Microsoft .NET platforms build on it.
[4] http://{nette.org, djangoproject.com, rubyonrails.org, playframework.com}

Furthermore, the three-layered architecture supports component fragmentation inside of each layer, i.e., a system can have three presentation layer components: a UI, a Web Service and a Console. The business rules tangle throughout all components as they determine validation and access restrictions.

Besides business rules, there are other cross-cutting concerns [10]. The challenge lies in their addressing and reuse as they are considered in various places throughout the system. Furthermore, when we consider different technologies and possibly programming languages used for the implementation of different parts of a system [7], concern reuse becomes even more difficult. There are some attempts to overcome this gap [1, 7], although there are no architectures, design approaches, or frameworks providing a generic mechanism. Consequently, in most cases, developers are unable to capture a concern in a single focal point and then reuse it anywhere it is needed [10]; thus it usually results in high information restatement and code duplication [7, 12]. Unaddressed, tangled concerns are responsible for low cohesion of components [10], which deteriorates readability. The maintenance of such code is highly error-prone and inefficient [2].

3 Design Approaches and Domain Models

Despite the absence of a standard solution, there are approaches addressing the challenge and minimizing information restatement. This section summarizes their fundamental ideas and discusses their benefits and limitations in the context of cross-cutting concern encapsulation, cohesion, coupling, and reuse.

3.1 Anemic Domain Model in the Three-layered architecture

The three-layered architecture [3] splits up the application 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 ADM [4, 9] and the *Transaction Script* [3] design patterns. They both determine the structure and qualities of a system as they directly define responsibilities and information distribution. ADM captures only data in a domain model with neither additional functionality nor dependencies. It is pushed into upper layers such as application and presentation, including business rules and logic.

Migration of business rules from ADM to the upper layers causes inconsistencies in the rules because they must be restated in all operations (transactions) performed over the model [8]. Furthermore, there are other cross-cutting concerns, which apply to more than one location, e.g., security policy, presentation widgets, or localization, but with this model, there is no single location to capture them. The approach strictly limits capabilities of the model as it does not capture anything but data. In consequence, all these additions have to be mixed in upper layers, which tangles them through code at the cost of low information encapsulation and high restatement [10].

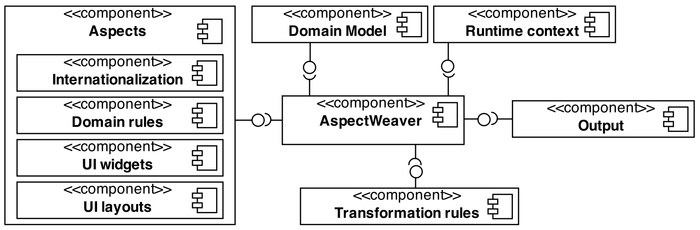


Fig. 1: Aspect-driven decomposition of a system

3.2 Aspect Domain Model

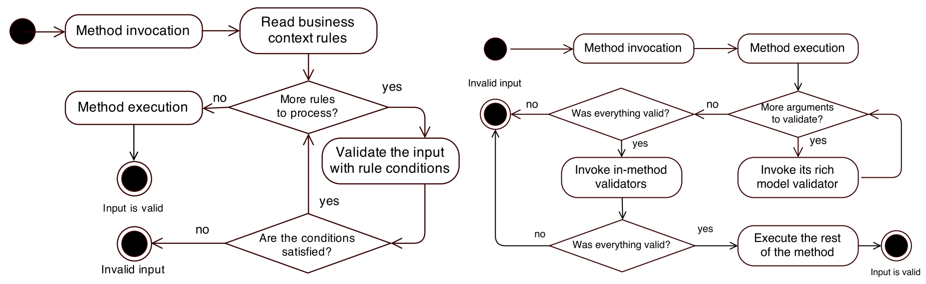
The motivation for use of the transaction script pattern within the three-layered architecture lies in business operations (transactions), which reflect user's intentions. Each operation defines its own assumptions (preconditions) about the application state and a user's context. In this paper, we refer these 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 [8]. Similar issues apply also to UI development and maintenance as it also strongly depends on business rules [11].

The Aspect-oriented extension [6] of the three-layered architecture resolves its inability to deal with the cross-cutting concerns. This approach decomposes cross-cutting concerns into their isolated descriptions (aspects) expressed in convenient, e.g., domain-specific, languages (DSL) and aggregates them in a single place, the Aspect Domain Model (AsDM). It provides a single focal point and a single place to update. For example, it decomposes business rules into independent business contexts. Then it integrates (weaves) those descriptions 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, imagine input validation in the application layer. The conventional three-layered architecture tangles business rules into operations, but the Aspect-driven approach designs operations themselves and business rules captures in the AsDM. Then, it addresses the considered rules from operations through a business context. Figure 2a shows an example of such a method invocation with an aspect interception.

3.3 Rich Domain Model

RDM [4, 9], contrary to ADM, 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 itself.


Method invocation
Read business context rules
More rules to process?
no
yes
Method execution
Validate the input with rule conditions
yes
Input is valid
no
Are the conditions satisfied?
Invalid input
Method invocation
Method execution
Invalid input
no
Was everything valid?
no
More arguments to validate?
yes
yes
Invoke in-method validators
Invoke its rich model validator
no
Was everything valid?
yes
Execute the rest of the method
Input is valid

(a) AsDM: aspect interception (b) RDM: model validators

Fig. 2: Input validation execution in different models.

Many fields and classes share a lot of configuration and mechanisms. The model 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.

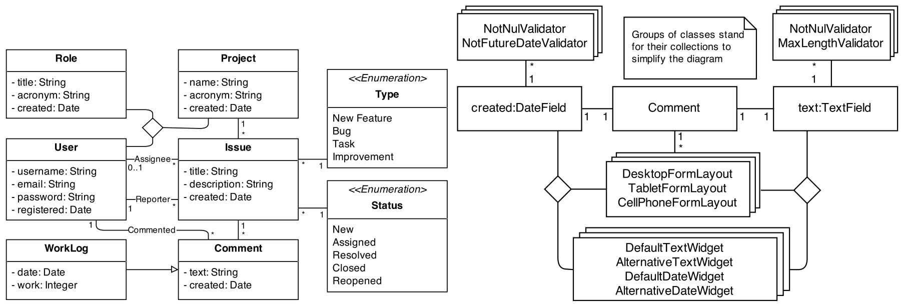
In consequence, RDM deals with cross-cutting concerns through rich data types. It captures them already in model fields, validators, and renderers, and tries to avoid their entangling into the rest of the application. However, there are some context-specific concerns and concerns cross-cutting multiple operations at once. As this model does not provide any mechanism to efficiently express and reuse them, it falls back to their repetition and object-oriented improvements described in [8]. For example, consider business operation-specific preconditions, which apply to multiple but not all operations of the class. As such, including those preconditions into the class validator is not an option and we are forced to duplicate rules in the operations themselves. Invocation and validation of such a business operation is shown in Figure 2b. It shows regular invocation of standard object-specific validators at first followed by the invocation of operation-specific rules defined in the body of the operation.

4 Case Study

In order to evaluate the models, we conduct a case study demonstrating the efficacy of all three types. 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 are differences the most visible.

4.1 Assumptions

We model an issue tracker with multiple projects composed of issues. Each issue has a set of work logs and comments. The system maintains a catalog of users,



(a) Anemic domain model (b) Rich domain model of Comment class

Fig. 3: Application domain model

where each user might have a different role in each project. The behavior is as usual, system implements following use cases: Users browse, report, edit, and resolve issues, make comments and work logs depending on their project role. Administrators also maintain catalogs of users and projects. There are no other roles or use cases. In overall, we identify 92 business rules, i.e., 63 model constraints, and 29 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.

We consider the Java EE 7 platform with JDK 7 for future implementation of the plain three-layered architecture with ADM and AsDM. The model of the latter assumes JBoss Drools Expert 5.5[5] for business rules definition, and the Aspect Faces 1.4[6] framework implementing the concept proposed in [7] for the UI implementation. We will use our own implementation of the aspect weaving core based on AspectJ[7] as there is no other public implementation. Figure 3a shows an ADM of the application. The schema is identical for both these designs as their overall system architectures follow 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.

MVC-like web framework Django 1.8 built on the Python language deploys RDM and we use it as a target platform in our case study. While the basic domain model structure remains same, i.e., classes are identical, there are differences reflecting the RDM character of the design. The detailed model diagram is very complex 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 in Figure 3b. It shows the UML object diagram of the simplest class, the Comment. The rest of the diagram follows the structure from Figure 3a with objects from Figure 3b.

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

Criterion / Approach	**Anemic DM**	**Aspect DM**	**Rich DM**
Restated rules	29	0	20[A,B]
Rules locations	96	1	38[A]
Business services	5	5	unsupported
Validators	0	0	11
UI widgets	unsupported	12	12
UI layouts	unsupported	3	3
UI views	60	10	10
Client-side valid.	limited	supported	supported[C]

[A] Only in the domain model.
[B] Restated only references to validators.
[C] Requires mapping of validators to client-side technology.

Table 1: The overall design efficiency

4.2 Model efficiency

Our case study consists of 8 model classes restricted by 63 constraints, and 29 operation preconditions. We design this application three times, always for a different model. Table 1 shows ADM has no support for concerns separation and reuse, so we end up with manual repetition of the 29 constraints. The total 63 constraints located in 96 places throughout the application layer transactions and the presentation layer views. *JSR 303: Bean Validation* [1] enables partial business rules reuse, which improves the results. Without this the total number of constraints locations would be much greater. We repeat business rules as well as UI widgets and layouts to deliver all combinations of required UI. While we want to provide 3 layouts and 2 different widget sets for 10 views, we have to implement 3 times 2 times 10 views, i.e., 60 views in total. As we see, this architecture is unable to efficiently address cross-cutting concerns and usually ends up with highly tangled code with plenty duplications.

The results in Table 1 confirm that AsDM describes all business rules in a single place and ensures their automatic runtime transformation as there are no restated rules and their are located in a single focal point. Our implementation of the aspect weaver combines rules, UI widgets and layout templates together with the annotated behavior classes to produce the application.

Finally, RDM uses complex data types and reusable field and class-specific validators, which is captured in the results as 20 restated validator references in a model. We model business operations as methods over the model, which allows us to reuse model validators and combine them with operation-specific conditions. Customized renderers produce the UI as we discuss earlier.

Table 1 summarizes overall efficiency of considered models. We see ADM delivers the worst result, as there is major business rules and UI views repetition. Maintenance of such a system is error-prone and requires many efforts. It results from its inability to reuse cross-cutting concerns. The table also shows significant improvement of the design by deploying AsDM. While it preserves the system's model structure and architecture, it removes all sorts of repetitions and establishes maximal concerns reuse. For example, contemporary implementations of the approach allow business rules reuse in all layers of the system including the client-side UI. The overall results in Table 1 suggest, the approach delivers even better design than the other contemporary approach. Although RDM delivers much better results than ADM, there is still some repetition. The most significant disadvantage is absence of the single focal point, which is shown in the results as 38 rules places.

4.3 Concerns representation

Coupling, cohesion and encapsulation are crucial qualities [13] strongly influencing development efficiency. 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.

Unfortunately, ADM does not provide any mechanism to represent and reuse cross-cutts such as business rules, widgets, and layouts. The results show, it results in high information restatement (96 locations, 29 restated rules) and concerns tangling. Although there is a standardized technique JSR 303 to reuse some of the model constraints, it does not cover operation preconditions. This concern tangling preserves information encapsulation inside transactions but all concerns are highly coupled, which significantly reduces code cohesion.

This limitation is resolved by AsDM, which deploys DLSs to efficiently describe these concerns. It aggregates all concerns and provides a single focal point, single place to maintain, which results confirm. This model significantly increases cohesion and reduces coupling. However, it deploys multiple DLSs and distributes concerns into multiple aspects, which significantly reduces their encapsulation. In consequence, although there is a single place to update, it might be a bit difficult to track relations among model and all aspects. Especially when the weaving is context-aware and performed at runtime.

RDM keeps everything tangled in the model as is suggested in notes to the measured repetitions in Table 1. This approach ensures very good encapsulation in the object-oriented manner, but as every class maintains all sorts of concerns and does not have direct support of the runtime context, the concerns are tangled. Coupling is high and cohesion low. The benefit is, that these worsen qualities apply only inside of a class. Inter-class coupling is low. Also the strong encapsulation significantly simplifies views and controllers. Unfortunately, concerns representation in the domain model includes use of the programming language for their expression, which makes them difficult to transform. In consequence, this design leads to server-centric system, where all logic and rules are in the model in the server we are unable to automatically transform it into the other technology or to propagate it in the client's side.

These qualities apply also in UI. While neither ADM nor standard technologies support automated UI generation (60 resulting views), AsDM enables us to generate the UI from collected aspects. It weaves them together on every request to deliver the context-aware UI. RDM uses widget and layout renderers bound to a class and its fields. The build process triggers UI generation methods invoking all particular renderers composing the resulting UI in the class interface. This method enables partially self-maintainable UI, but as the model is unable to automatically transform other concerns, the resulting UI lacks the functionality. In consequence, we must manually customize renderers to mix it in.

4.4 Usage Efforts

The three-layered architecture with ADM is very straightforward to deploy. It uses only one programming language and usually one mark-up for the UI description. On the other hand, in order to use AsDM, it is necessary to apply multiple supportive tools and frameworks. As the key idea relies on multiple DLSs, it needs compilers/interprets, and complex aspect weaver to describe and integrate all captured concerns together. Furthermore, there are various transformation rules producing 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].

Contrary, RDM is the central source of information for most of systems concerns. Such a model does not require complex tools or frameworks since it uses a single (programming) language. Also the learning curve is much less. However, every change in the system must be done in the domain model, which must be done only by developers. Then it requires recompilation and new deployment of the application.

4.5 Threats to Validity

We identify several internal and external threats possibly affecting the results. Among internal threats we consider validity of compared designs following the considered model types. A Java expert proposes ADM and AsDM designs as we design for Java EE platform. A Python expert with long professional experience proposes RDM design. All designs are peer reviewed. Two peers conduct manual measurement of the results independently, the results are double-checked.

We recognize validity of the overall case study as an external threat. Our case study is a small representative of a real enterprise system. All use cases, scenarios, and model classes create a core of production-size issue trackers, we just reduce the scope of the system. In consequence, the measured results are scalable to the production-size system including system out of the issue tracking domain as we do not use anything specific to this domain.

5 Related Work

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

The difficulty of business logic description and maintenance is discussed in [8]. 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 [7]. 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 [6].

The focus on maintainable, object-oriented software belongs among best practices, which are covered by architectural and design patterns explained in [3, 5]. 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 system architecture.

Cross-cutting concerns impact design and code because they represent functionality and features, which affect multiple classes, components, and layers at once. 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, whose 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[8], 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 types of domain models, the MDD is aimed to simplify the development process with respect to design approach and target architecture. Thus it can be used with all considered models as long as we are able to transform the abstract model into the chosen platform.

6 Conclusion

EISs development faces the pressure of fast-growing complexity and scope. Selection of an approach significantly impacts system development and maintenance efforts. In this paper, we evaluate three types of domain models and discuss their efficiency to deal with the cross-cutting concerns.

Our case study shows that the Anemic Domain Model usually used within the three-layered architecture fails to address cross-cutting concerns such as business rules. It has no mechanism to represent and reuse them, which leads to high information repetition, low cohesion and high coupling. Maintenance of such a system is very expensive and error-prone. Bright side of this model is its very

[8] http://projects.spring.io/spring-framework/

straightforward use. This standardized design relies on a single programming language and is vastly supported by many tools and frameworks.

The Rich Domain Model used by many MVC-like web frameworks shows to be much more efficient with cross-cuts. It relies on rich classes and fields containing all possible information. Every field carries its own validators, renderers, labels, etc., which is the way to represent cross-cutting concerns. Finally, although the RDM has some boundaries and tends to high coupling and low cohesion, the supportive tools and frameworks aims to remain as simple as possible which flattens learning curve and makes its use very efficient in most cases.

Finally, the Aspect Domain Model implemented by Aspect-driven design approach extends the Anemic Domain Model within the three-layered architecture. It decomposes cross-cuts in the system and describes them individually in multiple DSLs in the model itself. Such design delivers a single focal point, a single place to update. Concerns are automatically weaved into the rest of the system at runtime. Furthermore, it transforms them into various components and technologies. Such isolation efficiently avoid information repetition and coupling. Easy maintenance, low development efforts, low coupling and high cohesion are direct consequences of the model. Unfortunately, there are significant initial costs; steep learning curve and complex tools and frameworks are required. Furthermore, contemporary implementations limit us in concerns transformation.

The results presented in this paper are solid input for more extensive, industry-related, case study. In future work, we conduct a case study measuring efficiency of AsDM in the real production-size application.

Acknowledgements

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

References

1. Bernard, E.: JSR 303: Bean validation (November 2009), `http://jcp.org/en/jsr/detail?id=303`
2. Fowler, M., Beck, K.: Refactoring: Improving the Design of Existing Code. Object Technology Series, Addison-Wesley (1999)
3. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
4. Fowler, M.: Anemic domain model. Availible at http://martinfowler.com/bliki/AnemicDomainModel.html (November 2003)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
6. 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)

7. 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)
8. 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)
9. Iglesias, C.A., Fernández-Villamor, J.I., Del Pozo, D., Garulli, L., García, B.: Combining domain-driven design and mashups for service development. Springer (2011)
10. Kennard, R., Edmonds, E., Leaney, J.: Separation anxiety: stresses of developing a modern day separable user interface. In: Human System Interactions, 2009. HSI'09. 2nd Conference on. pp. 228–235. IEEE (2009)
11. Kennard, R., Edmonds, E., Leaney, J.: Separation anxiety: stresses of developing a modern day separable user interface. In: Proceedings of the 2nd conference on Human System Interactions. pp. 225–232. HSI'09, IEEE Press, Piscataway, NJ, USA (2009), `http://portal.acm.org/citation.cfm?id=1689359.1689399`
12. Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.M., Lopes, C.V., Maeda, C., Mendhekar, A.: Aspect-oriented programming. In: In ECOOP'97-Object-Oriented Programming, 11th European Conference. vol. 1241, pp. 220–242. Springer (June 1997)
13. Larman, C.: 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 edn. (2001)
14. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (December 2005), `http://doi.acm.org/10.1145/1118890.1118892`
15. Muthanna, S., Ponnambalam, K., Kontogiannis, K., Stacey, B.: A maintainability model for industrial software systems using design level metrics. In: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00). pp. 248–. WCRE '00, IEEE Computer Society, Washington, DC, USA (2000), `http://dl.acm.org/citation.cfm?id=832307.837117`
16. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE software 20(5), 42–45 (2003)