An Insight on Standardized Patterns in Model-Driven Software Development

PETAR RAJKOVIĆ, IVAN PETKOVIĆ, ALEKSANDAR MILENKOVIĆ and DRAGAN JANKOVIĆ,
University of Niš

The area of model-driven software development covers many different topics starting from process and organization and then via domain modeling and tool architecture to application platform development. For this reason, it is important to impose a standardized set of recommendations, or patterns, that could be used both for new or ongoing projects. For this research, we focused on "fixed budget shopping basket" organizational pattern and the implementation of metamodel followed by ignoring concrete syntax and separation of generated and non-generated code. The main aim of this paper is to present a general overview and give insight from our development team on the usage of standardization and patterns in model-driven software development.

1. INTRODUCTION AND MOTIVATION

Model-driven software development (MDSD) is the area of software engineering that is continuously developing in the last three decades. There are many different approaches defined and many successful projects came out from MDSD biosphere [Mohagheghi et al. 2013] [Brambilla et al. 2017]. Nevertheless, it is still easy to find many sources that not support MDSD as the paradigm, but strongly opposing it [Sneed 2007] [Osis 2018]. One of the main reasons, that is often pointed out, is the lack of overall standardization in the area. This fact also leads to the situations that the significant number of project managers tries to avoid MDSD in longer projects since they cannot see the clear benefits from the first sight [Klien and Ludin, 2019] [Vijayasarathy and Butler, 2015]. On the other hand, it is not easy to exactly measure the effect of many MDSD standard approaches [Christensen and Ellingsen, 2016] [Bolender et al. 2017], since when they got applied in significantly different environments they end up with different results. Both these two facts lead to the situation that MDSD still is not on the position in software development that, in our opinion, it should be.

The development team from our Laboratory of Medical Informatics advocates MDSD for the last 15 years stating that many parts of the projects can be finished faster and more reliably [Rajkovic et al. 2015] [Rajkovic et al 2017]. Primarily, we develop medical information systems based on the OpenEHR metamodel and with a set of data modeling and data generation tools following the standards given initially by Voelter [Voelter 2004]. Beside for medical information systems, we successfully used MDSD as a basic approach in manufacturing execution system development projects [Aleksic et al. 2017]. Usage of modeling and generation tools was proven successful and

This work is supported by the Ministry of Education and Science of Republic of Serbia (Project number #III47003).
Authors' addresses: Petar Rajkovic, University of Nis, Faculty of Electronic Engineering – Lab 534, Aleksandra Medvedeva 14, 18000 Nis, Serbia, e-mail: petar.rajkovic@elfak.ni.ac.rs; Ivan Petkovic, University of Nis, Faculty of Electronic Engineering – Lab 523, Aleksandra Medvedeva 14, 18000 Nis, Serbia, e-mail: ivan.petkovic@elfak.ni.ac.rs; Aleksandar Milenkovic, University of Nis, Faculty of Electronic Engineering – Lab 533, Aleksandra Medvedeva 14, 18000 Nis, Serbia, e-mail: aleksandar.milenkovic@elfak.ni.ac.rs; Dragan Jankovic, University of Nis, Faculty of Electronic Engineering – Lab 320, Aleksandra Medvedeva 14, 18000 Nis, Serbia, e-mail: dragan.jankovic@elfak.ni.ac.rs;.

In: Z. Budimac and B. Koteska (eds.): Proceedings of the SQAMIA 2019: 8th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, Ohrid, North Macedonia, 22–25. September 2019. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

made our development process faster, testing process more standardized and help us in an overall bug rate reduction [Rajkovic et al 2015].

After several successful implementation and upgrade projects based on model-driven engineering, we wanted to contribute to the MDSD community by supporting the standardization processes through this, rather technical, paper. Since MDSD is mostly presented through the papers that explain and evaluate specific approaches and coding techniques, with a little or no attention to identify and promote certain standards. Thus, an important segment of the paradigm that we wanted to present here are standardized parts in MDSD and enrich the initial definition with our insights.

One of the most important sources for existing standardization in MDSD is the original work [Voelter 2004]. Claiming [Alexander 1977] and [Gamma 1995] as the motivational starting point and influenced by [Bettin 2002] and [Bettin 2003], the author tried to formalize the systematics in MDSD processes. The ideas shown in the mentioned work are further developed, and elaborated in [Buschmann 2007], [Possatto et al. 2015] and [Syriani et al. 2018].

Since the area of MDSD became wide and voluminous it was necessary to identify the most important sub-concepts and to join the most important coding techniques, processes, routines and organizational sets of recommendation into logical groups. According to [Voelter 2004] [Strembeck 2009], they are structured and organized in four main groups - domain modeling, process and organization, tool architecture and application platform development. Since these groups are different by its content and focus, the common name must be chosen to describe any of the items for any of the groups. The name pattern itself was chosen in [Voelter 2004] after Alexander's [Alexander 1977] discussion on patterns – "pattern represents our current best guess as to what arrangement of the physical environment will work to solve the problem presented. The empirical questions center on the problem—does it occur and is it felt in the way we describe it?—and the solution—does the arrangement we propose solve the problem?" and "the patterns are still hypotheses, all 253 of them—and are, therefore, all tentative, all free to evolve under the impact of new experience and observation". The usage of the term pattern could seem unappropriated due to its formal definition, and even though the work [Alexander 1977] is from the area of civil engineering and architecture, it is adopted as the common name for all the MDSM standardized processes, software patterns, and organizational routines.

Starting from the domain modeling step we can say that this area is well covered by many different tools and approaches such are architectural-centric metamodel and formal metamodel. Our experience with data modeling is based on the and usage of locally developed modeling tools together with standardized metamodels. It gives us the results as presented in [Rajkovic et al. 2015].

The main challenge we wanted to point attention here is from the area of process and organization. The pattern is called fixed budget shopping basket and it should give the recommendations for project managers how to reach the goal in a given time with an assigned workforce. Since our experience says that the projects are often understaffed and everchanging (from the point of view of the requirements collection), the existence of any software tool that can speed up the development process is more than desired. For this reason, using the most effective way to generate code from the model is the next major area where standardization can help. Our research showed that the generator tool can reduce the time needed for development up to two thirds [Rajkovic et al. 2015]. To us was essential to follow the pattern of the implementation of the metamodel when comes to the tool architecture. In the application platform development area, we tend to have an adaptable system that can easily switch to the different programming environment and to generate a series of different software component. From this area we identified ignoring concrete syntax and the separation of generated and non-generated code as the most important patterns we used to follow.

In this paper, we will present mentioned four (fixed budget shopping basket, implementing a metamodel, ignore concrete syntax and separation of generated and non-generated code), at least for us, the most influential patterns and give the general overview and our internal insights.

2. FIXED BUDGET SHOPPING BASKET

Starting from process and organization, the one common problem in the MDSD engineering process is work organization in a way to develop a product aligning the requests from the customer in a given amount of time. From the product manager's point of view given amount of time multiplied by the cost per hour makes a nightmare called fixed budget. In that light, the pattern imposing application of the iterative development to ensure proper result on time is named as a fixed budget shopping basket. The initial definition presented in [Voelter 2004] and further developed in [Serrador et al 2015] and [Gregory et al. 2016] contains the following rules:

- *Use timeboxed iterations that are shorter than six weeks, validated by users/customers.*
- *Produce shippable code at least every three months.*
- *Ideally, deploy into production every three months to get "live" feedback.*
- *For the development of new business applications, "go live" within nine months, don't risk losing the team (mother) or the application (baby).*

Developing large scale projects is generally risk itself. When projects must last at least half of a year, there is a significant chance that problems either on the side of the customer or on the side of the engineering causes the delay. For this reason, project execution in pre-defined timeboxed iterations and validation sessions with customers is required. From our experience, the period should be shorter than six weeks, and usually weekly or bi-weekly meetings proved to be more convenient. Within the period of six weeks, the customer can easily lose the focus and start thinking in the directions different than specified in stakeholder requests. In [Rajkovic et al. 2013] we published the results based on our experience with different level of interaction with the customers. We had three focus clinics in our project, and, due to the differences in their organization, we established different communication routines. The best results were with the development of an information system for cardiological clinic beside the volume of the required work significantly exceeded one needed for other two clinics. Proper communication helped in later full software acceptance. In other two clinics we had only few meetings for over a year, and the level of common understanding was significantly lower, and the project took 50% longer to finish where not all the modules got accepted.

Better communication when developing an information system for cardiological clinic helped in the development process itself, especially in earlier bug discovery. Going in this line, production of shippable code also should be in iterations that lasts two to three weeks. Period of three months is longer than most of the end-users can stay active. More frequent deployments in the test environment will keep the customer interested and keep for the project.

On the other hand, deployments to the production environment should be less frequent than deployment to test suite. Regarding the deployment plan for the production server, it seems that it is more convenient not to have strict time defined deployment slots, but rather after a specific set of functionalities realization.

This approach could also be improved by iterative multi-track development. Depending on several team members the team can be divided, and the complete process could be split into several tracks and proper development approach and methodology can be applied. From the technical point of view, this approach seems optimistic, but in real development, many other problems can appear that can prevent the customer from getting an active system in desired time for a fixed budget. For example, during the iterations, the customer will easily get to the idea that something, in addition, should be

developed. Even when the customer accepts to sign the change request, the negotiation itself will take additional time which can delay the overall project.

This approach is also useful when it comes to the deployment of the developed information system in a medical institution. Using the interactive multi-track development approach we managed to reduce the time needed to deploy and customize software in Serbian medium-size primary care medical facilities by 50%. Initially we needed 60 days full of stress, but eventually we ended up with a less than one month, where number of support calls got reduced to about 30% [Rajkovic et al. 2015].

It is even harder to plan projects that should last longer. As longer the project timeline is, there are significantly more chances that the project will not be delivered on time and within budget. Eventually, the key to success in these types of projects is a mutual understanding of both sides and continual communication.

3. IMPLEMENTING A METAMODEL

Using models to improve software development is the leading idea of MDSD. The process should start with defining or adopting a metamodel, then creating specific models and based on them generating a set of various software components. This is an excellent idea and a streamlined process that looks logical and straightforward. Unfortunately, there are many technical problems hidden behind this process.

The first point to ensure is validation. Specific models must be validated against the metamodel and, at the time of software generation, generated components must be checked for consistency against the specific model. Thus, the definition from [Voelter 2004] [Wachsmuth, 2007] [Mens e al 2016] states: "*Implement the meta-model in some tool that can read a model and check it against the metamodel. This check needs to include everything including declared constraints. Make sure the model is only transformed if the model has been validated against the metamodel*". The definition for this pattern gives the only direction on "what", but not "how" the process should be done.

Once agreed with this, the next point is – "how to perform the validation". Should this be done manually, or through some of the available pieces of software or through some of the homebrewed generator tools? Manual validation is, in most cases, out of the scope. Models tend to be complex and manual check is usually time-consuming task with a high probability of errors. Next choice is to use existing model transformation pieces of software. In many cases, the organizations that maintain metamodels for some domains, also offer some set of validation tools. These tools could be efficiently used for specific model validation, but the situation with data generation tools is a bit different. The level of customization for a specific project could be significantly higher than support, so developers should choose between:

- Using the existing tool as much as possible and then perform manual coding for the missing parts
- Extend the existing tool to suit most of the needs and then perform manual coding for the missing parts
- Develop specific code generation tools/engines that would be able to generate specifically required components and then perform manual coding for the missing parts

In all three cases, the common part is "perform manual coding for the missing parts". Analysis of this segment is crucial to choose the strategy. Since the main aim of MDSD is to reduce the amount of the specifically typed manual coding, this is the key part in the whole MDSD, how we see it. Once the correct choice is done here, the implication for the project will be higher. Unfortunately, developing code generation frameworks/tools will take some time and can slow the initial development of all the "items from the shopping basket".

From our experience, it is worth to invest time in specific tool development especially when the team is focused on specific domain development. If a team is doing one spin-off project for a specific domain, then is questionable if this kind of development is needed at all. Using the existing tools is useful in most of the cases when specific models should be validated against the domain model. For large scale organization like OpenEHR or ISA-95, this looks prominent, but for some other organizations, the quality of offered tools could be questioned [Rajkovic et al. 2014].

We have experience with both approach and depending on a step in the software cycle, the usage of one or another has own pros and contras. The standard generation tools are usually better fitted for one category of generated components, but for other cases, custom-built tools have own advantage. In [Rajkovic et al. 2015] we compared our results for component generation. For example, standard tools had a slight advantage when generating classes and standard logic for the components such are Windows forms, but our custom build generator tools gave better results when some specific parts should be generated such are custom logic generation, automatic configuration files creation and automatic test case builds. Our experience is that custom build MDSD suite can reduce development time in some steps up to 50%.

Since project managers need to balance the shopping basket, the decision is mostly on them. We just need to point out that our view is maybe biased by the fact that the domains of our projects (medical information systems and manufacturing execution systems) came from highly regulated areas and some team members are continuously involved in development for last 15 years.

4. IGNORE CONCRETE SYNTAX

Ignoring concrete syntax during code generation as longest as possible is a pattern which requirements are often neglected during the development of code generation tools (Fig. 1.). Again, looking at the whole development process as "the shopping basket" with limited resources, the project managers strived to bring the results sooner as possible and thus tend to reduce all of the unnecessary developments and then got some more time for "more important parts" of the project. This greedy approach can bring benefits in cases when the organization works with a single technology and has no plans to change it in longer terms, or when the generated components are relatively simple and require no further updates after generation.

The directive from [Voelter 2004] [Paige 2016] defines the process around model transformation in the exact following three phases:

- *convert the input model into some in-memory representation of the metamodel (typically an object structure),*
- *then transforms the input model to the output model (still as an object structure)*
- *transform the target model to a concrete syntax.*

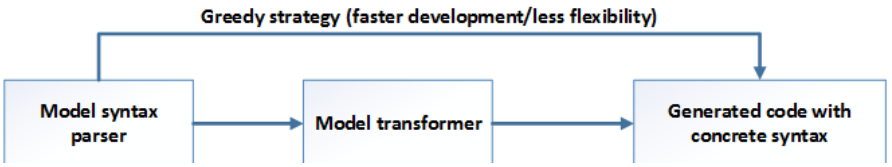

Fig. 1. Steps in ignore concrete syntax pattern (adopted from [Voelter 2004])

This pattern requires an implementation of the parser for the concrete syntax of the application model, then transformer that converts input (meta) model to specific model, and eventually the "unparser" or "petty printer" that will eventually convert the model to the concrete syntax.

Usual implementations tend to skip the second step and to implement direct transformation from the initial model to the textual output. This will reduce the system's flexibility but speed up the development to approximately half of the required time.

Initially, we tend to go for the mentioned greedy approach, especially because all our models were from the same category and internal transformations will bring no significant, by our opinion at the time, benefits. Eventually, we switch to the full three-stage model and successful implementation is later used for manufacturing execution systems [Aleksic et al 2017].

5. SEPARATION OF GENERATED AND NON-GENERATED CODE

When comes to the generated parts of the code it is important to clearly define how the generated code should be maintained. Generated code should be handled in such a way that will not interrupt manually written code. The most common problem is the fact that code generators will completely re-generate its parts of the code. Any change in manual code will be then lost and such changes are hard to track and revert if needed. Two most common approaches are the separation of generated and non-generated code and so-called forced pre and post code. The definition from [Voelter 2004] states:

Keep generated and non-generated code in separate files. Never modify generated code. Design an architecture that clearly defined which artifacts are generated, and which are not. Use suitable design approaches to "join" generated and non-generated code. Interfaces, as well as design patterns such as factory, strategy, bridge, or template method, are good starting points.

The code generation itself is not one simple process, especially because of different possible relations between generated and non-generated code. Depending on the relation between the generated and non-generated code different types of relations should be considered. In [Voelter 2004] and [Torres, et al. 2017] five base cases are examined (Fig. 2.):

a) Generated code calls non-generated code
b) The non-generated code calls generated code
c) The non-generated code calls generated code through the adapter class
d) Generated classes are sub-classes of non-generated classes
e) Usage of template method pattern

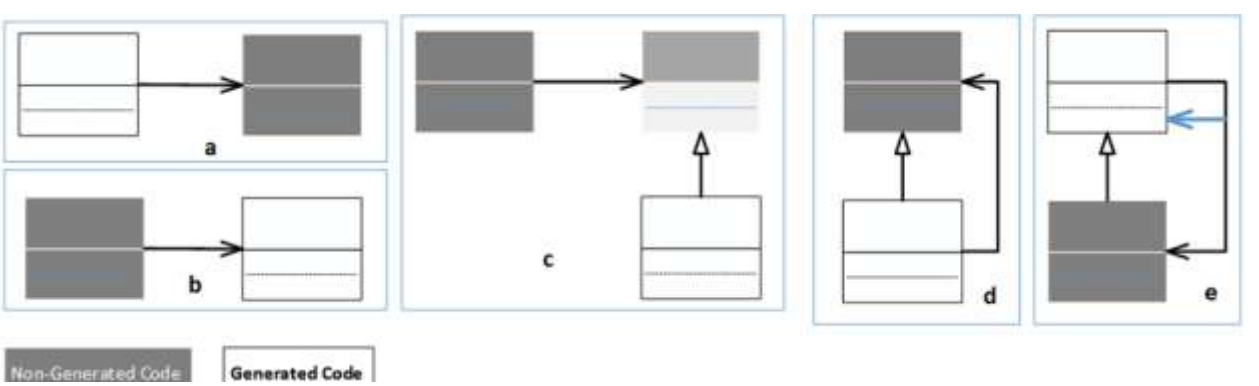

Fig. 2. Five major relations between generated and non-generated code (adopted from [Voelter 2004])

When generated components call the non-generated library, case a is the safest environment for the code execution. The generated code relies on already running and validated functionality which means that any new bug that could appear, should appear in the generated code. Testing, in this case, can then be reduced only to test the newly created library. We assume this approach as the easiest and the most convenient for implementation. It is used when the array of components

sharing the same existing functionality should be automatically generated by the model, and where no additional dependencies are introduced.

Case b is the situation when an existing non-generated code should call the functionality from the generated code. This approach is used when some options need to be introduced into the existing functionality and it usually requires some intervention on the existing code. This intervention usually consists of introducing abstract classes or extracting specific methods and make them virtual. Then, the base functionality calls, by the default its initial behavior, while newly generated code contains the implementation of abstract or virtual methods.

The extension of this approach is the case c when the link between non-generated and generated code is an adapter class. An adapter class is built on the way that is partly generated, since the methods needed to interact with the non-generated code came from the existing library, and the adaptation methods are generated. We used this approach to handle situations where different plug-ins or completely interchanged functionality are developed. For example, the classes that support connection to different data sources are implemented this way [Rajkovic et al. 2015].

The case d is basically the extension of the case a. Generated code calls non-generated, but generated class is defined as a sub-class. This approach is used when generated code shares many functionalities with the base code, but some minor changes are applicable from case to case. We used them for different validators when validation against the same set of values could be altered to show some different aspect – i.e. in the description of neurological statuses.

The last of the presented cases is the case when the template method pattern is used during the generation. Base class (non-generated code) contains one template method (which can even be virtual) and few virtual (or abstract) and non-virtual methods. Template method consists of the sequence of calls which is strictly defined. The generated code contains classes that derive a base class and implements or overrides only requested method. In later usage, the proper instantiation leads to the execution of specific code. Within our system, we used this approach when developing generator tools. Since we use the same model to develop multiple different classes of the components, we used this approach to speed the implementation up and to make the process more effective.

6. CONCLUSION

MDSD is generally envisioned as a paradigm that should make easier to handle all the major steps of the information systems life cycle. Considering all levels of complexity, the time needed for system development is, often, much longer than it is envisioned by project management. Colloquially, the system will not fit in its shopping basket. Making the software project on the way to fit in budget and time constraints was one of the requests that drive MDSD evolution. Evolution itself went through many steps and it is still active. The new generation of MDSD frameworks offer much more than even five years ago and, in some area, such as an object-relational model representation of databases, they are considered as a standard.

On the organizational side of the process, we decided to present a fixed budget shopping basket as one of the hardest to achieve goals in project management. Along with basic recommendation, we present our findings that are in favor of the general opinion, but with a higher focus on communication with customers.

Implementing a metamodel is considerably core of the MDSD. Generating code based on the model and having in mind potential compromises on the way from system flexibility to fastest deployment is discussed here and enriched by the examples from our development projects. Two additional patterns important for the code generation itself is mentioned too – ignoring concrete syntax and separation of generated and non-generated code.

Since MDSD is a considerably wide area, starting from process organization and ending with code generation, one can tell that it lacks specific focus and that is out of standardization. With this

paper, we wanted to present major parts of the MDSD environment with belonging standardized sets of recommendation (or patterns) followed by our insight based on the decade and a half long experience of our development team.

REFERENCES

Alexandre Torres, et al. 2017, "Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design." information and software technology 82 (2017): 1-18.

Bente Christensen and Gunnar Ellingsen. Evaluating model-driven development for large-scale EHRs through the openEHR approach. International journal of medical informatics 89 (2016): 43-54.

Christopher Alexander. A pattern language: towns, buildings, construction. Oxford university press, 1977.

Dejan Aleksic, Dragan Jankovic, Petar Rajkovic, 2017, Product configurators in SME one-of-a-kind production with the dominant variation of the topology in a hybrid manufacturing cloud, Int. J. Adv. Manuf. Technol. 92 (2017) 2145–2167. DOI:10.1007/s00170-017-0286-1. (M22, IF 2.748) (https://link.springer.com/article/10.1007/s00170-017-0286-1)

Erich Gamma 1995, Design patterns: elements of reusable object-oriented software. Pearson Education India

Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. 2018, "Systematic mapping study of template-based code generation." Computer Languages, Systems & Structures 52 (2018): 43-62.

Frank Buschmann, Kevin Henney, and Douglas C. Schmidt. 2007. Pattern-oriented software architecture, on patterns and pattern languages. Vol. 5. John Wiley & sons, 2007.

Guido Wachsmuth, 2007, "Metamodel adaptation and model co-adaptation." European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, 2007.

Harry M. Sneed, 2007, "The drawbacks of model-driven software evolution." IEEE CSMR 7 (2007).

Janis Osis, and Erika Asnina. 2018, "Is Modeling a Treatment for the Weakness of Software Engineering?." Intelligent Systems: Concepts, Methodologies, Tools, and Applications. IGI Global, 2018. 310-327.

Jorn Bettin. 2002. Measuring the potential of domain-specific modeling techniques. Proceedings of the 2nd Domain-Specific Modelling Languages Workshop (OOPSLA), Seattle, Washington, USA. 2002.

Jorn Bettin. 2003. Ideas for a Concrete Visual Syntax for Model-to-Model Transformations. Proceedings of the 18th International Conference, OOPSLA. 2003.

Leo Vijayasarathy, and Charles Butler. Choice of software development methodologies: Do organizational, project, and team characteristics matter? IEEE Software 33.5 (2015): 86-94.

Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017, "Model-driven software engineering in practice." Synthesis Lectures on Software Engineering 3.1 (2017): 1-207.

Marcos Antonio Possatto, and Daniel Lucrédio. 2015, Automatically propagating changes from reference implementations to code generation templates. Information and Software Technology 67 (2015): 65-78.

Marcus Voelter 2004, Patterns for Model-Driven Software-Development, http://www.voelter.de/data/pub/MDDPatterns.pdf

Mark Strembeck, and Uwe Zdun. 2009, "An approach for the systematic development of domain-specific languages." Software: Practice and Experience 39.15 (2009): 1253-1292.

Parastoo Mohagheghi et al. 2013, "Where does model-driven engineering help? Experiences from three industrial cases." Software & Systems Modeling 12.3 (2013): 619-639.

Pedro Serrador and Jeffrey K. Pinto. 2015, "Does Agile work?—A quantitative analysis of agile project success." International Journal of Project Management 33.5 (2015): 1040-1051.

Peggy Gregory et al. 2016, "The challenges that challenge: Engaging with agile practitioners' concerns." Information and Software Technology 77 (2016): 92-104.

Petar Rajković, Ivan Petković, Dragan Janković, Case Study: Using Model Based Component Generator for Upgrade Projects, in: Proc. Sixth Work. Softw. Qual. Anal. Monit. Improv. Appl., Belgrade, 2017: p. 13:1-13:8. http://ceur-ws.org/Vol-1938/paper-raj.pdf

Petar Rajkovic, Dragan Jankovic, Aleksandar Milenkovic 2013, Developing and Deploying Medical Information Systems for Serbian Public Healthcare - Challenges, Lessons Learned and Guidelines, Computer Science and Information Systems (ComSIS), vol. 10, no. 3, pp. 1429-1454, doi.org/10.2298/CSIS120523056R, (M23, IF 0.675) http://www.comsis.org/archive.php?show=pprdev2-04

Petar Rajković, Dragan Janković, Aleksandar Milenković, 2014, Improved Code Generation Tool for Faster Information System Development, SAUM 2014, Nis, Serbia, 12 - 14 November 2014, pp. 273 -276, Conference Proceedings, ISBN: 978-86-6125-117-7.

Petar Rajković, Ivan Petković, Dragan Janković. 2015, Benefits of Using Domain Model Code Generation Framework in Medical Information Systems. Fourth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications SQAMIA 2015, pp. 45-52, Maribor, Slovenija, Jun 8 – 10th, 2015, http://ceur-ws.org/Vol-1375/SQAMIA2015_Paper6.pdf

Ralph Kliem and Irwin Ludin. Reducing project risk. Routledge, 2019.

Richard F. Paige, Nicholas Matragkas, and Louis M. Rose. 2016, "Evolving models in model-driven engineering: State-of-the-art and future challenges." Journal of Systems and Software 111 (2016): 272-280.

Tom Mens, and Pieter Van Gorp. 2016, "A taxonomy of model transformation." Electronic Notes in Theoretical Computer Science 152 (2006): 125-142.
Tim Bolender, Bernhard Rumpe, and Andreas Wortmann. "Investigating the Effects of Integrating Handcrafted Code in Model-Driven Engineering." MODELS (Satellite Events). 2017.