Synergy of Organizational Patterns and Aspect-Oriented Programming

1st Peter Berta
Institute of Informatics, Information Systems and Software Engineering, Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Slovakia
pepoberta@gmail.com

2nd Valentino Vranić
Institute of Informatics, Information Systems and Software Engineering, Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Slovakia
vranic@stuba.sk

***Abstract*—By observing Conway's law, which explains how organizing people directly affects the code they produce, and by taking into account the specifics of different programming paradigms, a lot can be done to improve the software development process. We analyzed organizational patterns of agile software development from the perspective of aspect-oriented programming, whose aim is to improve the separation of concerns in code, which, in turn, allows for better separation of tasks performed by people. We find aspect-oriented programming to be highly related to at least nine out of more than a hundred of organizational patterns in Coplien and Harrison's catalog, namely: Work Split, Team per Task, Sacrifice One Person, Divide and Conquer, Conway's Law, Form Follows Function, Shaping Circulation Realms, and Hallway Chatter. Aspect-oriented programming can support organizational patterns in division of labor, treating distractions, and increasing decoupling between the software modules developed by different parts of the organization. On the other hand, organizational patterns can help mitigate pointcut fragility and aspect obliviousness in general.**

***Index Terms*—modularization, asymmetric aspect-oriented programming, symmetric aspect-oriented programming, organizational patterns, Conway's law, use cases, people, agile software development**

I. Introduction

Conway's law reveals a direct connection between people and code. It states that organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations [1]. Consequently, an organization that develops a software system should be structured the way that this software system is desired to be structured.

To structure means to separate components and interlink them in an organized way. This is driven by a human need to act separately. Programming paradigms have their limitations in what can be separated. For example, in Java, a method can't be defined outside a class and then made a part of it. But using an aspect-oriented extension to Java called AspectJ, this can be achieved.

Good ways of structuring software development organizations have been documented as organizational patterns [2]. As with other software patterns and patterns in general, they deal with balancing contradicting forces. For example, (software) architects need to focus on the overall structure, but they should not lose contact with the development reality. The Architect also Implements pattern puts these forces into a balance by letting the architect participate in actual programming [2].

In this paper, we analyze what organizational patterns are particularly suitable for aspect-oriented programming and vice versa. For this, we apply scientific reasoning to identify analogies between aspect-oriented programming techniques and organizational patterns. This analysis might be useful to those who would like to apply aspect-oriented programming and good organizational practices along. Section II briefly explains aspect-oriented programming. Section III explains how we approached identifying suitable organizational patterns. Section IV presents the project management patterns and their relationship to aspect-oriented programming. Section V presents the organizational style patterns and their relationship to aspect-oriented programming. Section VI discusses related work. Section VII concludes the paper.

II. Aspect-Oriented Programming

At the peak of the popularity of aspect-oriented programming, there were dozens if not hundreds of aspect-oriented programming languages available. Given the nature of aspect-oriented programming, they mostly represented aspect-oriented extensions to established programming languages. Consequently, there were many variants and classifications of aspect-oriented programming. Among these, a particularly important distinction is whether aspects are perceived as special modules that affect the base modules or the whole design is built out of aspects. The former is known as asymmetric aspect-oriented programming, while the latter is being denoted as symmetric aspect-oriented programming [3].

Although originally related only to HyperJ [4], an abandoned IBM's prototype aspect-oriented programming language, symmetric aspect-oriented programming features can be observed in programming languages not explicitly denoted as aspect-oriented, such as Scala (traits), Ruby (open classes), and JavaScript (prototypes) [5]. Furthermore, symmetric aspect-oriented programming can be emulated in AspectJ [5], the reference asymmetric aspect-oriented programming language. It is worth noting that peer use cases (use cases with no dependencies between them) essentially represent symmetric aspect-oriented modularization [5]. Consider the use case diagram depicted in Figure 1. Place an Order and Cancel an Order are two peer use cases. As can be seen in Figure 2, the realization of these use cases is based around the same classes, but seen differently from the perspective—or aspect—of each use case.

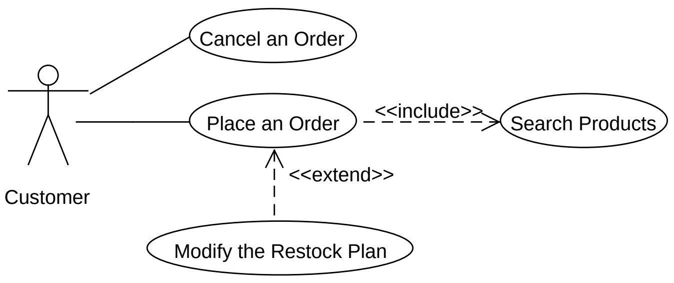

Fig. 1. Use cases as aspects.

The extend relationship represents an instance of asymmetric aspect-oriented modularization, as observed by Jacobson [6], [7]. In the use case diagram depicted in Figure 2, the Modify the Restock Plan use case affects the Place an Order use case with the latter not being aware of this. Contrast this to the include relationship, in which Place an Order explicitly invokes Search Products. This is more evident in the actual use cases:

UC Place an Order

Basic Flow: Place an Order

1. Customer selects to place an order.
2. UC Search Products is being activated.
3. Customer confirms the product selection and adjusts its quantity.
4. If the product is available, System includes it in the order.
5. Customer continues in ordering further products.
6. Customer chooses the payment method, enters the payment data, and confirms the order.
7. Customer can cancel ordering at any time.
8. The use case ends.

Extension points:
Checking Product Availability: Step 4

UC Modify the Restock Plan

Alternate Flow: Modify the Restock Plan

After the Checking Product Availability extension point in the Place an Order use case:

1. System checks the available quantity of the product being ordered.
2. If the quantity is below the limit, System adds the quantity under demand to the restock plan.
3. The flow continues with the step that follows the triggering extension point.

In AspectJ, the extend relationship can be preserved by the following aspect, which enforces restocking the plan instead of having to call this functionality explicitly from the Ordering class:

```
public aspect RestockPlan {
    ...
    void around(Product product):
        call(* Ordering.productAvailable(..) &&
            args(product) {
        ... // increase the product quantity in the restock plan
    }
    ...
}
```

Speaking more generally about AspectJ, which is the most elaborated representative of asymmetric aspect programming, we may say that aspects modify program execution in join points, such as method call or execution, constructor call or execution, or field (attribute) access. Join points are determined by pointcuts, which can be seen as declaratively specified sets of join points. Modifications are expressed as so-called advice. A piece of advice is the code executed before, after, or around (instead of) a join point. Aspects can add new elements and inheritance relationships to classes using so-called inter-type declarations, which is a way to emulate symmetric aspect-oriented programming in AspectJ [5].

III. Identifying Organizational Patterns Related to Aspect-Oriented Programming

In identifying organizational patterns suitable for aspect-oriented programming, we focused on Coplien and Harrison's comprehensive catalog [2], which includes over a hundred patterns forming four interlinked pattern languages. This catalog might be denoted as a culmination of the work that can be traced back to at least 1994 [8]–[10]. Furthermore, Sutherland, Coplien, et al. rephrased Scrum into organizational patterns [11]. All the while, Ambler talks about process patterns [12], and some other authors took the route of warning about bad organizational practices expressing them as antipatterns [13]–[15] (a unifying antipattern catalog has been reported [16]).

In the following two sections, we present the organizational patterns from Coplien and Harrison's catalog we found to be related to aspect-oriented programming. However, we do not claim there are no other organizational pattern from this catalog or in general to be related to aspect-oriented programming.

IV. Project Management Patterns

In the Project Management pattern language, we identified four patterns related to aspect-oriented programming: Work Split, Team Per Task, Sacrifice One Person, and Surrogate Customer. Figure 3 depicts their position within this pattern language. The Size the Schedule pattern is introduced as the

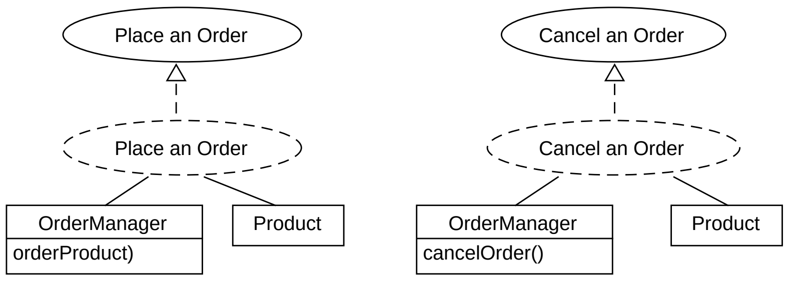

Fig. 2. Realization of peer use cases in the symmetric aspect-oriented way.

root of the Project Management pattern language despite it wasn't among the patterns we identified as being related to aspect-oriented programming. The edges depict the order of the application as proposed by Coplien and Harrison [2]. Dashed edges mean that the order is indirect, i.e., there are other patterns in between.

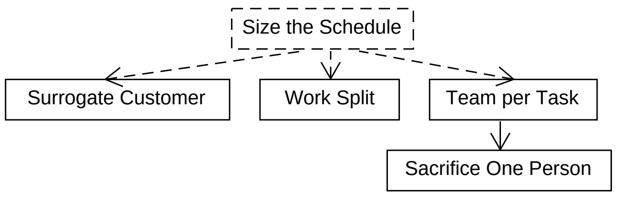

Fig. 3. The patterns related to aspect-oriented programming within the Project Management pattern language.

Each of the following sections treats one organizational pattern. It first brings its brief description based on Coplien and Harrison's catalog [2]. Subsequently, the relationship to aspect-oriented programming is explained.

A. *Work Split*

During the software development process, big tasks can make it seem that no progress is being made. To make the progress visible, the team should split the task at hand into two parts: the urgent part and deferred part. By focusing on the urgent part, the tasks will get finished and progress will be made, relieving the team from the feeling of being stuck. This is the Work Split pattern.

However, once the team gets onto the deferred part, it may become necessary to adapt the urgent part. Consider a complex use case with several alternative flows or a use case extended by one or more use cases. Obviously, the basic flow of this use case would be an urgent part, while alternative flows and extension use cases would be deferred. In effect, alternative flows are internal extensions, so in both cases asymmetric aspect-oriented programming can be used as presented in Section II.

B. *Team per Task*

A crisis occurring during the software development process needs to be handled. To ensure that progress will be made and issues solved, a subteam is created to solve the crisis, allowing the main team to keep working on the main line. This is the Team per Task pattern.

Dividing large development teams into smaller, dedicated teams, when done correctly, is beneficial for the entire software development process. Developers within a dedicated team can fully focus on the task assigned to them. They do not need to draw their attention to other problems that are not theirs to solve. To fully exploit the benefits of such division of labor, the dedicated teams can utilize aspect-oriented programming to keep the implementation of the tasks less coupled. Consequently, they will interfere with each other to a lesser extent and will be able to make progress independently.

C. *Sacrifice One Person*

The Sacrifice One Person pattern resolves a similar situation as the Team per Task pattern (Section IV-B). The main team can work on the assigned task, while one person is working on several minor distractions. This one person can handle the distractions using aspect-oriented programming to keep the main team free of having to adapt their implementation in order to incorporate the implementation of the distractions. Instead, with asymmetric aspect-oriented programming, the code resulting from the distractions can affect the corresponding join points in the code resulting from the main task.

D. *Surrogate Customer*

When a software development team needs to make a decision about certain requirements, and the customer is not available, a surrogate customer role may be created. It is assigned to one member of the software development team, who acts and thinks like a customer. This is the Surrogate Customer pattern.

Surrogate Customer role resembles the Cuckoo's Egg aspect-oriented design pattern [17]. With the Cuckoo's Egg pattern, a given object can be replaced by another object, which usually exhibits a (slightly) different behavior. The surrogate customer role can be seen as this new object that is replacing the old object, which, in turn, corresponds to the missing or dysfunctional customer. This is as if aspect-oriented design is applied to organizing people.

V. Organizational Style Patterns

In the Organizational Style pattern language, we identified five more patterns related to aspect-oriented programming:

Divide And Conquer, Conway's Law, Form Follows Function, Hallway Chatter, and Shaping Circulation Realms. Figure 4 depicts their position within this pattern language. As with the Project Management language treated in the previous section, the root of the Organizational Style pattern language, the Few Roles pattern, is introduced despite it wasn't among the patterns we identified as being related to aspect-oriented programming. Again, the edges depict the order of the application as proposed by Coplien and Harrison [2], while dashed edges mean that the order is indirect, i.e., there are other patterns in between.

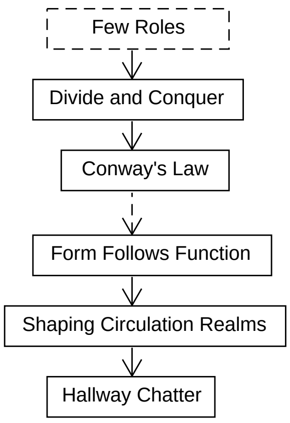

Fig. 4. The patterns related to aspect-oriented programming within the Organizational Style pattern language.

As with the Project Management pattern language, each of the following sections treats one organizational pattern. It first brings its brief description based on Coplien and Harrison's catalog [2]. Subsequently, the relationship to aspect-oriented programming is explained.

A. Divide and Conquer

Large organizations are hard to maintain and lead. Assigning tasks or communicating requirements among sizable teams is difficult. To make it easier, people should be divided into teams based on their roles. This is the Divide and Conquer pattern.

Dividing people into independent teams and assigning them software modules to be developed directly may be easier with symmetric aspect-oriented programming. This is appropriate for dealing with peer use cases (recall Section II). As with the Team per Task pattern (recall Section IV-B), the teams will interfere with each other to a lesser extent and will be able to make progress independently.

B. Conway's Law

As has been said in the introduction, Conway's law states that organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations [1]. Consequently, an organization that develops a software system should be structured the way that this software system is desired to be structured. This can be perceived as an organizational pattern: the Conway's Law pattern.

This is a general pattern and, in general, aspect-oriented programming can help in achieving a greater level of decoupling between the software modules developed by different parts of the organization. This includes reversing dependencies if necessary: recall the implementation of the extend relationship between use cases presented in Section II).

C. Form Follows Function

The Form Follows Function pattern recommends grouping together roles that are similar, manipulate similar artifacts, or operate within the same domain.

One of the main goals of aspect-oriented programming is to untangle source code and make it more comprehensible. Form Follows Function also focuses on grouping similar roles and creating a system from disarranged structure. By creating groups with similar activities and employing symmetric aspect-oriented programming similarly to the implementation of peer use cases mentioned in Section II, code gains certain level of structure or modularity, and the development process becomes easier to maintain.

D. Shaping Circulation Realms

Communication is crucial for organizations. While this is not specific to software development, the collaborative nature of software development tends to make the importance of communication more visible. A manager cannot expect communication to be spontaneous, but rather should encourage it by creating structures that make it effortless. This is the Shaping Circulation Realms pattern. By establishing communication structures, it serves as a basis for other patterns.

Pointcut fragility is a long-known issue in aspect-oriented programming [18]. It is caused by the changes in identifier names and by the context in which they occur, such as moving a method to a different class or a class to a different package [18]. For example, this pointcut captures all the calls to methods provided by the Stock class whose name begins with calc:

```
pointcut calculations(): call(* Stock.calc*(..));
```

Calls to the methods such as calculateRevenues() or calculateDifference(). The pointcut is robust enough to capture the calls to the calculating methods regardless of their argument list. It would even survive a slight change in the naming policy such as that calculating method names begin with calc or cal. However, if this becomes compute, the pointcut would break, i.e., fail to capture what it was it intended to.

Existing strategies to decreasing pointcut fragility focus on improving the way they are expressed by, for example, using sufficiently general regular expressions rather than specific identifier names or relying on annotations to methods and fields (attributes) rather than on their names. However, much of pointcut fragility can be avoided by encouraging communication among developers, which is were Shaping Circulation Realms can help. More organized communication will help developers make more stable decisions regarding identifier names on one side and pointcut definitions on the other side, as well as being informed of inevitable changes early enough to take appropriate actions.

E. Hallway Chatter

All the people on a project can't communicate all the time, but all the people need to be informed what's happening on a project in a timely manner. This means that there must be some way of spreading news and even gossip throughout different parts of the organization, which typically reside in different parts of the building. The news should be allowed to spread in informal communication among some of the people belonging to these different parts of the organization. This is the Hallway Chatter pattern.

Aspect-oriented modularization is based on affected code being oblivious of the aspects that affect it. However, this means that the developers or teams whose code is affected by the aspects others have developed may be oblivious of them, too. Consequently, they may wrongfully interpret the changes in the program behavior and lose a lot of time until they identify the real source of the problem. This is particularly prominent with otherwise unannounced changes introduced in dire straits. Hallway Chatter can contribute to raising the awareness of what's happening even with respect to such seemingly very low level programming decisions at least by just spreading the gossip that some aspects are being prepared to get over the current situation.

VI. Related Work

Others have tried to relate organizational and development perspectives, too. Thus, Muller [19] reports that aspect-oriented programming is beneficial to agile software development iterations in terms of both time they take and code that needs to be developed. This speaks in favor of our findings how aspect-oriented programming is in line with some organizational patterns of *agile* software development. However, Muller's study is limited to AspectJ, hence ignoring symmetric aspect-oriented programming.

Parsons [20] managed to unify the aspect-oriented software development ontology published by van den Berg et al. [21] with his own agile software development ontology, which also speaks in favor of our findings. Parsons also observes difficulties in identifying the relationships between some notions.

Santosa et al. [22] report on how pair programming, which Coplien and Harrison consider to be an organizational pattern [2], is related to aspect-oriented programming related mistakes. Pair programming seems to increase mistakes in implementation logic, choice of the advice type, and code duplication, while decreasing mistakes in compilation, refactoring completeness, and excessive refactoring.

Burrows et al. [23] report a study on performing maintenance tasks with aspect-oriented programming in a pair programming constellation. The study showed certain implementation strategies to be more fault-prone than others such as specific techniques for accessing data from base code modules and binding advice to pointcuts.

Apart from Coplien and Harrison [2], throughout whose organizational patterns of agile software development resonate general software development practices, Sagenschneider [24] reports a pattern language for aligning object-oriented and general programming practices with organizing people in an office. This ranges from using comments to categorize work to using multithreading to get more tasks done.

Pícha et al. [25]–[27] propose a common model for project pattern analysis. They focus on detection of organizational antipatterns in software artifacts, such as repository commits, but not in code itself.

VII. Conclusions and Further Work

By observing Conway's law, which explains how organizing people directly affects the code they produce, and by taking into account the specifics of different programming paradigms, a lot can be done to improve the software development process. We analyzed organizational patterns of agile software development from the perspective of aspect-oriented programming, whose aim is to improve the separation of concerns in code, which, in turn, allows for better separation of tasks performed by people.

We find aspect-oriented programming to be highly related to at least nine out of more than a hundred of organizational patterns in Coplien and Harrison's catalog [2], namely: Work Split, Team per Task, Sacrifice One Person, Divide and Conquer, Conway's Law, Form Follows Function, Hallway Chatter, and Shaping Circulation Realms. Aspect-oriented programming can support organizational patterns in division of labor, treating distractions, and increasing decoupling between the software modules developed by different parts of the organization. On the other hand, organizational patterns can help mitigate pointcut fragility and aspect obliviousness in general.

Next steps should lead to the patterns that the ones we analyzed refer to, i.e., to considering pattern sequences or pattern sublanguages [28]. We assume that the most promising candidates are those patterns referred to by several other patterns proven to be related to aspect-oriented programming, such as: Organization Follows Location (5), Gate Keeper (4), Organization Follows Market (4), and Engage Customers (3). The number in parentheses indicates how many patterns out of those proven to be related to aspect-oriented programming refer to the given pattern.

The People and Code pattern language embraces some patterns that are very close to design patterns, such as Standards Linking Locations, Variation Behind Interfaces, Hierarchy of Factories, Parser Builder, Factory Method, and Loose Interfaces, whose aspect-oriented implementation should be explored for its relatedness to effective people organization.

It would be interesting to explore how are organizational patterns related to microservices as a popular architectural style, in particular for their noted uses in combination with aspect-oriented programming [29].

Acknowledgments

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/0759/19 and by the Research & Development Operational Programme Research and Innovation for the project Research of advanced methods of intelligent information processing, ITMS NFP313010T570, co-funded by the European Regional Development Fund.

References

[1] M. E. Conway, "How do committees invent?" *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[2] J. O. Coplien and N. B. Harrison, *Organizational Patterns of Agile Software Development*. Prentice-Hall, 2004.

[3] W. H. Harrison, H. L. Ossher, and P. L. Tarr, "Asymmetrically vs. symmetrically organized paradigms for software composition," IBM Research, Tech. Rep. RC22685, 2002.

[4] H. Ossher and P. Tarr, "Multi-dimensional separation of concerns and the hyperspace approach," in *Software Architectures and Component Technology*. Kluwer, 2002.

[5] J. Bálik and V. Vranić, "Symmetric aspect-orientation: Some practical consequences," in *Proceedings of NEMARA 2012: International Workshop on Next Generation Modularity Approaches for Requirements and Architecture, AOSD 2012*. Potsdam, Germany: ACM, 2012.

[6] I. Jacobson and P.-W. Ng, *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2004.

[7] I. Jacobson, "Use cases and aspects – working seamlessly together," *Journal of Object Technology*, vol. 2, no. 4, 2003.

[8] J. O. Coplien, "Borland software craftsmanship: A new look at process, quality and productivity," in *Proceedings of 5th Borland International Conference*, Orlando, FL, USA, 1994.

[9] J. O. Coplien and J. Erickson, "Examining the software development process," *Dr. Dobb's Journal of Software Tools*, vol. 19, no. 11, pp. 88–95, 1994.

[10] J. O. Coplien, "A generative development-process pattern language," in *Pattern Languages of Program Design*, J. O. Coplien and D. C. Schmidt, Eds. ACM Press/Addison-Wesley Publishing, 1995, pp. 183–237.

[11] J. Sutherland, J. O. Coplien *et al.*, *A Scrum Book: The Spirit of the Game*. The Pragmatick Bookshelf, 2019, https://sites.google.com/a/scrumplop.org/published-patterns/book-outline – http://www.scrumbook.org/.

[12] S. W. Ambler, *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press, 1998.

[13] W. J. Brown, H. W. S. M. III, and S. W. Thomas, *AntiPatterns in Project Management*. John Wiley & Sons, 2000.

[14] W. J. Brown, R. C. Malveau, H. W. M. III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

[15] P. A. Laplante and C. J. Neill, *Antipatterns: Identification, Refactoring, and Management*. Auerbach Publications, 2005.

[16] P. Brada and P. Picha, "Software process anti-patterns catalogue," in *Proceedings of the 24th European Conference on Pattern Languages of Programs, EuroPLoP '19*. Irsee, Germany: ACM, 2019, to appear.

[17] R. Miles, *AspectJ Cookbook*. O'Reilly, 2004.

[18] C. Koppen and M. Stoerzer, "PCDiff: Attacking the fragile pointcut problem," in *Proceedings of 1st European Interactive Workshop on Aspects in Software, EIWAS 2004*, Berlin, Germany, 2004.

[19] J. Muller, "What are the benefits of aspect oriented programming to project iterations developed using agile processes?" 2005, https://pdfs.semanticscholar.org/d04e/d891fb64ea396f58689137971455727e2b51.pdf.

[20] D. Parsons, "An ontology of agile aspect oriented software development," *Research Letters in the Information and Mathematical Sciences*, vol. 15, pp. 1–11, 2011.

[21] K. van den Berg, J. M. Conejero, and R. Chitchyan, "AOSD ontology 1.0 – public ontology of aspect-orientation," AOSD-Europe, Tech. Rep. IST-2-004349-NOE AOSD-Europe, 2005, https://www.researchgate.net/publication/232905610_AOSD_Ontology_10_-_Public_Ontology_of_Aspect-Orientation.

[22] A. Santosa, P. Alvesa, E. Figueiredoa, and F. Ferrari, "Avoiding code pitfalls in aspect-oriented programming," *Science of Computer Programming*, vol. 119, no. C, pp. 31–50, 2016.

[23] R. Burrows, F. Taïani, A. Garcia, and F. C. Ferrari, "Reasoning about faults in aspect-oriented programs: A metrics-based evaluation," in *Proceedings of 2011 IEEE 19th International Conference on Program Comprehension, ICPC 2011*. Kingston, ON, Canada: IEEE, 2011.

[24] D. Sagenschneider, "OfficeFloor: Using office patterns to improve software design," in *Proceedings of 18th European Conference on Pattern Languages of Programs, EuroPLoP 2013*. Irsee, Germany: ACM, 2015.

[25] P. Pícha and P. Brada, "ALM tool data usage in software process metamodeling," in *42nd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2016*. Limassol, Cyprus: IEEE, 2016.

[26] P. Picha, P. Brada, R. Ramsauer, and W. Mauerer, "Towards architect's activity detection through a common model for project pattern analysis," in *Proceedings of 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017*. Gothenburg, Sweden: IEEE, 2017.

[27] P. Picha and P. Brada, "Software process anti-pattern detection in project data," in *Proceedings of the 24th European Conference on Pattern Languages of Programs, EuroPLoP '19*. Irsee, Germany: ACM, 2019, to appear.

[28] W. Sulaiman Khail and V. Vranić, "Treating pattern sublanguages as patterns with an application to organizational patterns," in *Proceedings of 22nd European Conference on Pattern Languages of Programs, EuroPLoP '17*. Irsee, Germany: ACM, 2017.

[29] T. Cerny, "Aspect-oriented challenges in system integration with microservices, SOA and IoT," *Enterprise Information Systems*, vol. 13, no. 4, pp. 467–489, 2018.