

# Lightweight Aspect-Oriented Software Product Lines with Automated Product Derivation

Jakub Perdek<sup>[0009–0003–3616–4373]</sup> and Valentino Vranić<sup>[0000–0001–9044–4593]</sup>

Institute of Informatics, Information Systems and Software Engineering  
Faculty of Informatics and Information Technologies  
Slovak University of Technology in Bratislava  
`xperdek@stuba.sk`, `vranic@stuba.sk`

**Abstract.** Aspect-oriented software product lines are not a new idea, but their application is facing two obstacles: establishing software product lines is challenging and aspect-oriented programming is not that widely accepted. In this paper, we address exactly these two obstacles by an approach to establishing lightweight aspect-oriented software product lines with automated product derivation. This is particularly relevant for data preprocessing systems, which are typically custom-built with respect to the data and its structure. They may involve data cleaning, reduction, profiling, validation, etc., which may have variant implementations and may be composed in different settings. The approach is simple and accessible because developers decide about variation points directly in the code without any assumption on development process and applied management. Also, it allows for variability management by making the code readable, configurable, and adaptable mainly to scripts and code fragments in a modular and concise way. The use of annotations helps preserve feature models in code. We presented the approach on the battleship game and data preprocessing pipeline product lines, which include the configuration of features, product derivation mechanism based on annotations applied by the user on certain classes and methods, implementation of features according to the feature model, and the possibility to generate all given software product derivations.

## 1 Introduction

Software product lines (SPL) are an efficient approach to software reuse. The secret of their success lies in limiting the effort for reuse to a set of related software systems within one organization (sometimes denoted as a family). Such systems have a lot of common features, but they also have some variable features. These have to be implemented in such a way that would enable them to vary. Consequently, a design of a software product line starts by mapping the features it should cover. This requires some kind of feature modeling to be employed, which is not necessarily the academic FODA-like notation [15,3].

It is obvious that variable features are best implemented in a pluggable fashion, but this is not easy to achieve in traditional object-oriented programming

because their implementation tends to crosscut many other features [25]. Aspect-oriented programming addresses exactly the issue of crosscutting concerns. The best known aspect-oriented programming language is AspectJ, which is an extension to Java, but many other programming languages exhibit aspect-oriented features, e.g., Python, JavaScript, etc. [9,10].

Aspect-oriented software product lines are not a new idea, but their potential application is facing two obstacles: establishing software product lines is challenging and aspect-oriented programming is not that widely accepted. In this paper, we address exactly these two obstacles by an approach to establishing lightweight aspect-oriented software product lines with automated product derivation. This is particularly relevant for data preprocessing systems, which are typically custom-built with respect to the data and its structure. They may involve data cleaning, reduction, profiling, validation, etc., which may have variant implementations and may be composed in different settings.

The rest of the paper is organized as follows. Section 2 explains the position of aspect-oriented programming in software product lines. Section 3 provides an overview of establishing a software product line for a battleship game. Section 4 introduces annotations and expressions which help to choose content to copy into the resulting project presented on the game derivation based on actual configuration settings. Section 5 presents evaluation and discussion. Section 6 compares the approach proposed in this paper to related work. Section 7 concludes the paper.

## 2 Software Product Lines and Aspect-Oriented Modularization

Variability is the main part of software product development and products are built by resolving it in a way that can build customer specific products [8]. Software products often emerge from the success of the market with different needs that can be provided by actual knowledge determined by features, relationships among them and between them, and software artifacts that provide the implementation of these features. These known actual knowledge sources are used for systematic reuse introduced by software product line engineering [22]. Evolving products then depend on their real-world applications which should provide a flexible way how to apply constantly changing needs. Given features may affect several places in models and code which can cause problems during its adaptations with other already included features. These places where changes occur [14] are called variation points and represent possible ways how to model variability. Interaction of features can cause a need to generate tailored software artifacts or software artifacts for their next modification for the final application. It is easier to configure the process by applying a change to code at the places where components are generated [25].

Aspect-oriented model driven software product line development models variability on model level and aspects to implement these models, primarily their crosscutting features [23]. Models provide more abstract views of features to be

separated and be more effectively managed such as traced according to customer requirements, in comparison to their configuration on code level [23].

The main problems of AspectJ are the necessity to maintain certain conventions about names of methods and classes [17], as well as to divide the application into appropriate parts for changing their behavior and introduce hook methods only to hang up aspect on them [16]. Other issues can arise with certain types of applications. For example, during the refactoring of the database system, there was necessary to use privileged aspects to access non-public variables, but this violates encapsulation [16]. Aspect behavior can modify and use these variables. Aspects often require mentioned supervision on the design of the final solution otherwise the solution will be less maintainable.

### 3 Establishing a Software Product Line

Our approach assumes that an initial software product exists and that it exhibits good object-oriented modularization. We rely on this, so that we can introduce variable features using aspect-oriented programming. After this step aspects implemented by AspectJ are inseparable parts of the resulting products and product line code. We map both common and variable features by a feature model, an example of which can be found in the next section.

We will explain our approach on an implementation of the battleship game product line. We adopted and adapted the basic game from a publicly available resource.<sup>1</sup> We refactored it significantly to improve its object-oriented modularization.

After changing the visibility of variables, adding a configuration file for the base state of an application, and moving appropriate content to newly created classes, we needed to design configurable features, for which we relied on known aspect-oriented practices [16,8]. We did this using AspectJ, a relatively stable aspect-oriented extension of Java.<sup>2</sup>

We based variable features on AspectJ because this way we can additively integrate concerns where each is implemented as a separate aspect. Thanks to presented modularity the solution is more extendable. For example, the feature for setting player names requires adding another method before creating the functionality to set names to players. Without aspects, only condition statements will be used instead. There is also no possibility to add a player name variable to the Player class only for this case. But, all required functionality such as new variables and methods can be added into segregated aspects. It is also possible to configure or exclude them according to the configuration which can be loaded when the application starts or changing values at runtime. By using aspects, we only need to specify a pointcut that provides the mapping of the certain location where players are created to the executed method. In this aspect method, the program loads names which are typed by players from the input. The whole functionality is in one aspect.

<sup>1</sup> <https://github.com/juletx/BattleshipFeatureIDE>

<sup>2</sup> <https://www.eclipse.org/aspectj/>

We found that aspects well suited the configuration of settings in the solution by directly putting conditions from the configuration to manage the optional inclusion of given concerns. Different implementations of the same aspect can be used to apply specific configurations reflecting the customer’s needs. The configuration is applied only by replacing the parameters of overloaded methods with the loaded values from the configuration file before calling these methods.

## 4 Product Derivation

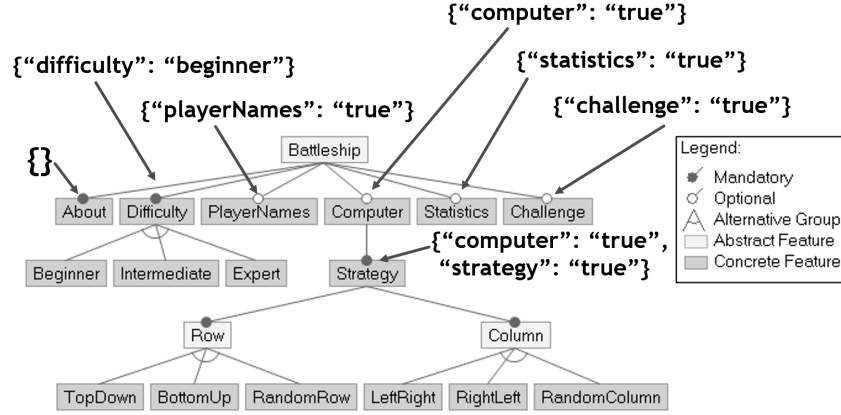
Since aspects in AspectJ become active simply if they are included in the compilation, the product derivation in our approach is realized by simply copying the corresponding aspects to the project folder. For non-aspect features, the classes that implement them are copied, of course. This process is managed by annotating certain code parts and specifying the condition inside expressions that are inserted into them. Feature inclusion must follow the constraints set by the feature model in terms of variability if the modeling of the variability is based on feature models. Furthermore, a feature can be included only if its parent is included, even if it is mandatory. Other models can be incorporated accordingly by changing annotations or variables to meet given restrictions.

### 4.1 Annotating Variation Points

The comprehensive and readable rules which will contain the mapping for the features are needed. We come up with the idea of creating annotations with expressions that allow specifying a condition when a feature should be copied and when not. Each annotation starts with the comment characters (`//`) followed by an identifier for the given annotation type. The last part consists of an expression in JSON format. It contains variables with assigned values, especially operators as reserved ones (AND or OR) to evaluate the truthfulness of grouped variables in a certain way.

Mapping of some variables with given values to the feature model can be seen in Figure 1. Each rule is prescribed and handled by a given annotation type and thus changing its purpose needs to be applied in the derivator. Derivation happens in design time by evaluating all these expressions inside the project folder. The result of this operation has an impact on whether this prescribed functionality will be applied or not. For example, if the variable named `playerNames` mapped to the feature with the same name is set to true and all expressions where its included are evaluated positively, then all necessary (annotated) code parts for this feature will be copied in the final solution. If an expression in the rule is empty, then it is evaluated as true. If a file contains no annotations or all rules have their annotations evaluated as false, then the file will not be copied into the final solution.

In order to copy only modular code fragments and marginally their crosscutting functionality, these annotations are used:



**Fig. 1.** The battleship game feature model with derivation rules.

- //@{} This annotation type is used to annotate classes, interfaces, or aspects to copy them along with their source file. An additional check can be added to the program that one of the keywords related to the used programming language is located after annotation.
- //#{ } This annotation type is suitable to include or exclude given methods. If the file should be copied, then at least one condition in the annotation should evaluate to true. This annotation type should not be mixed with the first one because nested annotations are not supported.
- //%{ } By using this annotation type, some import statements (each represented as one line of code) are included or excluded according to including or excluding a given method from the file. Consequently, it should be used restrictively with the second annotation.

Making a product ready is only a matter of splitting features into modular parts and making clones of selected code parts according to the product specification. Effective decomposition into separate files allows copying whole files rather than code fragments. The aspects woven into the code are essential here. If fragmentation would cause these modular parts to lose their main responsibilities, then the latter two annotations are used. The inner content then can be managed effectively. Annotated methods can make other dependencies by importing classes that will not be part of the final product. Their import statements are, thus, managed similarly. The third annotation type solves these problems by including or excluding only one line of code.

A typical example is the functionality for setting names (the PlayerNames feature) which is affecting the Computer and Player classes displayed in the feature model in Figure 1. Methods that are setting names are implemented in a separate aspect and are annotated with the second annotation. Both of them contain the playerNames variable in their expressions. Except for the first annotation, the second one contains also the computerPlayer variable which prevents

problems if the functionality which simulates an opponent (the Computer feature) will be unavailable (not copied) for a given variant. In this case, the import statements of this functionality are omitted by using the third annotation type.

For making more complex conditions, logical *and* and *or* can be inserted into each other to represent hierarchical dependencies between related features in the feature model. Figure 2 shows an example of this.

```
{
  "AND": {
    "OR": {
      "variable1": "false",
      "AND": {
        "variable2": "true",
        "variable3": "true"
      }
    },
    "variable4": "true"
  }
}
```

**Fig. 2.** A complex derivation rule.

## 4.2 Product Derivation

The product derivation can start by launching the derivator after variable code parts are annotated according to the proposed mapping of the variability model, especially the feature model. This happens in design time by configuring these mapped values. Each one is set to the one that fits the resulting product in this step. No other action is needed from developers, except evaluating errors caused by improperly chosen or configured expressions in place of variation points.

Figure 3 shows the classes involved in product derivation. The `DerivationManager` class manages product derivation. The `ProjectCopier` class is used to copy an empty project and the `FileCopy` class manages the reading and writing of a given file as a stream. During copying, it is necessary to determine whether a file should be copied or not. For this purpose, the program needs to find annotations and evaluate the expression they contain, which is performed by the `DerivationAnnotationManager` and corresponding annotation specific classes. This search is repeated after any annotation is found until the end of the file. The `DerivationVariableProcessor` class helps the `DerivationAnnotationManager` class to recursively evaluate expressions associated with the annotation.

The derivator is available and open for further customization.<sup>3</sup>

<sup>3</sup> <https://github.com/jperdek/productLineAnalysisGame>

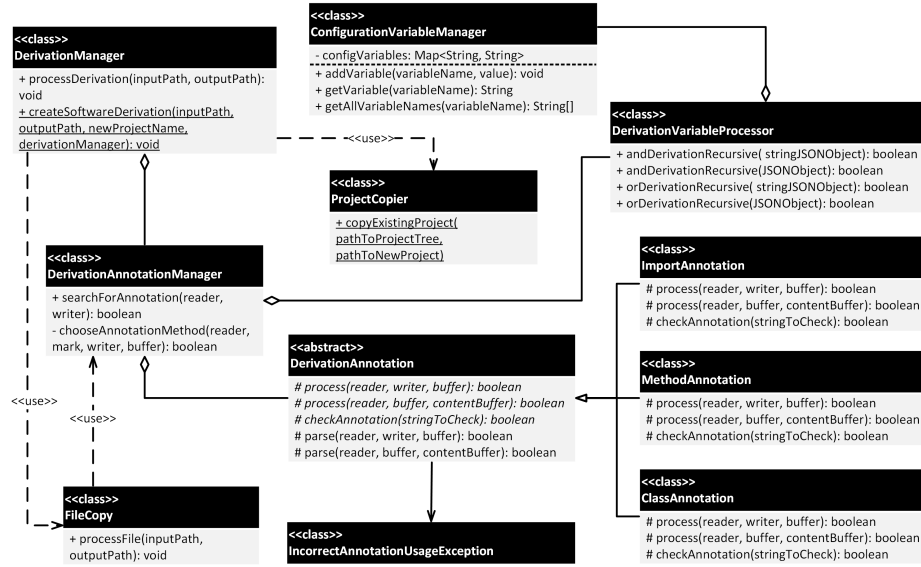


Fig. 3. Product derivator

## 5 Discussion

Aspects help to move the logic of crosscutting concerns into modular units and enhance functionality without modification of existing code. Well-modularized crosscutting concerns can be easily reused in the form of modules such as in HealthWatcher [11]. Typical examples are large parts of interacting code for logging, exception handling, and third-party functionality which is moved to aspects. According to these observations, we designed a mechanism to easily copy certain parts of a given project and let the project be functional and possibly prepared for the next development phase.

Applying the functionality of AspectJ to given join points can be restricted by adding conditions to their pointcuts. Whole functionality is thus available in resulting products. Many of these conditions remain redundant, except in the case when they need to be selected and changed dynamically at runtime. Our annotations are language-independent and are removed from all derived products, thus they are selected before compilation in design time. Also, if features affect each other, then modularity cannot be preserved with aspects without decomposing this functionality further.

Appropriate domain analysis is required to design common and variable features of a given system in order to understand which products are possible to generate. For this purpose, we used feature models. In reality, many of these models can exist, but only a few of them support the changing needs of the customers [4]. We intended to find a mechanism to preserve feature models in code and make the integration process easier by letting developers manage it

directly in an easily readable way as a response to the main problems observed by Bischoff et al. [5].

Sometimes, features can collide and some features can't be used together. Often, adding another concern to a separate layer can fix this issue [6]. But in case of serious restrictions, it's necessary to think about the derivation of a given product and develop certain features without those colliding ones.

Our approach can be applied for the both revolutionary and evolutionary development of software product lines [7]. Although we assume an initial product to exist, this is not inevitable as the initial product can be developed by establishing the product line. It suits more in the case of already applied domain analysis to construct feature models. According to it, we should be able to explicitly formulate an expression for each feature whether it should be included or not.

In the revolutionary development of software product lines, it is necessary to develop modular applications with all their features. All products can be derived after annotating given code fragments with direct mapping to features in the form of introduced expressions.

The evolutionary development of software product lines requires constructing modules for solutions incrementally. After their creation, these modules are annotated, and included expressions are properly configured to manage their integration. In the end, the derivation mechanism omits not annotated code fragments.

## 6 Related Work

The most similar to our approach is the frame technology, which is also capable of creating easily adaptable components with a language-independent mechanism [18]. However, the code is mixed with tags and it is not possible to compile it. The tag hierarchy can be complex and the tag endings in it can be difficult to pair. On the contrary, expressions in our approach can be recursively evaluated. Unlike the tags in the frame technology, they are in the form of annotations adaptive preserve feature models directly in the place where they occur in a much more modular way.

Another similar work [12] introduced mapping between given models based on a template model as a representation of each available product. Our expressions are not only presence conditions, but rather, as JSON representations, provide space to include more information for further configuration related to domain requirements. Not only removing unwanted features from the feature model but also adding new functionality apart from it, is then possible. This makes our approach not restricted to negative variability. Another significant difference is in the hierarchic organization of variables under used operators which is more comprehensive for the hierarchic structure of feature models. Additionally, the mechanism for evaluating expressions can be configured according to the expected key and value pair.



Plastic partial components were introduced to manage variability inside components [19]. They are just component fragments that are mainly reused by applying invasive composition techniques [1] when aspects are used to manage their selection in the final product. In our approach, annotated parts can evolve together as an already instantiated product or as a derived application on its own.

The variability on the code level can be also handled by `pure::variants` [20], which provides also many tools, especially for tracing. In our approach, expressions have the same role as the configuration rules, but differ with used JSON format and also are recursively evaluated to directly adapt to the nature of feature models in code. The used format allows processing them by using distributed computing. Also, other functionality relevant to other roles associated with annotated variation points can be configured there. In `pure::variants`, endings are used, similar to tag endings in the frame technology or conditional compilation. Annotated code is thus less readable. Opposite to this, we want to force developers to annotate only aspects and classes as the primary types of modules, and methods, if necessary.

In our approach, developers make all decisions about variability directly in code, but these are in `pure::variants` configured and visualized by plugins and tools. Their configuration through used tools is restricted to programmed functions, which do not allow immediately adding developers' own variables with special semantics for changed functionality of the derivator. Such cases can be optional functionality to pretest variants by creating additional logs, letting variants generate data for analysis, adding integration with their tool, or adapting certain business functionality related to the variation point without directly polluting the code.

The key activities for the individual product derivation, such as product configuration, requirements engineering, additional development, integration, deployment, and, finally, product line evolution [21], in our approach are managed not much differently than in the most used derivation approaches such as DOPLER or Pro-PD. Mostly, developers introduce variation points that are configured mainly according to the feature model which is discussed with the customer or chosen for the given domain. As in DOPLER and Pro-PD, the customer requirements are fulfilled iteratively also in our approach. The collaboration with the customer depends then on the project management methodology being applied, but even more on variability modeling. The focus on product configuration should minimize product specific development as in Pro-PD. This benefit is not similar to DOPLER which is more collaborative than our approach and Pro-PD. This is recognizable in situations when unsatisfied user requirements are perceived as product specific implementations [21].

In our approach, it is not necessary to create a specific language for product derivation as it is in FAST [2]. It involves less planning for the product derivation. Activities, products, and roles are not described as in PuLSE-I [26]. The development process is not limited to iterative fulfilling of requirements as in COVAMOF [13].

## 7 Conclusions and Future Work

In this paper, we introduced an approach to establishing lightweight aspect-oriented software product lines with automated product derivation which is simple and accessible because developers decide about variation points directly in the code without any assumption on the development process and applied management. Also, it allows for variability management by making the code readable, configurable, and adaptable mainly to scripts and code fragments in a modular and concise way. The use of annotations helps preserve feature models in code through their ability to store information about the hierarchy of features.

The information extracted from variation points in the form of JSON format can be processed by known big data systems, such as Hadoop, Pig, or Hive to analyze dependencies and relations between used variables.

We presented the approach on the battleship game and data preprocessing product lines. These examples include the configuration of features, product derivation mechanism based on annotations applied by the user on certain classes and methods, implementation of features according to the feature model, and the possibility to generate all given software product derivations. We need only three types of basic annotations to mark the variety of code fragments for the next separation. We generated all 48 possible derivations and tested their functionality. It's also possible to run one of the supported product instances directly using the base project by setting configuration values in the configuration file. Developers are also allowed to change provided simple derivator functionality to promptly handle demands on data processing.

The approach proposed in this paper can be applied to efficiently handle the variability associated with the customization of data preprocessing mechanisms, derivation of adjusted products which are ready for their next validation, or processing expressions that contain knowledge about variation points by using big data analysis techniques.

We intend to explore this more thoroughly in our future work mainly focused on the ways how to create, process, and evaluate data that contains a lot of variability. Fractals are a good source of it. We will effectively derive and validate a variety of them according to metrics that support the aesthetic perception of users or use other assumptions. The knowledge from variation points will be used during the prediction phase. A small modification of the derivation mechanism is necessary to adapt it to web languages, mainly those where the conditional compilation is unavailable, such as JavaScript or TypeScript.

Derived products from the introduced product line can serve as the basis for another one to implement contradictory features efficiently. They can be quickly improved to support the extraction of such features by introducing or changing the annotations.

We are also exploring how our approach could be applied to manage the diversity of services featured in a research setting of developing rurAllure,<sup>4</sup> an innovative pilgrimage support system [24].

---

<sup>4</sup> <https://rurallure.eu/>

**Acknowledgements** The work reported here was supported by from the European Union’s Horizon 2020 research and innovation program under grant agreement No. 101004887 (rurAllure), the Operational Program Integrated Infrastructure for the project: Support of Research Activities of Excellence Laboratories STU in Bratislava, project no. 313021BXZ1, co-financed by the European Regional Development Fund (ERDF), and the Operational Programme Integrated Infrastructure for the project: Research in the SANET network and possibilities of its further use and development (ITMS code: 313011W988), co-funded by the ERDF, and by the Slovak Research and Development Agency under the contract No. APVV-15-0508.

## References

- [1] Aßmann, U.: Invasive Software Composition. Springer-Verlag, Berlin, Heidelberg (2003)
- [2] Bayer, J., Gacek, C., Muthig, D., Widen, T.: PuLSE-I: Deriving instances from a product line infrastructure. In: Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, ECBS 2000 (2000)
- [3] Berger, T., Rublack, R., Nair, D., Atlee, J.M., Becker, M., Czarnecki, K., Wąsowski, A.: A survey of variability modeling in industrial practice. In: Proceedings of 7th International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS ’13. ACM, Pisa, Italy (2013)
- [4] Beuche, D., Dalgarno, M.: Software product line engineering with feature models. <https://www.pure-systems.com/fileadmin/downloads/pure-variants/tutorials/SPLWithFeatureModelling.pdf> (2006)
- [5] Bischoff, V., Farias, K., Gonçalves, L.J., Victória Barbosa, J.L.: Integration of feature models: A systematic mapping study. *Information and Software Technology* **105**, 209–225 (2019)
- [6] Blair, L., Pang, J.: Aspect-oriented solutions to feature interaction concerns using AspectJ (2003)
- [7] Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley (2000)
- [8] Botterweck, G., Lee, K., Thiel, S.: Automating product derivation in software product line engineering. In: Proceedings of Software Engineering 2009. LNI P-143, Gesellschaft für Informatik e.V. (2009)
- [9] Bystrický, M., Vranić, V.: Preserving use case flows in source code. In: Proceedings of 4th Eastern European Regional Conference on the Engineering of Computer Based Systems, ECBS-EERC 2015. IEEE, Brno, Czech Republic (2015)
- [10] Bálik, J., Vranić, V.: Symmetric aspect-orientation: Some practical consequences. In: Proceedings of Proceedings of International Workshop on Next Generation Modularity Approaches for Requirements and Architecture, NE-MARA 2012, at AOSD 2012. ACM, Potsdam, Germany (2012)
- [11] Cherait, H., Bounour, N.: History-based approach for detecting modularity defects in aspect oriented software. *Informatica* **39**(2), 187–194 (2015)

- [12] Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: *Generative Programming and Component Engineering*. pp. 422–437. Springer, Berlin, Heidelberg (2005)
- [13] Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: A case study. *Journal of Systems and Software* **74**(2), 173–194 (2005)
- [14] Jacobson, I., Griss, M., Jonsson, P.: *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley (1997)
- [15] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA): A feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA (1990)
- [16] Kastner, C., Apel, S., Batory, D.: A case study implementing features using AspectJ. In: *Proceedings of 11th International Software Product Line Conference, SPLC 2007*. IEEE, Kyoto, Japan (2007)
- [17] Laddad, R.: *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning (2003)
- [18] Loughran, N., Rashid, A.: Framed aspects: Supporting variability and configurability for AOP. In: *Proceedings of 8th International Conference on Software Reuse, ICSR 2004*. LCNS 3107, Springer, Madrid, Spain (2004)
- [19] Perez, J., Diaz, J., Costa-Soria, C., Garbajosa, J.: Plastic partial components: A solution to support variability in architectural components. In: *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*. IEEE, Cambridge, UK (2009)
- [20] pure::systems: PLE & code—managing variability in source code. <https://youtu.be/R1UYjWhJFkM> (2020)
- [21] Rabiser, R., O’Leary, P., Richardson, I.: Key activities for product derivation in software product lines. *Journal of Systems and Software* **84**(2), 285–300 (2011)
- [22] Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., Bettin, J. (eds.): *Domain Engineering, Product Lines, Languages, and Conceptual Models*. Springer (2013)
- [23] Voelter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: *11th International Software Product Line Conference (SPLC 2007)*. IEEE, Kyoto, Japan (2007)
- [24] Vranić, V., Lang, J., López Nores, M., Pazos Arias, J.J., Solano, J., Laseca, G.: Use case modeling in a research setting of developing an innovative pilgrimage support system. *Universal Access in the Information Society* (2023), accepted.
- [25] Vranić, V., Táborský, R.: Features as transformations: A generative approach to software development. *Computer Science and Information Systems (ComSIS)* **13**(3), 759–778 (2016)
- [26] Weiss, D.M., Lai, C.T.R.: *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley (1999)