ICIC Express Letters
Volume **18**, Number **3**, March **2024**
 ISSN 1881-803X
pp. **293–301**

A PROPOSED JAVA WEB FRAMEWORK TO SUPPORT SOFTWARE PRODUCT LINE ENGINEERING

Maya Retno Ayu Setyautami*, Daya Adianto, Ade Azurat
and Eko Kuswardono Budiardjo

Faculty of Computer Science
Universitas Indonesia
Kampus UI, Depok, Jawa Barat 16424, Indonesia
{ dayaadianto; ade; eko }@cs.ui.ac.id
*Corresponding author: mayaretno@ui.ac.id

Received April 2023; accepted July 2023

Abstract. *Software product line engineering (SPLE) offers a valuable approach for reusing software components and efficiently developing diverse products within a specific domain. In the context of web development, SPLE can significantly enhance variability management and promote systematic reusability. This study proposes a framework for web application development based on SPLE principles. The web framework is designed based on variability modules for Java (VMJ), an architectural pattern supporting delta-oriented programming in Java. Core functionalities are implemented within the core Java module, while variabilities are accommodated in the delta Java module. The development process is facilitated by a low-code tool, enabling the creation of Java web services for the web application's backend. To demonstrate the practicality of the proposed approach, a case study involving the development of an adaptive information system for charity organizations is presented. The VMJ web framework proves its efficacy by generating a fully functional web application tailored to the specific requirements of charity organizations. In conclusion, this research contributes to the advancement of SPLE methodologies based on delta-oriented programming (DOP) by introducing a Java web framework. The framework offers enhanced variability management capabilities and facilitates efficient web application development.*
Keywords: Code generator, Model transformation, Software product line engineering, Variability modules, Web framework

1. **Introduction.** Web applications are a standard form of media communication in many domains. A generic web application for a similar domain can be developed based on a typical business process. Customization for specific requirements is usually managed using a *clone-and-own* approach. An existing product is cloned into a new repository to develop another product variant. A modification is created in the new repository if a product requires specific feature changes. Although the repositories use the same initial source code, modifications are scattered across them. Therefore, requirements changes are difficult to trace, and variability cannot be managed systematically.

Software product line engineering (SPLE) is an approach for developing various applications based on commonality and variability [1, 2]. Common features are implemented as reusable components, while variants are designed to support mass customization. The main motivations for using SPLE are reduced development costs, enhanced quality, and reduced time to market [2]. Various products are designed together as a software product line (SPL), and a product variant is generated based on the requirements of a specific customer.

DOI: 10.24507/icicel.18.03.293

The main motivations of SPLE are shared along with the low-code application platform (LCAP) that gains momentum in recent years. LCAP is a platform that lets developers rapidly develop and deploy custom applications by reducing the amount of coding work [3]. It employs similar approaches found in SPLE, such as model-driven engineering (MDE) and domain modeling, to produce applications [4]. This research aims to apply SPLE and low-code approaches to developing web applications. Various web applications in the same domain can be designed as a web-based SPL. First, commonality and variability of a web-based SPL are analyzed in domain engineering. Then, a variant of a web-based SPL can be generated based on selected features. Applying SPLE drives systematic reusability in web development, and the low-code approach supports automated code generation. Combining SPLE and low-code could produce higher-quality web applications in less time and at a lower cost.

In this research, we design a VMJ web framework based on variability modules for Java (VMJ). VMJ is an architectural pattern that uses Java module systems and design patterns [5]. The development process in the VMJ web framework follows the principles of SPLE, starting with domain analysis and modeling a web-based SPL using a feature model and the UML diagram. Other frameworks for building a web-based SPL use various modeling approaches, such as the common variability language (CVL) [6], a feature model [7, 8, 9], and a combination of UML and feature model [9, 10].

Mostly web frameworks for SPLE provide tools to support model transformations or product configuration, without generating running applications. The process of generating a web application is only explained by [6, 7, 9, 10]. In this work, the VMJ web framework is also integrated with FeatureIDE [11, 12]. FeatureIDE is an Eclipse-based Integrated Development Environment (IDE) that supports feature-oriented software development. Based on the feature selection in FeatureIDE, a back-end of web applications is generated as Java web services, which are produced from the VMJ web source code. The web front-end is a JavaScript application generated from interaction flow modeling language (IFML). The contribution in this paper is the VMJ web framework, so our explanation is more focused on the domain implementation using VMJ web.

The remainder of this paper is organized as follows. In Section 2, we explain variability modules for Java (VMJ), as a theoretical foundation in this research. Section 3 presents the structure of the VMJ web framework to develop a web-based SPL. The applicability of the VMJ web framework is shown in Section 4 using a case study. In Section 5, we evaluate the development process of the VMJ web framework. Section 6 explains the related work, and Section 7 offers conclusions and recommendations for future work.

2. **Variability Modules for Java.**

Variability modules for Java (VMJ) is designed based on variability modules (VM). VM concept extends the capability of DOP to realize a multi-product line (MPL) [13]. An MPL consists of several product lines that share dependencies [14]. VMJ uses the Java module system and design patterns to implement an SPL [5]. The decorator pattern is used to model variations, and the factory pattern is used to create appropriate object variants. A product line in VMJ is represented as a Java project consisting of Java modules.

The decorator pattern is applied in a delta module that modifies a core module. A delta module decorates a core module by specifying the modification behavior. A product module configures objects based on the selected features. A product module can access functionality in the selected features by defining Java module dependency. Objects in the core and the delta modules are created using the factory pattern. The factory pattern also manages the delta application order if more than one delta modules implement a feature.

Based on the VMJ architectural pattern, a Java module in VMJ is categorized as follows.

- The *core module* is defined as a Java module that consists of packages with common capabilities in an SPL. A core module can be reused by delta modules or product modules. Following the decorator pattern, a core VMJ module consists of interfaces, abstract component classes, concrete component classes, and an abstract decorator class.
- The *delta module* is defined as a Java module that modifies a core module. A delta module consists of a concrete decorator class that changes the concrete component class in the core module.
- The *product* module is also defined as a Java module that has a dependency on core modules and delta modules. The dependency is defined based on the required features in the product variant.

Each VMJ module has a module declaration, file (`module-info.java`), specifying its name and dependencies. Dependencies are managed using *export* and *import* mechanisms in Java module systems. The *export* term is used to declare which packages are visible to other modules, while the *import* term is used to declare other required modules. Packages in the core module must be exported in the module declaration to be reused in other modules. Then, a delta module imports related core modules to reuse or modify the implementation in the core module.

The factory design pattern supports a product specification in the product generation process. A class in the core module can be modified by one or more delta modules using the decorator pattern. As a result, the same class can be reused in different delta modules. A fully qualified name (FQN) is used to distinguish the same class in different variants. The factory pattern is used to create an object based on its FQN. Therefore, we can choose a specific variant from selected features during the product generation.

3. **VMJ Web Framework.** The VMJ web framework aims to support web-based SPL development based on DOP [15] and VMJ architectural pattern [5]. As defined in VMJ, the VMJ web framework uses Java module systems and design patterns. The architecture of the VMJ web framework is separated into several layers to distinguish the development concerns. Figure 1 shows the structure of the VMJ web framework. The framework contains three layers, *presentation*, *domain*, and *persistence*. The VMJ web libraries are also developed to support web development in those layers.

- The *presentation layer* handles HTTP requests from clients and returns the response in JSON format. Two libraries in this layer, `VMJRouting` and `VMJAuth`, are developed using Java modules to support the functionalities in the presentation layer. The `VMJRouting` library consists of the `VMJServer` to manage server initiation and activation, `VMJExchange` to receive and manage data from the HTTP protocol, and `Route` and `Router` to handle endpoint configuration. Authentication and authorization in the VMJ web framework are managed by the `VMJAuth` library. Therefore, developers can grant or restrict access to each service individually.
- The *domain layer* contains an implementation of business logic and domain model. The development of the domain layer conforms to the VMJ structure, such as core, delta, and product modules. The core and delta modules are separated into *model* and *resource layers*. The core module implements the commonality, and the delta module modifies the core module to implement the variability. The product modules are generated based on selected features.
- The *persistence layer* utilizes HibernateORM to manage database connectivity and operations. `VMJHibernate Integrator` library is developed to bridge database access from the VMJ web framework. HibernateORM maps VMJ web classes into relational databases and manages CRUD (create, read, update, and delete) operations. The generated database schema is adjusted based on the feature's selection. Therefore, CRUD operations are only available for selected features.

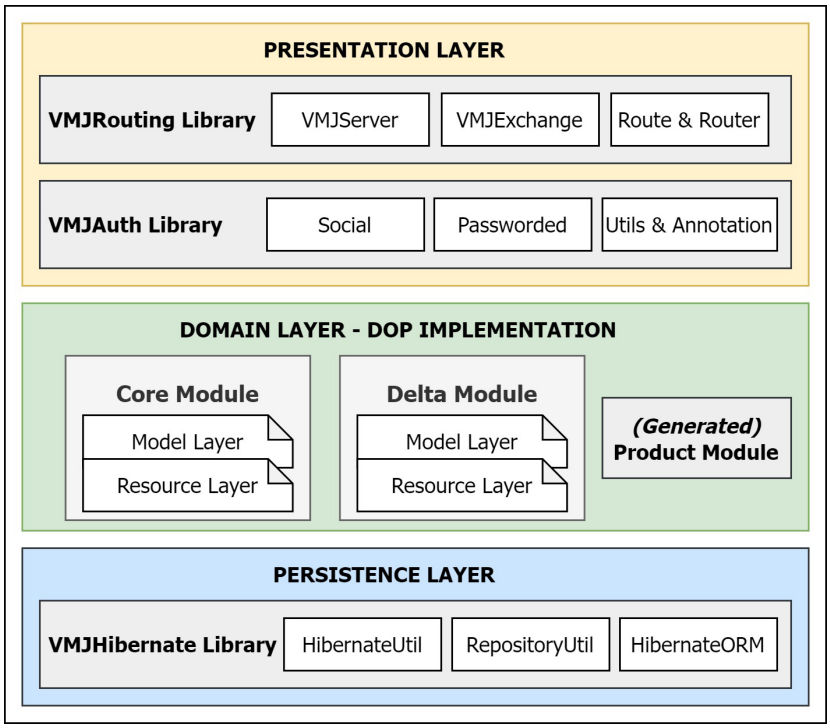

FIGURE 1. The structure of the VMJ web framework

As shown in Figure 1, the domain layer is a part of SPLE implementation. The other layers can be seen as supporting libraries for web development. The workflow of the VMJ web framework is started when a client sends HTTP requests using URLs or available endpoints. The URL represents a method in the *domain (resource) layer*. The `VMJRouting` library processes the request and calls the related method in the *resource layer*. Then, the *resource layer* calls *repository* in the persistence layer if the method needs database operations. *Repository* uses HibernateORM library and then, HibernateORM connects to the database and asks for the required data.

A tool support for web-based product line development with VMJ web framework is designed in FeatureIDE. FeatureIDE is an integrated framework to support feature-oriented development of software product lines [12]. It is designed based on Eclipse IDE as an Eclipse plugin. It has several composers that manage the domain implementation and product generation process. We can choose the composer based on the domain implementation preference, such as AHEAD, Munge, AspectJ, FeatureHouse, and FeatureC++. In this research, we add a new composer to FeatureIDE to support domain implementation with VMJ web, namely VMJ web composer.

The development of VMJ web composer in FeatureIDE aims to integrate the feature model and the domain implementation. First, we design the feature diagram, a graphical notation to specify a feature model, in FeatureIDE. The feature diagram represents commonality and variability in the product line. In the VMJ web framework, a commonality is implemented in a core module, and variability is implemented in delta modules. The mapping between features in the feature diagram and their implementation in the Java modules is defined in a JSON file.

The product generation is performed based on feature selection, which is defined in a configuration file in FeatureIDE. A configuration file consists of selected features from the feature diagram. We can have many product variants in the product line, and each variant is defined as a configuration file. To generate a specific product, we can select a configuration as a *current configuration*. Then, related modules are composed, and a valid product module is automatically generated. The product can be compiled and run using the VMJ web composer in FeatureIDE.

4. **Running Example.** In this section, we use an adaptive charity organization system called AMANAH as a running example to show the implementation of a back-end of web applications using the VMJ web framework. A snippet of the AMANAH feature diagram is shown in Figure 2. The feature diagram represents commonality and variability in charity organization systems. As shown in the diagram, *Program* and *Income* are mandatory features. Other features, such as *Expense*, *ArusKasReport* and *Confirmation*, are optional. In VMJ web, the common features are implemented in the core Java module, and the variabilities are implemented in the delta Java modules using the decorator pattern.

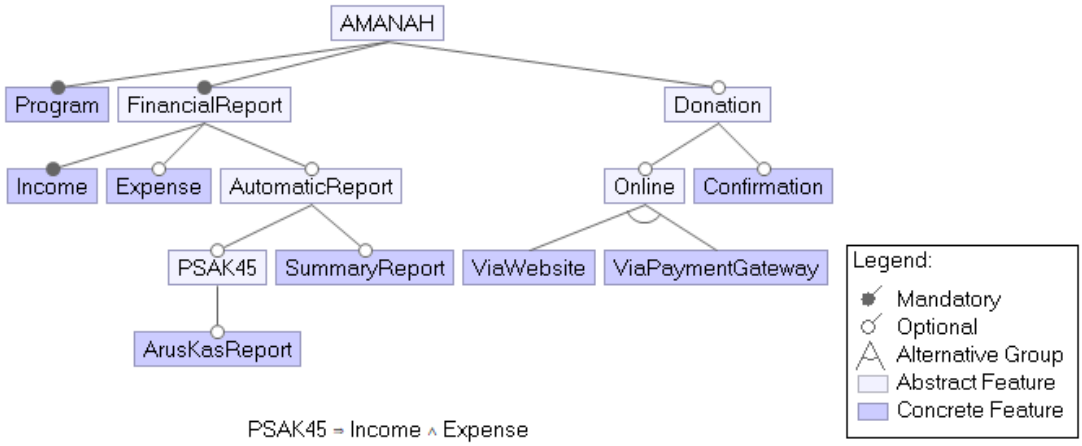

FIGURE 2. AMANAH feature diagram

For example, in *FinancialReport* feature in Figure 2, three Java modules are developed in the *domain (model) layer*: a core module *FinancialReport*, delta modules *Income* and *Expense*. Following the decorator pattern, the core module consists of *interface*, *abstract component class*, *concrete component class*, and *abstract decorator class*. The delta module consists of the *concrete decorator class*. The UML-VMJ generator [16] generates classes in the model layer of VMJ web, which consists of fields and *getter setter* methods.

The resource layer in the VMJ web framework should be completed by developers. This layer contains an implementation of business logic in web applications. Listing 1 shows a code snippet of method `saveFinancialReport()` in the *FinancialReport* core module. The resource layer utilizes `RepositoryUtil` in `VMJHibernateIntegrator` library to access the persistence layer (database). A method call requiring database access is shown in Line 10 that will save *financial report* data into the database. The available endpoints (*routing*) are also configured manually in the resource layer. The endpoint URL is defined

```
1  package amanah.financialreport.core;
2  ..
3  public class FinancialReportResourceImpl extends FinancialReportResourceComponent {
4          ..
5          @Restricted(permissionName="ModifyFinancialReportImpl")
6          @Route(url="call/financialreport/save")
7          public List<HashMap<String,Object>> saveFinancialReport(VMJExchange record) {
8                  ....
9                  FinancialReport data = createFinancialReport(record);
10                 frRepository.saveObject(data);
11                 return getAllFinancialReport(record);
12         }
13 }
```

LISTING 1. VMJ web resource layer

by adding the `@Route` annotation in the method declarations (see Line 6 in Listing 1). As a result, methods with the `@Route` annotation can be accessed using `HTTP` request.

In the VMJ web framework, the product variants can be generated automatically. The product module is generated based on feature selection in FeatureIDE. For example, `BisaKita` product requires features *Program*, *Income*, *Expense*, *ArusKasReport* and *Donation*. The required modules are compiled into JAR files and the JAR files can be deployed as a running application.

The generated (Java) back-end is combined with generated (JavaScript) front-end to produce a running web application. JavaScript application in the front-end is generated from the IFML diagram [17]. An example of generated website for product `BisaKita` can be accessed in `https://bisakita.amanah.cs.ui.ac.id`. The available features for `BisaKita`, such as *Program*, *Income*, *Expense*, *Donation*, and *ArusKasReport*, are shown at the menu bar.

5. **Evaluation.** For evaluation purpose, we design a scenario for requirements changes in the AMANAH case study. A new organization, namely `HeroFoundation`, requires a new feature in AMANAH, annual financial reports. This feature provides an automatic financial report for one year. Figure 3 shows the modification of AMANAH feature diagram (a new feature is denoted by a red rectangle).

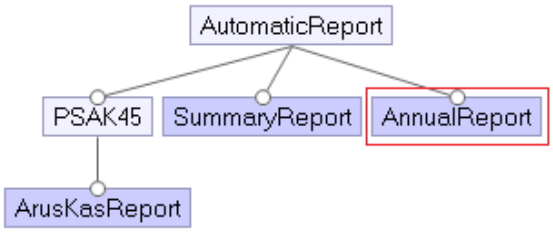

FIGURE 3. A new feature in AMANAH feature diagram

The VMJ web framework is compared to another Java web framework (Spring Boot)[1] to evaluate the development and requirements changes process. We choose Spring Boot. We implement the AMANAH case study, and the requirements changes scenario in Spring Boot. We compare how to implement a new feature in VMJ web and Spring Boot to analyze the preplanning effort for generating a new product variant. Ideally, preplanning in SPLE aims to minimize the effort to change existing implementation [18].

The comparison of implementing a new feature in WinVMJ and Spring Boot is summarized in Table 1. We categorize the process into four stages: *preparation*, *implementation*, *modification*, and *product generation*. At the preparation stage, a new module is created in WinVMJ. In Spring Boot, we have to create a new project, clone an existing project, and create a new package. At the implementation stage, the WinVMJ and Spring Boot process is similar, creating a new class. At the modification stage, WinVMJ uses the decorator pattern that preserves behavior in the existing classes. In Spring Boot, changes are made to existing classes, as in the standard *clone and own* approach. At the product generation stage, all required classes are generated in WinVMJ, as defined in SPLE. However, a new main class must be manually developed in Spring Boot.

Based on the process in Table 1, WinVMJ requires three new classes to develop a new product `HeroFoundation` with a new feature *AnnualReports*: class `Year.java`, decorator class, and (generated) product main class. In Spring Boot, assume there are nine existing classes to implement the Financial Report feature. To develop a new product, `HeroFoundation`, these nine classes are cloned to the new project, and then two new

[1]Based on https://hotframeworks.com/languages/java, Spring Boot is the most popular Java web framework.

TABLE 1. Comparison of requirements changes in VMJ web and Spring Boot

Process	VMJ web	Spring Boot
Preparation	1. Create a new Java (delta) module for annual reports `amanah.automaticreport.annual`	1. Create a new Spring Boot project `HeroFoundation` 2. Clone existing AMANAH project to the new Spring Boot project (`HeroFoundation`) 3. Create a new package for annual reports
Implementation	2. Create a new class `Year.java` in the model layer of the new module	4. Create a new class `Year.java` in the model layer of the new package
Modification	3. Create a new decorator class and add a new method to group the automatic report by year in the *resource* layer of the module	5. Modify existing class `AutomaticReport.java` by adding a method that implements annual reports
Product Generation	4. Generate a new Java (product) module `amanah.product.herofoundation`	6. Create a new main class *HeroFoundation.java*

classes (class `Year.java` and product main class) are added. Eleven new classes are created in total. Therefore, the preplanning effort to develop a product variant with a new feature in WinVMJ is lower than the effort in Spring Boot.

6. **Related Work.** In our previous work, a web-based SPL framework based on DOP is defined on top of the ABS language in [8]. The ABS microservices framework supports the development of microservice product line applications [8]. It produced a back-end of web applications using generated Java code from ABS language. In this work, the VMJ web framework is also designed based on DOP but it uses Java programming language in the development. Using standard Java in the VMJ web framework makes development easier as it removes the need to learn a new programming language.

Web frameworks based on SPLE have been studied by [6, 7, 9, 10]. They use standard programming languages, e.g., Java, HTML, and JavaScript, that are not designed to implement commonality and variability in SPLE. At the implementation level, variability is managed in an ad-hoc manner. Therefore, the relationship between variants of features and their implementation is difficult to define. In this research, we use DOP to implement a web-based SPL in the solution space. DOP has good support for feature traceability, an ability to trace a feature from the problem domain to implementation in the solution domain [18].

7. **Conclusion and Future Work.** This paper proposes a framework for web-based SPL development called VMJ web. The VMJ web framework is designed based on VMJ, an architectural pattern to implement SPLE in Java. The development process in the VMJ web framework follows delta-oriented software product lines in VMJ. The core and delta modules are defined in Java (modules) using the decorator and factory patterns. Systematic reuse is achieved in product generation by defining required dependencies to the existing core and delta modules. The VMJ web framework is also integrated with the FeatureIDE tool to support automatic product generation.

The practical application of the VMJ web framework is demonstrated in this paper using a case study, a charity organization system. The steps to develop a web-based SPL include

domain analysis, model transformation, and domain implementation with the VMJ web framework. A specific product from a web-based SPL can be generated based on feature selection. Although we present a small case study, it is representative of SPL problems in web applications. The VMJ web framework can be used in any problem domain of web applications. In this research, we evaluate the VMJ web framework qualitatively by defining a scenario. We compare the preplanning effort of requirements changes in VMJ web and another Java web framework (Spring Boot).

In this research, tool support (VMJ web composer) is developed in FeatureIDE [12]. The FeatureIDE, which is designed on top of Eclipse IDE, integrates a feature model and implementation in VMJ web source code. In future work, we plan to integrate other low-code tools, such as UI generator and UML generator, into Eclipse IDE. Therefore, the whole development process can be conducted in an integrated development environment.

Acknowledgment. The authors would like to thank Hanif A. Prayoga, Falah P. Waluyo, Samuel T. Febrian, and C. Samuel for contribution to the VMJ web framework. This research is funded by Directorate of Research and Development, Universitas Indonesia under Hibah PUTI 2023 (Grant No. NKB-019/UN2.RST/HKP.05.00/2023).

REFERENCES

[1] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, Boston, MA, 2002.

[2] K. Pohl, G. Bockle and F. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*, Springer-Verlag, Berlin, 2005.

[3] O. Matvitskyy, K. Ijima, M. West, K. Davis, A. Jain and P. Vincent, Magic quadrant for enterprise low-code application platforms, *Gartner*, https://www.gartner.com/doc/reprints?id=1-2F7NELJY&ct=231004, 2023.

[4] D. Di Ruscio, D. Kolovos, J. de Lara, A. Pierantonio, M. Tisi and M. Wimmer, Low-code development and model-driven engineering: Two sides of the same coin?, *Software and Systems Modeling*, vol.21, no.2, pp.437-446, 2022.

[5] M. R. A. Setyautami and R. Hähnle, An architectural pattern to realize multi software product lines in Java, *The 15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21)*, New York, NY, USA, 2021.

[6] J. M. Horcas, A. Cortiñas, L. Fuentes and M. R. Luaces, Integrating the common variability language with multilanguage annotations for web engineering, *Proc. of the 22nd International Systems and Software Product Line Conference – Volume 1 (SPLC'18)*, pp.196-207, 2018.

[7] V. Vranic and R. Taborsky, Features as transformations: A generative approach to software development, *Comput. Sci. Inf. Syst.*, vol.13, no.3, pp.759-778, 2016.

[8] M. A. Naily, M. R. A. Setyautami, R. Muschevici and A. Azurat, A framework for modelling variable microservices as software product lines, in *Software Engineering and Formal Methods. SEFM 2017. Lecture Notes in Computer Science*, A. Cerone and M. Roveri (eds.), Cham, Springer, 2018.

[9] J. M. Horcas, A. Cortiñas, L. Fuentes and M. R. Luaces, Combining multiple granularity variability in a software product line approach for web engineering, *Inf. Softw. Technol.*, vol.148, 106910, 2022.

[10] G. H. Alferez and V. Pelechano, Context-aware autonomous web services in software product lines, *2011 15th International Software Product Line Conference*, pp.100-109, 2011.

[11] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake and T. Leich, FeatureIDE: An extensible framework for feature-oriented software development, *Science of Computer Programming*, vol.79, pp.70-85, 2014.

[12] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich and G. Saake, *Mastering Software Variability with FeatureIDE*, Springer International Publishing, Cham, 2017.

[13] F. Damiani, R. Hähnle, E. Kamburjan, M. Lienhardt and L. Paolini, Variability modules for Java-like languages, *Proc. of the 25th ACM International Systems and Software Product Line Conference – Volume A (SPLC'21)*, pp.1-12, 2021.

[14] G. Holl, P. Grnbacher and R. Rabiser, A systematic review and an expert survey on capabilities supporting multi product lines, *Information and Software Technology*, vol.54, no.8, pp.828-852, 2012.

[15] I. Schaefer, L. Bettini, V. Bono, F. Damiani and N. Tanzarella, Delta-oriented programming of software product lines, in *Software Product Lines: Going Beyond*, J. Bosch and J. Lee (eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, 2010.

[16] F. P. Waluyo, M. Setyautami and A. Azurat, UML transformation to Java-based software product lines, *Jurnal Ilmu Komputer dan Informasi (Journal of Computer Science and Information)*, vol.15, no.2, 2022.
[17] H. S. Fadhlillah, D. Adianto, A. Azurat and S. I. Sakinah, Generating adaptable user interface in SPLE: Using delta-oriented programming and interaction flow modeling language, *Proc. of the 22nd International Systems and Software Product Line Conference – Volume 2 (SPLC'18)*, pp.52-55, 2018.
[18] S. Apel, D. Batory, C. Kästner and G. Saake, *Feature-Oriented Software Product Lines*, Springer-Verlag, Berlin, 2013.