Jurnal Ilmu Komputer dan Informasi (Journal of Computer Science and Information)
15/2 (2022), 119-129. DOI: http://dx.doi.org/10.21609/jiki.v15i2.1070

UML Transformation to Java-based Software Product Lines

Falah Prasetyo Waluyo, Maya R.A. Setyautami, Ade Azurat

Faculty of Computer Science, Universitas Indonesia, Kampus UI, Depok, 16424, Indonesia

Email: falah.prasetyo01@ui.ac.id, mayaretno@cs.ui.ac.id, ade@cs.ui.ac.id

Abstract

Software product line engineering (SPLE) is an emerging approach that enables variability management in software development. SPLE offers tremendous benefits, but lack of tool support becomes a barrier in the adoption of SPLE. Variability modules for Java (VMJ) is an implementation approach that is defined based on the variability modules (VM) concept to support SPLE. VMJ combines Java modules system and design patterns that are commonly used by software developers. VMJ is accompanied by a UML profile, called UML-VM profile, which extends UML notation to model variability in the UML diagram. UML-VM diagram is used to model the problem domain, and VMJ is used in the domain implementation. In this research, we design a model transformation from Unified Modeling Language (UML) diagram into VMJ. The transformation rules are defined based on the UML-VM profile and implemented in the Eclipse Acceleo model to text transformation. As a result, a UML diagram can be transformed automatically into Java-based software product lines. The transformation tool is evaluated using a case study by comparing the generated code and the actual implementation.

Keywords: *model transformation, software product line engineering, uml profile, variability modules for java*

1. Introduction

Software product line engineering (SPLE) is a paradigm to develop a family of software by utilizing commonality and variability [1, 2]. SPLE was inspired by product line engineering (PLE) in other industries, such as the automotive industry. Due to the increasing demand for various software, PLE began to be used in software development. SPLE takes advantage of variability management in the same domain to produce various software products in a single development. In *domain engineering*, commonality and variability are analyzed and implemented, and in *application engineering* a product variant can be generated.

As a novel approach, one of the problems in SPLE is the lack of tool supports for applying the principles of product line engineering easily [2]. FeatureIDE [3], a standard tool for the SPLE, supports various implementation approaches, such as feature-oriented programming (FOP) with AHEAD-/Jak, FeatureHouse, FeatureC++, delta-oriented programming (DOP) with DeltaJ [4], and aspect-oriented programming (AOP; Table 1 includes a description of abbreviations and acronyms used) with AspectJ [5]. However, those implementation approaches are not easily applied by software developers because they are not commonly used in standard software development.

Variability modules for Java (VMJ) is proposed to support SPLE development based on Java programming languages [6]. VMJ is a practical implementation of variability modules (VM) concept [7].VM is designed to solve interoperability problems in product line variants. VMJ provides an architectural pattern that combines design patterns and Java modules to develop a product line application. VMJ is also completed with a UML profile designed to capture VM notation in UML, called UML-VM profile. UML profile is a mechanism in UML to extend the UML notation for a particular domain [8].

In model-driven software engineering (MDSE), model transformation and code generation are the main ingredients to support traceability between model and implementation [9]. We utilized these concepts in SPLE to improve the traceability between the problem domain and solution domain. Unified modeling language (UML) is a standard modeling language in software development. Since UML is not designed to model variability in SPLE,

we use a UML diagram completed with UML-VM profile, called UML-VM diagram. The UML-VM profile is also defined as a bridge between UML and VMJ (Java). VMJ is used as an implementation approach in the solution domain.

In this research, we design a model transformation mechanism from the UML-VM diagram to VMJ source code. The input is a UML-VM diagram in Eclipse modeling framework (EMF) format, and the output is a skeleton of Java source code. The transformation rules are defined based on the UML-VM profile. Since VMJ uses the decorator and factory design pattern, the transformation rules also follow those patterns in generating source code. The transformation rules are implemented in Eclipse using the Acceleo model to text transformation. As a result, the transformation tool can be exported as an Eclipse plugin.

Other studies [6] present an architectural pattern and UML-VM Profile to implement multi software product lines in Java. This paper presents the relationship between UML-VM Profile and VMJ code in more detail by showing transformation rules of how each UML element mapped to the VMJ code and providing a transformation tool using those transformation rules. [10] presents the relationship between UML-DOP Profile and implementation code in detail, the language used is ABS modeling language. This paper uses Java with VMJ architectural pattern as an implementation code and UML-VM Profile to model variability in UML. UML-VM is an extension of UML-DOP with more notation to model VMJ in UML.

Table 1. List of abbreviation and acronyms used in the paper

Abbreviation	Explanation
ABS	Abstract Behavioral Specification
AOP	Aspect-Oriented Programming
ATL	Atlas Transformation Language
DOP	Delta-Oriented Programming
EMF	Eclipse Modeling Framework
FOP	Feature-Oriented Programming
M2T	Model-to-Text
MDSE	Model-Driven Software Engineering
MPL	Multi-Product Line
MTL	Model to Text Language
PLE	Product Line Engineering
SPLE	Software Product Line Engineering
UML	Unified Modeling Language
UML-DOP	UML profile for delta-oriented programming
VM	Variability Modules
VMJ	Variability Modules for Java
XMI	XML Metadata Interchange

The structure of this paper is as follows: Section 2 explains summary of SPLE and DOP. In Section 3, VMJ are explained by using an example. UML transformation to VMJ is described in Section 4 by explaining the transformation rules and the implementation in Eclipse. Section 5 shows an evaluation by applying transformation tool to a case study. Related work is discussed in Section 6. Conclusion and future work are explained in Section 7.

2. Software Product Line Engineering

SPLE is an approach in software development to develop various products in a single development. A product line is a set of products that share commonalities and variabilities [2]. Commonalities, requirements required by all products, are defined as a set of reusable parts. Different requirements for any product are managed as variabilities. A product can be derived based on the commonalities and chosen variabilities.

SPLE consists of two main stages, domain engineering and application engineering [2]. Domain engineering is a process to define the commonality and variability of product lines. There are four steps in the domain engineering, i.e., *domain requirement engineering, domain design, domain realization,* and *domain testing*. In the beginning, the economic aspect of the product line is analyzed in product management. Then, the variability is analyzed in the requirement engineering, modeled in the design phase, implemented in the realization, and tested in the last stage.

Application engineering is a process to build a product variant by reusing domain engineering artifacts. It consists of four sub-processes, i.e., *application requirement engineering, application design, application realization*, and *application testing*. Each sub process in the application engineering utilizes artifacts from related subprocess in the domain engineering. For example, in domain engineering, artifacts x_1, x_2, x_3, ..., x_n are defined. In the application engineering, a product variant V1 requires artifact x_2, x_3, x_5. In the application realization, those required artifacts are composed to derive a specific product.

Feature-oriented programming (FOP) is an approach to implement SPLE based on the composition of features [11]. In FOP, commonality and variability are represented as a set of features in a feature model. The feature model consists of variation points, variants, and constraints [2]. The model is used as a reference to implement a reuse mechanism across the entire software life cycle [11]. Constraints describe the dependency or limitation when selecting a list of features. A product variant is identified by a list (subset) of features by conducting a feature

selection process. The product is valid if it satisfies all constraints defined in the feature model.

Delta-oriented programming (DOP) is a paradigm based on FOP to create a product line by composing core modules and delta modules [12]. Core module contains a set of classes that implement a basic product. Delta modules modify the core module to implement some variant of a feature. The modification that can be made include adding, deleting, or modifying code in the core module. A product is derived by applying zero or more corresponding delta modules to the core module.

3. Variability Modules for Java

Variability modules (VM) is an extension of a software module system that captures variability at the level of modules [7]. VM is designed to solve interoperability problems in product line variants. Multiple variants from a similar product line sometimes can not coexist together. Furthermore, managing the dependency of multi variants in different product lines is also challenging. So, VM adopts the module mechanism to manage variability and dependency.

An architectural pattern is designed by [6] to realize the VM concept in Java, called Variability Modules for Java (VMJ). The VM concept is combined with the Java module system (available from Java 9) and design patterns. The main advantage of VMJ is an extension of the Java module system that supports SPLE and multi-product line (MPL). VMJ is more intuitive for anyone familiar with the Java programming language.

In this paper, VMJ is explained using a restoSPL case study. RestoSPL produces applications to manage restaurants' services. Figure 1 shows a snippet of restoSPL feature diagram. RestoSPL have three features, *Menu*, *Booking*, and *Promo*. *Menu* feature is responsible for storing and displaying available foods, *Booking* feature handles reservations, and the promo feature is used to store and verify existing promos. *Menu* is a mandatory feature in the feature diagram, and the others are optional.

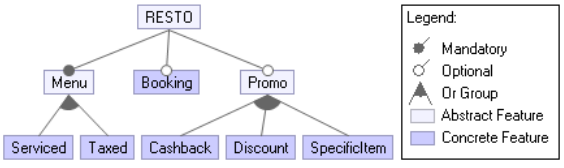

Figure 1. Feature model for restoSPL

In VMJ, the decorator pattern is used to model delta behavior in DOP [6]. "Decorators provide a flexible alternative to sub-classing for extending functionality" [13]. The decorator pattern can change an object's behavior dynamically. The new behavior is added after creating the original object. Thus, the behavior in the existing object (coremodule) can be maintained. In DOP, the core module contains the basic implementation that is common for all features. Following the decorator pattern, in VMJ, a core module consists of interface, *abstract component class*, *concrete component class*, and *abstract decorator class*. The delta module consists of *concrete decorator class*.

Figure 2 shows an illustration of applying decorator pattern to *Promo* feature. Package `MPromo` represents a core module that contains common fields and methods. Interface `Promo` consists of two methods that will be implemented by the other classes. Each *concrete class* that represents a feature's variant implements the same interface. In decorator pattern, this interface is implemented by an *abstract component class*. Class `PromoComponent` also consists of fields that are common for all variants. This *abstract component class* are extended by *concrete component class* `PromoImpl` and *abstract decorator class* `PromoDecorator`.

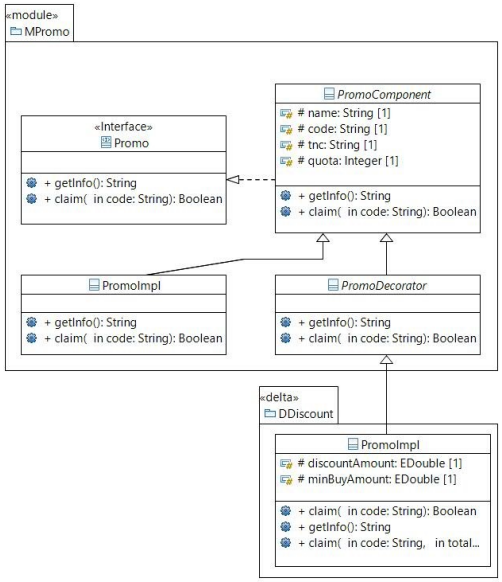

Figure 2. UML Diagram with Decorator Pattern

As defined in the *decorator pattern*, existing behavior in the *concrete class* can be wrapped by additional behavior in the *decorator class*. An abstract *decorator class* is a subclass of *component class* so that the *decorator class* implements the same interface. The decorator class is used to add new behavior to the *component class*. Following the DOP

approach, the additional behavior is implemented in the delta modules. The delta module defines fields and methods that only exist in specific variants [6]. The delta module in VMJ is represented by the java module, which consists of *concrete decorator class* that adds, modifies, or deletes fields and methods performed by the *delta*.

Suppose a variant of promo can give a discount with a specific amount (See Figure 1). It can be done by creating a delta module that implements *discount* variant. In Figure 2, package `DDiscount` is a delta module that modifies module `MPromo`. This package contains a *concrete decorator class* `PromoImpl` that extends class `PromoDecorator`. This *decorator class* implements *Discount* feature, a variant of *Promo* feature. This class consists of two new fields *discountAmount* and *minBuyAmount*. Furthermore, this class also has a new claim mechanism that is implemented in a new method.

The *factory design pattern* is used in VMJ to choose an appropriate variant of features [6]. For example, *Promo* feature has three variants: *Cashback*, *Discount*, and *SpecificItem*. In the decorator pattern, the core module can be modified by three different delta modules. The factory pattern allows creating groups of related objects without specifying their *concrete class* [13]. Therefore, a specific delta can be applied during the product generation process. In VMJ, the factory pattern is implemented by *factory class*, as shown in Listing. 1.

```
1  package resto.promo;
2  import resto.promo.core.Promo;
3  ...
4  public class PromoFactory{
5    ...
6    public static Promo createPromo(String
       fullyQualifiedName, Object ... base)
7    {
8      Promo record = null;
9      ...
10     Class<?> clz =
        Class.forName(fullyQualifiedName);
11     Constructor<?> constructor =
        clz.getDeclaredConstructors()[0];
12     record = (Promo)
        constructor.newInstance(base);
13     ...
14     return record;
15   }
16 }
```

Listing 1. *Factory class* code PromoFactory.java

The *factory class* is required to refer to the correct variant in the object creation. Method *createPromo* in Line 6 is a factory method to create an object of interface type *Promo*. This method has two parameters that specify the variants of an object. The first parameter is filled with *fullyQualifiedName* of the class that is created. In line 10, *fullyQualifiedName* is used to get the appropriate class and constructor. The second parameter is a *variable-length argument* that can take zero or more arguments. This parameter contains fields that are required to create an object. A *variable-length argument* is needed because each variant may have various number of arguments. Then the object creation is done in line 12 with the *variable-length argument base*.

The factory fattern is utilized in the product derivation stage. In VMJ, the product derivation of SPLE is implemented by creating a java module that represents a new product. A product consists of selected features. The *feature selection* in VMJ is specified in the *module declaration* (*module-info.java*) [6]. For example, `KingFood` restaurant requires features menu with tax, discount promo, and specific item promo. The module declaration of `KingFood` product is shown in Listing. 2. `KingFood` product has dependencies on *menu* features with *taxed* variants and *promo* features with *discount* variants. The build script will read the *product module declaration* to determine which module is required and check whether the product does not violate some constrain. If the product is valid, then the build script will generate a jar file for each module and store it in a folder. The product can be run by executing the product module. Product variations can be done by changing the contents of *module-info.java* with other features.

```
1 module resto.restaurant.kingfood {
2   requires resto.menu.core;
3   requires resto.menu.taxed;
4   requires resto.promo.core;
5   requires resto.promo.discount;
6   requires resto.promo.specificItem;
7 }
```

Listing 2. *Product module declaration* code module-info.java

4. UML to VMJ Transformation

Unified modeling language (UML), a standard modeling language in software development, is not designed to model variability in SPLE. A UML profile for delta-oriented programming (UML-DOP) profile is defined in [10] to model variability in UML. VMJ is also accompanied by a *UML profile* called UML-VM which map VM elements into the UML diagram [6]. UML-VM is an extension of the UML-DOP profile that can model VMJ in UML notation.

Figure 3 is an example of UML-VM diagram of RestoSPL that models *Promo* feature. As defined in Figure 1, *Promo* feature has three variations *Cashback*, *Discount*, and *SpecificItem*. In DOP, each variation is implemented by delta modules. For example, in Figure 3, there is two UML packages with <<delta>> stereotypes that represent delta modules

in DOP. `DDiscount` delta modifies `PromoImpl` class in the core module. This modification is modeled as `PromoImpl` class in `DDiscount` delta that has <<modifiedclass>> stereotype.

A feature is modeled as UML component with <<feature>> stereotype. The relation between deltas and features are modeled as UML dependency. For example, `Discount` is implemented by `DDiscount` delta. In Figure 3, there is a UML dependency with <<when>> between UML component `Discount` and UML package `DDiscount`. This dependency implies that `DDiscount` delta is applied to the core module when `Discount` feature is selected in a product. A product is represented as UML component with <<product>> stereotype, e.g., `KingFood` product in Figure 3.

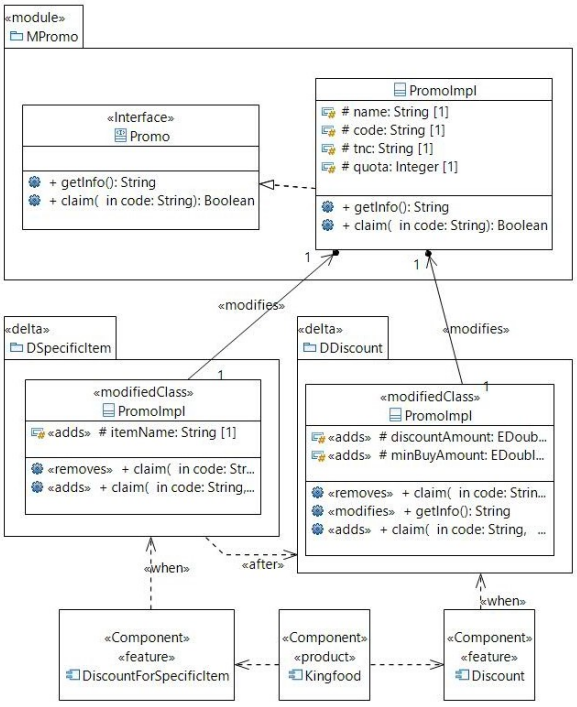

Figure 3. snippets of UML-VM diagram for restoSPL

In MDSE, a model transformation approach is used to generate a source code based on models [9]. This research uses the model-to-text (M2T) transformation approach to generate Java source code from the UML-VM model. M2T transformation takes models as input and produces text (source code) as output. Code generation is an application of M2T transformation to achieve the transition from the model level to the code level [9].

The model transformation mechanism in this research is shown in Figure 4. The input is a UML-VM diagram, a UML diagram that uses a UML-VM profile. We use a textual representation of the UML diagram in XML Metadata Interchange (XMI) format. The XMI format follows the Eclipse modeling framework (EMF). The UML diagram is processed by 'UML to VMJ' transformation tool to produce VMJ source code. The tool is implemented in Eclipse Acceleo based on transformation rules. The transformation rules are defined based on the UML-VM profile. The UML-VM profile maps VM elements to UML notation so that the transformation rules can be derived from the UML-VM profile.

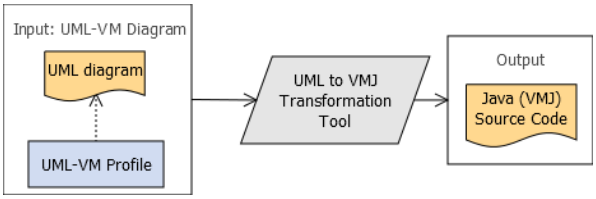

Figure 4. Transformation mechanism

4.1. Transformation Rules

Transformation rules from UML-VM to VMJ source code are defined based on UML-VM profile. UML-VM profile supports the UML-VM model transformation into any language that supports DOP, in this case, Java with VMJ. For example, a UML package with stereotype <module>> represents a core module in VM. Thus, this package is transformed into Java core module in VMJ. Summary of UML to VMJ transformations rules are defined in Table 2. The following subsections describes how to transform UML elements into VMJ (Java) source code, starting from package elements to component elements.

4.1.1. UML Package. The UML package represents a core module or delta modules in VMJ. In VMJ, the naming convention is used to differentiate those modules. The module name consists of three parts: <productline-name>.<feature-name> .<module-name>. A core module name ends with `core` and a delta module use delta's name in the module-name. For example, `resto.promo.core` represents a core module, and `resto.promo.discount` represents a delta module *Discount*.

In Figure 3, there are three packages. `MPromo` package has <<module>> stereotype. This stereotype indicates that `MPromo` is a core module. The other two packages have <<delta>> stereotype, indicating that the package is a delta module. In VMJ, the core module and the delta module are represented

Table 2. UML to VMJ Transformation Rules

UML Elements	**Stereotype**	***VMJ Elements***
Package	<<module>>	Java (core) module. Java package inside the module
Package	<<delta>>	Java (delta) module. Java package inside the module
Interface	<<interface>>	Interface in Java core module
Class	–	Several classes in Java core module, following decorator pattern, such as: Abstract Component Class, Concrete Component Class, Abstract Decorator Class
Class	<<modifiedClass>>	Concrete decorator class in Java delta module
Attribute	<<adds>>, <<removes>>	A new attribute in concrete decorator class. Create new constructor and throw exception in related setter getter
Operation	<<adds>>, <<removes>>, <<modifies>>	A new method in concrete decorator class. Throw an exception in concrete decorator class. Override method in concrete decorator class
Component	<<feature>>	Feature declaration in file ck.properties, and define relates to delta modules
Component	<<product>>	Java product module

by the Java module. The name of the Java module depends on the package name and its stereotype. A Java module contains several java packages. The main package in the Java module has the same name as the Java module name.

A Java module has *module declaration* in `module-info.java` file. This file specifies exported *java packages* and required *external modules*. In UMl-VM, those information can be retrieved from the packages dependencies. For example in figure 3, the *class* in the `DDiscount` package has an association with the *class* in `MPromo`. Therefore *module-info.java* in the `DDiscount` package requires Java module `resto.promo.core`.

Each core module in VMJ has a *factory class* to create an appropriate variant using the factory pattern. A template for a *factory class* is provided so that the *factory class* is generated automatically. The name of the *factory class* is similar to the interface name and followed by the Factory keyword. For example, in Figure 3, there is an interface `Promo`. Therefore, class `PromoFactory` is created using the template. The *factory class* is also responsible to handle the *delta application order* [6]. If a feature is implemented by more than one delta, *delta application order* must be specified. Delta application order can be retrieved from UML the *dependency* with the <<after>> stereotypes. In figure3, `DSpecificItem` has a *dependency* with `DDiscount` package. When feature *DiscountForSpecificItem* is selected, delta `DSpecificItem` must be applied to the core module after delta `DDiscount`. This condition is specified as a constraint in *factory class* when applying the decorator pattern.

4.1.2. UML Interface. The core module must have an interface that defines abstract methods. Interface is used for making each feature variant *interoperable*. Figure 5 shows an interface `Promo` in RestoSPL. As shown in Figure 3, this interface is a part of the core module. This indicates that the Java module `'resto.promo.core'` has a Promo interface. The transformation from UML interface to VMJ is straightforward because UML interface has similar meaning with Java interface. For example, the result of transforming interface `Promo` is shown in Listing. 3.

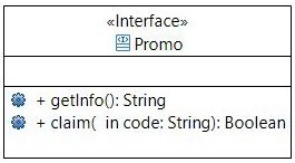

Figure 5. Interface in MPromo core module

```
1 package resto.promo.core;
2 public interface Promo {
3   Boolean claim(String code);
4   String getInfo();
5 }
```

Listing 3. Code for Interface Promo

4.1.3. UML Class. In the UML-VM diagram, there are two kinds of class: a class in the core module and a class in the delta module. In the core module, a UML class does not have stereotypes because it has a similar meaning to a standard UML class. In the delta module, a UML class can have <<modified class>>, <<addedClass>>, or <<removedClass>> stereotypes. Thus, the transformation from UML class to VMJ is separated into two conditions: a class in *core* and delta modules.

1. Core Module

As shown in Figure 3, `MPromo` package has `PromoImpl` class (Figure 6) that implements `Promo` interface. Following the decorator pattern, a class in core module is transformed into three classes

in the VMJ: *abstract component class*, *concrete component class*, and *abstract decorator class*. As a result, based on `PromoImpl` class in the UML-VM diagram, Listing. 4 and Listing. 5 can be generated. Listing. 4 shows the *abstract component class* that contains fields and abstract method. Listing. 5 shows the implementation of *concrete component class* in the core module that extends an *abstract component class*.

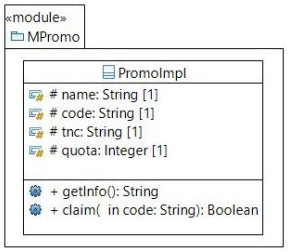

Figure 6. Class in MPromo core module

```
1  package resto.promo.core;
2  public abstract class PromoComponent implements
       Promo {
3    protected String name;
4    ...
5    public PromoComponent(String name, String code,
       String tnc, int quota) {
6      this.name = name;
7      ...
8    }
9    public String getName() { return name; }
10   public void setName(String name) { this.name =
       name; }
11   ...
12   public abstract String getInfo();
13 }
```

Listing 4. *abstract component class* code PromoComponent.java

```
1  package resto.promo.core;
2  public class PromoImpl extends PromoComponent {
3    ...
4    public String getInfo() {
5      //implementation method
6    }
7  }
```

Listing 5. *Concrete component class* code PromoImpl.java

2. Delta Module

In DOP, a delta module can modify a core module by adding, modifying, or removing classes, interfaces, fields, or methods. In Figure 3, there is a package with stereotype `<<delta>> DDiscount`. Delta `DDiscount` modifies `PromoImpl` class in the core module by adding new attributes and methods, removing a method, and modifying a method. In the UML-VM diagram, this delta package consists of `PromoImpl` class that has <<modifiedClass>> stereotype. As shown in Figure 7, these modified class has modified methods and fields.

As defined in Table 2, a class with <<modifiedClass>> stereotype is transformed into a *concrete decorator class* in VMJ. The *concrete decorator class* can add, modify and delete fields and methods from the decorated component. In the UML-VM diagram, this modification is represented by <<adds>>, <<modifies>>, or <<removes>> stereotypes in the fields or methods. Listing. 6 shows a *concrete decorator class*, a result of transforming UML-VM diagram into VMJ. Lines 5-6 show the code of adding fields and line 19 shows the added method. Following the decorator pattern, modified constructor in Line 8 wraps `PromoComponent` object with the new fields.

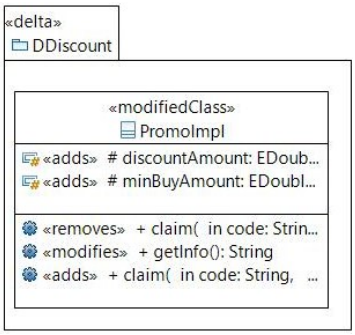

Figure 7. Class in DDiscount delta module

```
1  package resto.promo.discount;
2  ...
3  public class PromoImpl extends PromoDecorator {
4    //new fields
5    double discAmount;
6    double minBuyAmount;
7    //modified constructor
8    public PromoImpl(PromoComponent record, double
       discAmount, double minBuyAmount) {
9      super(record);
10     this.discountAmount = discountAmount;
11     this.minBuyAmount = minBuyAmount;
12   }
13   ...
14   public Boolean claim(String code){
15     throw new UnsupportedOperationException();
16   }
17   public Boolean claim(String code, double
       totalPrice){
18     // implementation method
19   }
20 }
```

Listing 6. *Concrete decorator class* code

4.1.4. UML Component. In the UML-VM diagram, a UML component can have <<feature>> or <<product>> stereotypes. For example, in Figure 3, `Discount` component has <<feature>> stereotype. This component represents a feature in a product line, as shown in the feature diagram. In DOP, a feature is implemented by one more deltas. As

shown in Figure 3, `Discount` feature has a dependency to `DDiscount` delta. Relation between deltas and features are defined in the configuration knowledge [14]. In VMJ, the configuration knowledge is defined in file *'ck.properties'*. This configuration file maps the name of the Java delta module to a feature. Therefore, the UML dependency between deltas and features in the UML-VM diagram is transformed into configuration files.

A product in the product line is represented as a Java module in VMJ. Different with core Java module and delta Java module, a product Java module is identified with a specific name: `<productline-name> .product.<product-name>`. The UML component with <<product>> stereotype is transformed into product Java module. For example, `Kingfood` is a product in the restoSPL. In Figure 3, `Kingfood` is modeled as component with <<product>> stereotype. In VMJ, `Kingfood` is realized as a Java module `resto.product.kingfood`. A product consists of selected features. This feature selection is modeled in the module declaration, as shown in Listing. 2. The product declaration is generated from UML dependency between the `Kingfood` component and the corresponding <<feature>> component.

4.2. Implementation of Automated UML-VMJ Transformation

In this research, an automated model transformation tool from the UML-VM diagram to VMJ source code is developed based on the transformation rules. We use Acceleo[1] model-to-text transformation as tool support to create the tool. Acceleo is a template-based technology based on Eclipse to create custom code generators. To generate a code, Acceleo requires an input model (in this case, UML-VM diagram) and a code template in the Acceleo model to text language (MTL). Transformation rules are implemented in Acceleo MTL. Acceleo language can iterate over a set of UML elements and transform those elements into a source code. In addition, Acceleo can invoke some Java code from Acceleo MTL.

Listing. 7 shows a snippet of transformation rules that are implemented in Acceleo MTL. This code transforms a UML class into a *concrete component class* in VMJ. Acceleo MTL has a built-in library, like functions in Java that return a string. In line 3, `aClass.name.toUpperFirst()` takes the class name, capitalizes its first letter to meet the Java class naming convention. Lines 7-10 iterate over operations in the UML class to create methods on the generated *concrete component class*. Since UML class diagram does not contain a method implementation, a developer must complete the implementation manually.

```
1  [template public generateImpl(aClass : Class)]
2  package [aClass.corePackageName()/];
3  public class [aClass.name.toUpperFirst()/]
       extends [aClass.classComponent()/] {
4    public
       [aClass.name/]([aClass.constructorArgs()/]) {
5      super([aClass.constructorArgsNoType()/])
6    }
7  [for (o: Operation | aClass.ownedOperation)
       separator('\n')]
8    [o.visibility/] [o.minMethodHeader()/] {
9      // TODO: implement this method
10   }
11 [/for]
12 }
13 [/template]
```

Listing 7. Snippet of *g*enerate_impl.mtl

All transformation rules in Section 4.1 have been implemented in Acceleo. Acceleo model transformation can be exported into the Eclipse plugin using eclipse modeling tools. As a result, an automated model transformation from UML-VM to VMJ is produced as an Eclipse plugin. This plugin can be used to transform any UML-VM diagram into VMJ source code. In the following section, an experiment is conducted to show the application of automated model transformation.

5. Evaluation

Evaluation of model transformation from UML to VMJ is conducted by applying the transformation tool to the restoSPL case study. RestoSPL case study consists of six concrete features, as shown in in Figure 1. The input of transformation tool is a UML-VM diagram of RestoSPL, that consists of eight packages and ten classes. The output is a skeleton code of Java programs that follow VMJ architectural pattern. The generated code must be completed manually by adding the implementation to produce a running product line application. As evaluation, we have implemented the restoSPL case study manually without using the transformation tool. We compare the generated VMJ code with the source codes that have been implemented manually.

Table 3 shows the result of comparison between the generated code and the completed source code. We evaluate the number of files, lines of code, classes, and modules. The number of files, classes and modules of the completed VMJ code is same as generated VMJ code. The transformation tools has successfully mapped elements from UML to VMJ source codes. We can conclude that the UML-VM

[1]https://www.eclipse.org/acceleo/overview.html

Table 3. Comparison of generated VMJ code with completed VMJ code

Compared in	*Generated Code*	*Completed Code*
Num of Files	38	38
Num of Lines of Code	829	906
Num of Classes	23	23
Num of Modules	11	11

diagram can model VMJ completely. The transformation tools also maintain consistencies between the UML diagram and implementation because the number of classes and modules remains the same, which means the classes and packages elements from the UML diagram successfully mapped to VMJ codes. There is no need to create new classes manually. Therefore, what is modeled on the UML diagram can describe the structure of VMJ code. If the VMJ code is done manually from scratch, the developer might forget to add some elements from the UML diagram to the VMJ code creating inconsistencies between the UML diagram and VMJ code. In Table 3, the number of files in generated code are similar to completed code, but 20 files in the generated code are modified to produce a running application. As a result, the number of lines of code between generated and completed code are different because several files must be completed manually. Therefore, the addition of lines of codes is happened as expected.

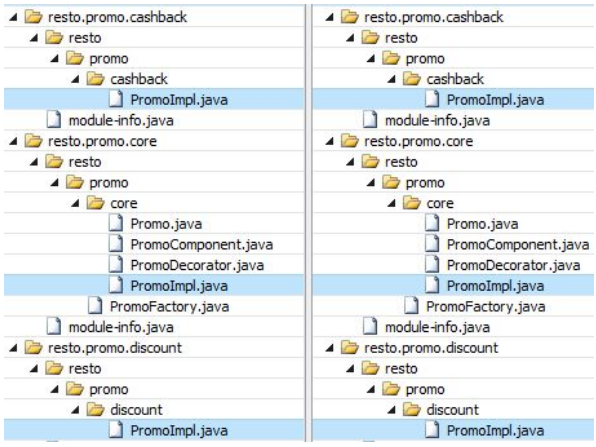

Figure 8. Folder comparison between generated code and completed code

Figure 8 shows folder comparison between generated code and completed code of core module Promo and its delta modules. Blue-colored files indicate that those files have been modified. We can see Impl files have been modified because those file contains concrete implementation that must be added

Figure 9. File comparison between generated code and completed code of PromoImpl.java

manually. In Figure 9, we can see the comparison between generated code and completed code of PromoImpl.java from core module Promo. Red-colored lines are deleted lines from the generated code, whereas green-colored lines are added lines from the completed code. We can see that the developer only needs to fill the method implementation. Class constructor and other things already come from generated code. The time spent developing the code could be reduced, and the developer could allocate their time to other things.

Based on our experiments, the UML to VMJ transformation tool can process all elements in the UML diagram and into a VMJ code. The structure of generated VMJ code contains all UML elements and follows the design patterns in VMJ. Because it follows the design patterns in VMJ, the quality of generated VMJ code can be on par with VMJ code that has been done manually. The developer must complete the generated VMJ code because the output of transformation tool is a code skeleton that follow structure behavior from the UML diagram. The dynamic behavior, such as method's implementation, must be added manually. The transformation tool only works with UML diagram that uses the UML-VM profile defined in [6]. Random UML diagram is not guaranteed to work with this transformation tool as the transformation tool use UML-VM profile as a foundation in the transformation rules to map UML diagram into VMJ code. The transformation tool is exported as an Eclipse plugin so that it can be used

by any Eclipse user.

6. Related Work

Model transformation is utilized to close the gap between models and source codes. The developers can use model augmentation such as UML profiles or leave the model specification open and fill the details at the code level. A research in [15] found that a UML profile can be used to support model transformation in Atlas Transformation Language (ATL). ATL is one of the transformation languages in Eclipse Modeling Framework (EMF) [16]. In this research, we utilize a UML profile in another EMF tool, Acceleo model-to-text transformation.

Practical implementation of using UML profile in model transformation can be found in [17], [18], and [19]. UML diagram is transformed to Alloy[2] in [17]. A UML profile for Alloy is defined to represent Alloy concepts in the UML and supports the model transformation. In [18] and [19], UML profile is used to support UML transformation to abstract behavioral specification (ABS)[3] and vice versa. Model transformation in [18] is implemented in ABS compiler, and in [19], the automated transformation is developed using Python programming language.

7. Conclusion and Future Work

In this research, a model transformation tool is developed to support SPLE. The tool can automatically transform a UML diagram into Java-based software product lines. The UML-VM profile is used as a foundation in the transformation rules that map elements from the UML diagram into VMJ source code. Based on the transformation rules, the tool is developed in Eclipse Acceleo. The transformation tool can process elements from the UML diagram into VMJ code. Therefore, the developer does not need to write code from scratch, and the generated code follows the structure of the UML diagram. This tool is evaluated using a case study by comparing the result of generated source code and the code implemented manually.

VMJ also supports a MPL, several interrelated product lines with dependencies. However, the model transformation does not support the MPL. Further research can be conducted to support the UML diagram of MPL to support the development of MPL. In addition, a web framework for VMJ, called WinVMJ, has been developed to support web-based software product lines. Since the structure of WinVMJ is a bit different from VMJ, some adjustments in the model transformation tools are required so that the generated code follows the structure of WinVMJ.

[2]Alloy is a high-level modelling language for specifying Object-Oriented systems (https://alloytools.org/)

[3]ABS is an executable modeling language (https://abs-models.org/)

References

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

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

[3] 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, experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

[4] J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani, "DeltaJ 1.5: Delta-oriented programming for Java 1.5," in *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, ser. PPPJ '14. New York, NY, USA: ACM, 2014, pp. 63–74.

[5] I. Groher and M. Voelter, *Aspect-Oriented Model-Driven Software Product Line Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 111–152.

[6] M. R. A. Setyautami and R. Hähnle, "An architectural pattern to realize multi software product lines in java," in *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, ser. VaMoS'21. New York, NY, USA: Association for Computing Machinery, 2021.

[7] F. Damiani, R. Hähnle, E. Kamburjan, M. Lienhardt, and L. Paolini, "Variability modules for java-like languages," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–12.

[8] OMG, *OMG Unified Modeling Language (OMG UML) Version 2.5.1*, Object Man-

agement Group, 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/PDF

[9] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[10] M. R. A. Setyautami, R. Hähnle, R. Muschevici, and A. Azurat, "A UML profile for delta-oriented programming to support software product line engineering," in *Proceedings of the 20th International Systems and Software Product Line Conference*, ser. SPLC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 45–49.

[11] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Berlin: Springer-Verlag, 2013.

[12] 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, pp. 77–91.

[13] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: "Elements of Reusable Object-Oriented Software"*. Addison-Wesley, 1994.

[14] R. Hähnle, "The abstract behavioral specification language: A tutorial introduction," in *Formal Methods for Components and Objects: 11th International Symposium, FMCO 2012, Bertinoro, Italy, September 24-28, 2012, Revised Lectures*, E. Giachino, R. Hähnle, F. S. de Boer, and M. M. Bonsangue, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–37.

[15] A. Randak, S. M. Perez, and M. Wimmer, "Extending ATL for native UML profile support: An experience report," in *Proceedings of the 3rd International Workshop on Model Transformation with ATL, MtATL@TOOLS 2011, Zürich, Switzerland, July 1st, 2011*, ser. CEUR Workshop Proceedings, I. Kurtev, M. Tisi, and D. Wagelaar, Eds., vol. 742. CEUR-WS.org, 2011, pp. 49–62.

[16] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "Atl: A model transformation tool," *Science of Computer Programming*, vol. 72, no. 1, pp. 31–39, 2008, special Issue on Second issue of experimental software and toolkits (EST).

[17] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from UML to alloy," *Softw. Syst. Model.*, vol. 9, no. 1, pp. 69–86, 2010.

[18] R. Muhammad and M. R. Setyautami, "Automatic model translation to UML from software product lines model using UML profile," in *2016 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*. IEEE, Oct 2016, pp. 605–610.

[19] M. R. A. Setyautami, R. R. Rubiantoro, and A. Azurat, "Model-driven engineering for delta-oriented software product lines," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, Dec 2019, pp. 371–377.