UNIVERSITAS INDONESIA

MODEL-DRIVEN ENGINEERING FOR DELTA-ORIENTED SOFTWARE PRODUCT LINES

DISSERTATION SUMMARY

MAYA RETNO AYU SETYAUTAMI
1706125134

FAKULTAS ILMU KOMPUTER
PROGRAM STUDI DOKTOR ILMU KOMPUTER
DEPOK
2023

ABSTRACT

Name	:	Maya Retno Ayu Setyautami
Study Program	:	Doktor Ilmu Komputer
Title	:	Model-Driven Engineering for Delta-Oriented Software Product Lines

Software product line engineering (SPLE) is an approach that enables the development of software with shared commonality and variability. It offers a reusable mechanism for creating various products within a specific domain. In this research, we aim to design a model-driven SPLE (MDSPLE) approach based on delta-oriented programming (DOP). The proposed approach encompasses the SPLE process in both the problem and solution domains. In the problem domain, we use feature models and Unified Modeling Language (UML). A UML profile, namely the UML-VM profile, s defined based on variability modules (VM) to model variations in UML diagrams. We introduce a new implementation approach called Variability Modules for Java (VMJ) in the solution domain. VMJ is an architectural pattern in Java that follows DOP principles. Furthermore, we employ the Interaction Flow Modeling Language (IFML) with DOP extension to model abstract user interfaces (UI). A set of tools has been designed within the Eclipse IDE to support the development process, called Prices-IDE. The process in Prices-IDE encompasses modeling in the diagram editors, model transformation, domain implementation, and product generation. The practical application of the proposed MDSPLE approach is demonstrated through a case study. The evaluation of the approach focuses on three perspectives: the applicability of the UML-VM profile as a foundation, the degree of automation in the code generator, and the process improvement. We also discuss threats that could affect the validity of this research. In conclusion, this research contributes to the advancement of SPLE methodologies based on DOP through the proposed MDSPLE approach.

Keywords:
delta-oriented programming, model-driven, software product line engineering, uml-profile, variability modeling

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODES

CHAPTER 1

INTRODUCTION

In this chapter, we describe the introduction of the research to illustrate the background and motivation. We state the research objectives and formulate the problems as research questions. At the end of this chapter, we summarize the research contribution.

1.1 Background

Software product line engineering (SPLE) is a software development approach based on commonality and variability (Clements & Northrop, 2002; Pohl et al., 2005). It offers a systematic reusability for developing a range of products within a specific domain, utilizing platforms and mass customization. Platforms enable the reuse of common components, while mass customization effectively manages variability. For instance, consider a series of smartphones with standard features such as *contact*, *calendar*, and *messaging*. Each smartphone type also exhibits variations in *resolution*, *screen size*, and *storage capacity*. By employing SPLE, such variability can be managed systematically, enabling the generation of various products based on selected features.

In contrast, without SPLE, similar products with variations are typically developed using a traditional approach known as *clone-and-own*. In this approach, an existing project is cloned into a separate repository to develop a different product. Any required modifications to a specific product are made within the new repository. Although all products use the same initial source code (codebase), this approach leads to scattered modifications across repositories. Requirements changes are difficult to trace, and variability cannot be managed systematically. Moreover, as the number of variations increases, development costs escalate, and the maintenance process becomes complex due to the separate source code management in individual repositories.

The adoption of SPLE, as emphasized by Pohl et al. (2005), is motivated by several key factors: reduction of development costs, enhancement of software quality, and reduced time to market. One of the challenges in SPLE is the lack of sufficient tool support to facilitate the application of product line engineering principles (Pohl et al., 2005). To address this, a model-driven approach is designed in this research to support SPLE implementation. Model-driven software engineering (MDSE) enhances the benefits of modeling through automated code generation. Previous studies by Ouali et al. (2013); Martinez et al. (2015); Arboleda & Royer (2012); Hernández-López et al. (2018) have proposed the model-driven approach for SPLE (MDSPLE). Ouali et al. (2013); Martinez et al. (2015) focused on modeling the problem domain and provided supportive tools for the modeling process. Meanwhile, Arboleda & Royer (2012); Hernández-López et al. (2018) proposed the MDSPLE approach that encompasses both the problem and solution domains. They designed the automation process for generating applications based on aspect-oriented programming (AOP).

This research makes a new contribution to MDSPLE by incorporating delta-oriented programming (DOP) within problem and solution domains. DOP is a paradigm that leverages core and delta modules for implementing SPLE (Schaefer et al., 2010). The core module handles commonalities, while delta modules (deltas) address variabilities. A delta can alter existing behavior without directly modifying a source code. It implements variations by adding, removing, or modifying elements in the codebase. DOP effectively supports requirements changes and variability management in product line applications. Two modeling languages that support DOP are DeltaJ (Koscielny et al., 2014) and abstract behavioral specification (ABS) (Johnsen et al., 2012; Hähnle, 2013).

In our previous work, we have developed tools to support SPLE in Precise Requirement Change Integrated System (PRICES) project[1]. PRICES is a framework for developing web-based software product lines that is supported by the PRICES toolkit. The development of the PRICES toolkit has been carried out incrementally between 2016 and 2019, employing different approaches and techniques. For instance, the model transformation tools have been developed using Python, while the diagram editor leverages the Eclipse IDE. In addition, the ABS compiler, Java, and JavaScript environments are required to execute the generated applications. The utilization of multiple environments and software components introduces complexity in product line development using PRICES. To address this challenge, we propose an integrated development environment (IDE) called Prices-IDE, which aims to streamline the development process based on MDSPLE.

Prices-IDE is designed on top of Eclipse IDE, which has standards support for model-driven engineering, such as data models, model transformation, and code generators. Each PRICES tool is deployed as an Eclipse plugin within Prices-IDE. Furthermore, we plan to integrate Prices-IDE with FeatureIDE (Kastner et al., 2009), a widely-used tool and framework for SPLE that is also built on the Eclipse IDE. FeatureIDE offers support for various implementation approaches, such as *feature-oriented programming* (FOP)

[1]https://rse.cs.ui.ac.id/?open=prices/index

with FeatureHouse, DOP with DeltaJ, and AOP with AspectJ (Meinicke et al., 2017). The integration of Prices-IDE and FeatureIDE expands the capabilities of FeatureIDE by introducing a new implementation approach for developing web-based software product lines.

1.2 Research Questions

As a novel approach in software development, the adoption of the MDSPLE approach requires additional support. During the literature review, existing problems are analyzed to find how this research will contribute to the existing knowledge base in MDSPLE. In this research, the scope of the system to be modeled is a web-based application. We consider problems at the modeling and implementation levels. Therefore, this research attempts to answer the following research questions:

RQ1 Which artifacts of a web application that can be modeled as a product line?
A web application consists of various artifacts with different characteristics and granularities. These artifacts should be designed to support web-based product line development and the scope of modeling and the level of abstraction should be decided.

RQ2 How to model the problem domain of a web-based product line?
The requirements engineering process analyzes commonality and variability in the problem domain. A uniform approach is required to model the variability of a web-based product line in different levels of abstraction and granularities.

RQ3 How to implement the product line based on the model in the problem domain?
The DOP approach is used in the domain implementation. The variability model in the problem domain can be mapped to the DOP implementation based on the UML profile. A model transformation mechanism is designed to bridge modeling language in the problem domain to the DOP language.

RQ4 How to integrate the code generation and product derivation process?
In the product derivation, a variant of a web-based product line can be generated based on selected features. An integrated development environment is required to assist the code generation and product derivation.

1.3 Research Objectives

Model-driven engineering for delta-oriented software product lines is proposed to support web-based product line development. The objectives of this research are:

1. Design an MDSPLE framework to develop software product lines based on DOP
2. Define a unified modeling mechanism for delta-oriented software product lines
3. Design a strategy in the domain implementation to generate running applications
4. Design model transformation tools to bridge the problem and solution domains
5. Design an integrated development environment for tools support in the MDSPLE framework

1.4 Research Limitations

This research presents a method to develop a software product line using MDSPLE. The MDSPLE approach covers the SPLE process in the problem and solution domains. The scopes for this proposed research are limited to:

1. The MDSPLE approach can be applied to any domain. At this moment, the development of supporting tools is still focused on generating web applications.
2. The engineering process is focused on design and implementation of SPLE. We do not consider issue about requirements, testing, nor user experience.
3. The integrated tools are developed in the Eclipse Modeling Tools.

1.5 Research Contributions

After solving all research questions, the contributions of this research are summarized as follows:

1. **Model-driven SPLE (MDSPLE)** approach is designed to guide and support the development of a web-based SPL. The process covers the entire processes of SPLE that can be applied in any problem domain.

2. **The UML-DOP profile** extends the UML metamodel to support variability modeling in SPLE. The UML-DOP profile can be used to design any problem domain with several levels of abstractions. The UML-DOP profile is the main ingredient to support traceability in SPLE.

3. **Variability modules for Java (VMJ)** is designed to support implementation of SPLE using JAVA programming language. VMJ is a new concept in domain implementation that can be used in any problem domain. This concept is realized in the web framework to develop a web-based product line application.

4. **Prices-IDE** is an integrated development environment to develop a product line application. Prices-IDE provides tool support with semi-automated model transformation and code generation. Prices-IDE is deployed as an Eclipse plugin and integrated with FeatureIDE to speed up the adoption of using a new approach.

CHAPTER 2

THEORETICAL FOUNDATIONS

This chapter describes the theoretical foundations of this research, such as model-driven software engineering, software product line engineering, and delta-oriented programming. We also explain the related work about mode-driven software product line engineering to analyze state of the art.

2.1 Model-driven Software Engineering

Model-driven software engineering (MDSE) is a methodology in software development that utilizes models as primary artifacts (Brambilla et al., 2012). MDSE is supported with two main ingredients, *model transformation* and *code generation*. Model-to-text (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 (Brambilla et al., 2012).

In MDSE, a modeling language is a tool to specify models of software in textual or graphical representations. Unified Modeling Language (UML) is a standard modeling language widely used in software development. As a graphical modeling language, the abstract syntax of UML is defined in the metamodel. UML metamodels contain class, attributes, and associations that define the modeling concepts and properties. The specification of UML, including the *abstract syntax*, *concrete syntax*, and semantics, are explained by Object Management Group (OMG) in OMG (2017).

UML has extension mechanisms to customize UML metamodel for specific purposes, called UML profile. The UML profile provides flexibility for UML so that UML can be used to model different platforms or domains. UML profile is defined by specifying stereotypes, constraints, and tagged values. Stereotypes are used to refine meta-classes by defining additional semantics to the element represented by the meta-class (Brambilla et al., 2012). A stereotype defines an extension for existing metaclasses in the UML diagram. A stereotype uses the same notation as a class, with keyword `<<stereotype>>` before the name of the class.

As part of MDSE, models can be transformed by using several scenarios, such as *merged* with other models, *refactored* into other models, *refined* to be more detailed, or *translated* into other languages. Those scenarios are implemented as model transformation. There are two kinds of model transformation, *model-to-model* and *model-to-text*. Model-to-model (M2M) transformation is a program which takes models as input to produce models as output (Brambilla et al., 2012). Model-to-text (M2T) transformation takes models as input and produces text (source code) as output.

2.2 Software Product Line Engineering

Software product line engineering (SPLE) is an approach in software development that can deliver various products in the same product line. A product line is a set of products that share commonalities and variabilities (Pohl et al., 2005). Commonalities, requirements that are 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. Figure 2.1 shows the SPLE framework in (Pohl et al., 2005) that consists of two main stages, domain engineering and application engineering. Domain engineering is a process to define the commonality and variability of product lines. Each step in domain engineering produces artifacts that are utilized in the application engineering stage.

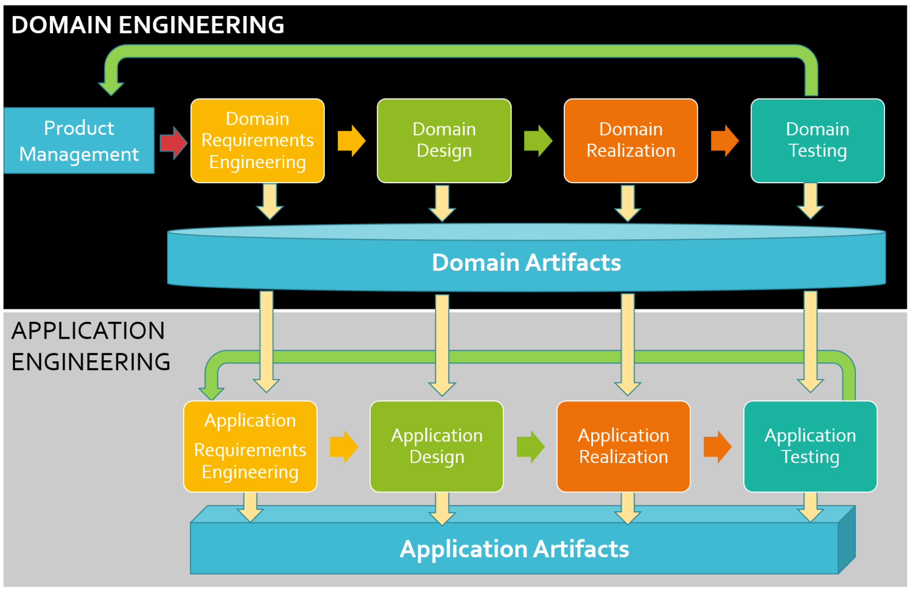


Figure 2.1: SPLE Framework
Source: Pohl et al. (2005)

As a new approach, the adoption of SPLE in software development is quite challenging. Several companies already have various products in the same domain, but they have developed using a traditional approach such as a *clone-and-own*. To adopt SPLE, a company prefers to utilize the existing artifacts instead of developing from scratch. The strategy may differ with another company that wants to try SPLE with a new domain without any existing product or artifact. Krueger (2002) defined three options to start software development with SPLE :

- **Proactive Approach**. The proactive approach provides a mechanism to develop a product line from scratch. The development processes are planned and conducted before generating any product.
- **Extractive Approach**. The extractive approach facilitates a company with various products and wants to transform the old development approach to SPLE. By using an extractive approach, a variability model can be identified based on existing products, and the implementation can be extracted from the existing source code.
- **Reactive Approach**. The reactive approach is an agile methodology to adopt SPLE. In this approach, the variability model is not defined entirely at the beginning. A company starts with an initial product of software product line. The variabilities are modeled and implemented incrementally when a new product variant is needed.

2.3 Delta-Oriented Programming

Delta-oriented programming (DOP) is a paradigm to implement SPLE by defining a core module and a set of delta modules (Schaefer et al., 2010). The core module consists of the implementation of a basic product in the product line. A basic product usually consists of common features required by all product variants. Delta modules (deltas) add, remove, or modify elements in the core module to implement a feature's variant.

A feature model is used to capture commonality and variability in the problem domain, as defined in feature-oriented programming (FOP) (Apel et al., 2013). A feature is implemented by one or more delta modules. DOP provides more flexibility than FOP for implementing product lines, such as the capability to remove an existing source code. A delta module can add, remove, or modify classes, interfaces, methods, and fields in the core modules.

The first language that implements the DOP paradigm is DeltaJava (DeltaJ) (Schaefer et al., 2010; Koscielny et al., 2014). DeltaJ version 1.0 is initially designed to show the feasibility of the DOP approach (Schaefer et al., 2010) and the latest version is DeltaJ 1.5(Koscielny et al., 2014). The core module is implemented in Java languages and represents a basic application in the product line. DeltaJ can modify the Java program by creating delta modules.

Abstract behavioral specification (ABS) is the second language that implements DOP (Johnsen et al., 2012; Hähnle, 2013). Unlike DeltaJ, which modifies a Java program, the delta in ABS modifies the core ABS module. ABS can be generated into programming languages such as Java, Erlang, and Maude.[1] As an executable modeling language, ABS code generators can perform simulation, execution, or visualization.

ABS supports model variability of product lines based on DOP by using textual feature modeling to define the commonality and variability. Based on the feature model, a delta is created to implement the behavior of specific features. A delta can modify existing functionalities without altering the original source code. The ABS configuration language defines relations between features and deltas. A product can be generated based on the feature selection by applying related deltas to the core module.

2.4 Related Work

In this research, the MDSPLE approach is used to develop web applications. The following sections summarize related works about model-driven SPLE and web-based SPLE. Then, the state of the art is analyzed by mapping those works to the problem and solution spaces. Therefore, the current situation is captured, and this research can significantly contribute to the related research field.

Figure 2.2 shows the existing studies about model-driven SPLE (MDSPLE). Most studies have focused more on modeling the problem domain. They used different modeling approaches, such as the common variability language (CVL) (Martinez et al., 2015), the feature model (Czarnecki et al., 2005), and Ouali et al. (2013) combined UML and feature model. Support tools are only defined in (Martinez et al., 2015; Ouali et al., 2013), but those tools are not designed to generate applications. Martinez et al. (2015) developed a tool for automating extraction process, and the tools in (Ouali et al., 2013) focuses on the modeling level.

Existing studies that cover the problem and solution domains in MDSPLE are based on aspect-oriented programming (AOP) (Groher & Völter, 2009; Arboleda & Royer, 2012; Hernández-López et al., 2018). The proposed approach by Groher & Völter (2009) facilitates variability modeling and implementation in problem and solution spaces. The solution spaces are divided into models and code parts. The automation process for

[1]https://abs-models.org/

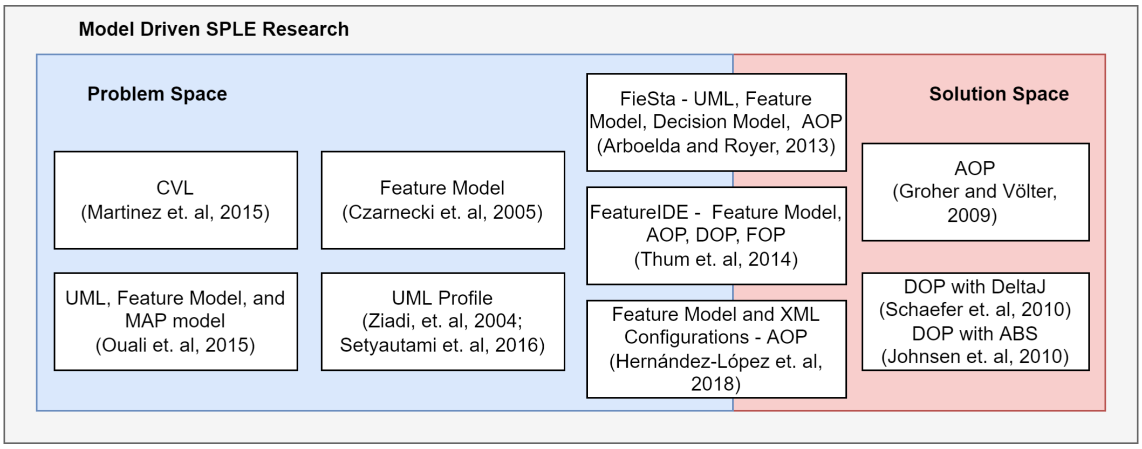

Model Driven SPLE Research
Problem Space
Solution Space
CVL (Martinez et. al, 2015)
Feature Model (Czarnecki et. al, 2005)
UML, Feature Model, and MAP model (Ouali et. al, 2015)
UML Profile (Ziadi, et. al, 2004; Setyautami et. al, 2016)
FieSta - UML, Feature Model, Decision Model, AOP (Arboelda and Royer, 2013)
FeatureIDE - Feature Model, AOP, DOP, FOP (Thum et. al, 2014)
Feature Model and XML Configurations - AOP (Hernández-López et. al, 2018)
AOP (Groher and Völter, 2009)
DOP with DeltaJ (Schaefer et. al, 2010) DOP with ABS (Johnsen et. al, 2010)

Figure 2.2: Model-driven SPLE research

generating applications, proposed by Arboleda & Royer (2012); Hernández-López et al. (2018), used AOP in the solution domain. They have developed a ”*configurator*” to handle a feature selection process. Configurator produces the selected feature and a file XML configuration. As a result, the JAVA application is generated, including the unit test.

MDSPLE is also applied to develop web applications. Model-driven and SPLE improve variability management and systematic reuse in web engineering. A set of web applications in the same domain is developed using MDSPLE to produce a web-based software product line (SPL). A variant of web applications can be generated based on required features.

The state-of-the-art of applying MDSPLE to develop of web-based SPL is shown in Figure 2.3. Existing frameworks to develop a web-based SPL are proposed using different variability modeling approaches, such as CVL (Horcas et al., 2018), the feature model and BPMN (Alferez & Pelechano, 2011b), the decision model (Martinez et al., 2009), and a combination of the UML and the feature model (Nerome & Numao, 2014; Laguna et al., 2009). These studies described the modeling process without showing the generated web applications.

Recent studies by Alferez & Pelechano (2011a); Naily et al. (2018); Aziz et al. (2019); Horcas et al. (2022) demonstrated the process of generating a running web application. Alferez & Pelechano (2011a) used the feature model and UML activity diagram in the problem domain and produced running JAVA web services based on selected features. Framework by Horcas et al. (2022) uses the feature model and multi-language annotations to develop web-based SPL. Naily et al. (2018); Aziz et al. (2019) developed a web framework based on DOP using ABS language to generate a running web application.

A comparison of state-of-the-art approaches in developing a web-based SPL is shown in Table 2.1. All approaches have variability modeling mechanisms in the problem space as

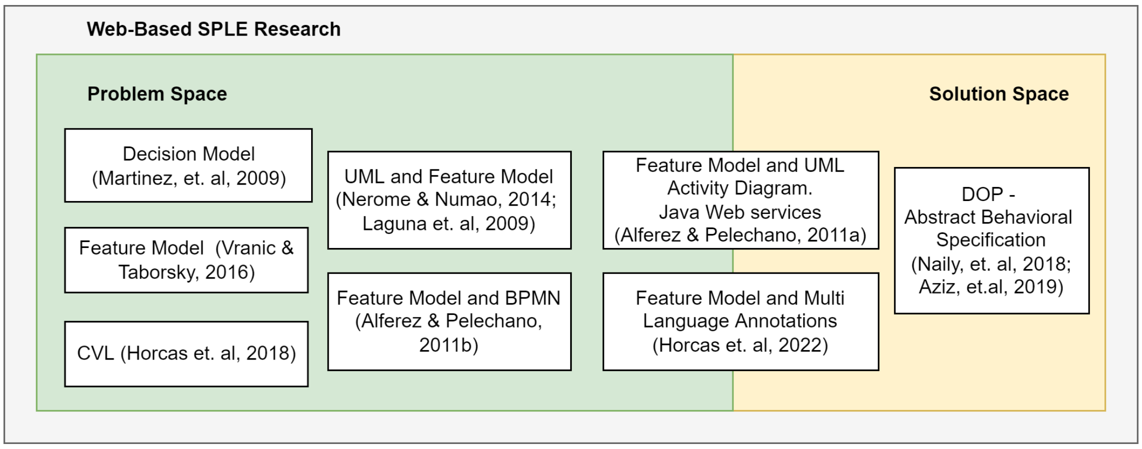

Web-Based SPLE Research
Problem Space
Solution Space
Decision Model (Martinez, et. al, 2009)
Feature Model (Vranic & Taborsky, 2016)
CVL (Horcas et. al, 2018)
UML and Feature Model (Nerome & Numao, 2014; Laguna et. al, 2009)
Feature Model and BPMN (Alferez & Pelechano, 2011b)
Feature Model and UML Activity Diagram. Java Web services (Alferez & Pelechano, 2011a)
Feature Model and Multi Language Annotations (Horcas et. al, 2022)
DOP - Abstract Behavioral Specification (Naily, et. al, 2018; Aziz, et.al, 2019)

Figure 2.3: Web-based SPLE research

defined in SPLE, such as the feature model, decision model, and CVL. Our approach uses a feature model and UML diagram in the problem space. The first novelty is the UML-DOP profile that extends the UML metamodel based on DOP. The UML-DOP profile is utilized in the UML diagram to model variability using delta-oriented notation. UML diagram in Laguna et al. (2009) also uses stereotypes to model variations. However, we define different stereotypes in this research based on DOP.

Table 2.1: A comparison table of state-of-the-art approaches

Approach	**Problem Space**	**Solution Space**	**Tool Support**
Martinez et al. (2009)	Decision Model	N/A	PLUM (Eclipse): modeling and generating a web prototype
Laguna et al. (2009)	Feature Model and UML	.NET and C#	Feature Modeling and Configuration Tool
Nerome & Numao (2014)	Feature Model and UML	N/A	Model Transformation Tool
Alferez & Pelechano (2011a)	Feature Model and UML activity diagram	Java	SALMon, MoRE-WS, and Swordfish to handle the product configuration
Alferez & Pelechano (2011b)	Feature Model and BPMN	N/A	BPMN Model Reconfigurator to generate BPMN from selected features

Vranic & Taborsky (2016)	Feature Model	.NET and C#	Model Transformation Tool
Horcas et al. (2022)	Feature Model	JavaScript	Hybrid Engine to delegate the product generation
Horcas et al. (2018)	CVL	JavaScript	SPL Web Engine to generate web applications
This research	Feature Model and UML	DOP	Prices-IDE (model transformation tool and code generator)

In the solution space, not all approaches show the implementation process of web-based product line development. Laguna et al. (2009); Vranic & Taborsky (2016); Alferez & Pelechano (2011a); Horcas et al. (2018, 2022) explain the implementation process. 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 is designed to implement variability in SPLE. It has good support for feature traceability, an ability to trace a feature from the problem domain to implementation in the solution domain (Apel et al., 2013). In DOP, the relationship between features and their implementation is systematically defined in the *configuration knowledge*. The second novelty in this research is providing a new implementation to developing a web-based SPL. We design an architectural pattern based on DOP using the JAVA module system and design patterns.

As depicted in Table 2.1, most supporting tools are designed to support model transformations or product configuration. The process of generating a running web application is only explained by Alferez & Pelechano (2011a); Vranic & Taborsky (2016); Horcas et al. (2018, 2022). In this research, the supporting tools cover problem space modeling, model transformation, and code generator. A web framework is also defined based on VMJ, called WinVMJ. Therefore, the third novelty is providing a framework and supporting tools to develop a web-based SPL.

CHAPTER 3

RESEARCH METHODOLOGY

This chapter explains research methodology in this research and the evaluation criteria. We also explain the stages conducted in this research, such as scope analysis, design variability model, design domain implementation, Prices-IDE integration, and evaluation.

3.1 Research Methodology

This research uses experimentation to design an integrated framework for a web-based software product line (SPL). Experiments can be viewed as a prototyping process to design an appropriate framework for a web-based SPL. As a new approach, the framework causes process changes in web development. Thus, the framework is completed with tools and an automation process to ease the adoption path.

A case study is used in the experiments to evaluate the framework and improve the tools. Wohlin et al. (2012) stated that a case study is suitable for the industrial evaluation of new methods and tools because it can avoid scale-up problems. Based on the application to the case study, the framework is evaluated using quality criteria defined by Apel et al. (2013). The quality criteria are compiled to assess the product-line implementation techniques. Since this research proposes a new implementation technique for SPLE, the following quality criteria are used for a qualitative evaluation:

1. ***Low Preplanning Effort***. The engineering process in SPLE allows adding a new variability at different stages: before, during, or after the development. *Proactive*, *reactive*, and *extractive* approaches in SPLE have different ways to facilitate the new features. Preplanning is required in the whole product line development process to implement a new required feature. Ideally, preplanning in SPLE aims to minimize the effort to change existing implementation.

2. ***Feature Traceability***. Feature traceability is the ability to trace a feature from the problem domain to the solution domain (Apel et al., 2013). Tracing features in design and implementation is essential to manage the variability and reuse mechanism. A feature can be realized in one more implementation artifacts. A mapping between

features and their implementation is defined to manage traceability.

3. ***Separation of Concerns***. In SPLE, separation of concerns deals with the design and implementation of features and aims to decompose a system into cohesive parts (Apel et al., 2013). Different features can be implemented in distinct artifacts to distinguish the concerns. However, sometimes we can not separate all concerns at the same time. A concern can be scattered across multiple other concerns (*code scattering*) or several concerns can have intermingled representation within a module *(code tangling)*. Therefore, the ability to separate concerns into cohesive implementations is an important quality criterion of product-line implementation techniques.

4. ***Information Hiding***. The key concept of information hiding is to decompose a system into modules or components (Apel et al., 2013). Each module is divided into an internal part, which is hidden from other modules, and an external part (known as an interface), which describes a contract to other modules. The key challenges are designing small and clear interfaces to make communication explicit and maximizing the hidden information.

5. ***Granularity***. A feature implementation can change a program at different levels of granularity. A level of granularity refers to the hierarchical structure of an implementation artifact, defined by a containment relation among the structural elements. A change that involves the addition of a new file to a given program is coarse-grained, adding a new member to a given class is medium-grained, and adding a new statement to a given method body is fine-grained (Apel et al., 2013).

6. ***Uniformity***. Based on the principle of uniformity, a general approach should represent all kinds of artifacts. All artifacts should be treated and managed similarly. Artifacts include code and non-code artifacts, such as requirements, documentation, and architecture designs. Some implementation SPLE techniques are designed in a language-independent manner. Language-independent tools are also designed to enhance uniformity. Therefore, an implementation technique can be applied to various artifacts.

3.2 Research Stage

Based on the research questions in Sect. 1.2, this research is conducted in several stages. The flow of this research is shown in Figure 3.1.

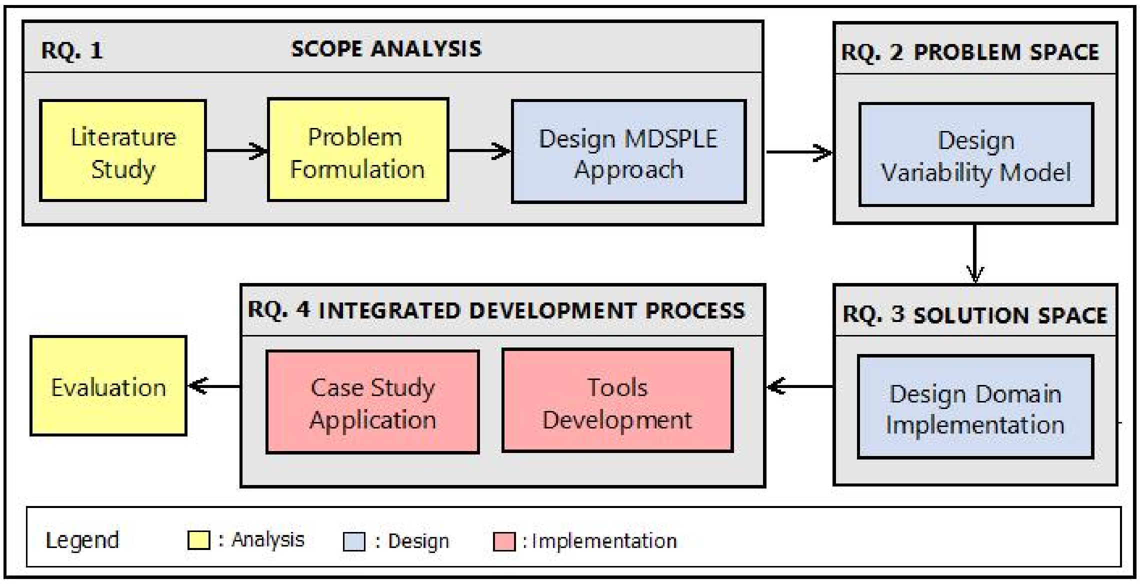


Figure 3.1: Research Flow

1. **Scope Analysis**

   This stage starts with *literature study* to explore theoretical foundations, related works, and state-of-the-art. The summaries of the literature study are explained in Chapter 2. Based on the literature study, the *problem formulation* is conducted to analyze the existing problems. The result is research questions, defined in Sect. 1.2.

   A new *model-driven SPLE (MDSPLE)* approach is designed to solve the problem. The MDSPLE process covers both problem and solution domains. As explained in Section 4.1, the MDSPLE process is supported with tools to support domain and application engineering. We apply the MDSPLE approach to develop a web application that consists of back-end functionality, user interface (front end), and server configuration. We have to analyze which components of the web applications can be modeled as product lines.

2. **Design Variability Model**

   At this stage, we design a unified variability model for the MDSPLE approach. First we use a feature model that captures system variability, as defined in feature oriented software product lines (Apel et al., 2013). The feature model is represented in a feature diagram which shows a graphical representation in a tree structure. Second, a *variability model* is designed based on the UML diagram. The UML diagram is completed with the UML-DOP profile to represent variability at the architectural level. The UML-DOP profile is an extension to model variability in UML (Setyautami et al., 2016). The UML-DOP profile is refined to support the re-engineering of microservice-based applications (Setyautami et al., 2020) and variability modules (VM) concept (Setyautami & Hähnle, 2021).

3. **Design Domain Implementation**
   We use the delta-oriented programming (DOP) approach in the solution space. This research proposes a new *domain implementation* approach called Variability Modules for Java (VMJ). VMJ is designed based JAVA module systems and design patterns to implement SPLE. The detailed definition of VMJ is explained in Chapter 5. WinVMJ framework is then designed based on VMJ to support the web back-end development with VMJ (Prayoga, 2020). For the web front-end, we use an Interaction Flow Modeling Language (IFML) to model the abstract user interface. The IFML meta model is also extended based on DOP to model variations in the user interface.

4. **Integrated Development Process**
   At this stage, the development processes and tools are designed into an integrated development environment, which is explained in Section 6.1. The integration covers feature modeling, UML modeling, user interface modeling, domain implementation, and product generation. The problem domain is mapped to the solution domain using a model-to-model transformation mechanism. As a new domain implementation is introduced, the *model transformation* tool must be developed to support the MDSPLE process. We also demonstrate the practical application using a *case study* by performing domain analysis, implementation, and product generation.

5. **Evaluation**
   We evaluate the MDSPLE approach by analyzing the variability model and the applicability of the UML-VM profile as a foundation of this research. We use evaluation criteria from Apel et al. (2013) that is designed to evaluate SPLE implementation technique. Some scenarios are used to demonstrate the product line development and requirements changes. We evaluate the automated code generation and process improvement in the MDSPLE approach.

CHAPTER 4

MODEL-DRIVEN SPLE

This chapter explains the proposed model-driven SPLE (MDSPLE) approach that is defined based on delta-oriented programming (DOP). The UML-DOP profile is used in the MDSPLE approach to model commonality and variability in the UML diagram. In this chapter, the definition of UML-DOP profile is explained by mapping the DOP elements into UML notations. Short practical examples are also explained in the last (two) sections to show the applicability of the proposed MDSPLE approach.

4.1 MDSPLE Approach

MDSPLE is an application of MDSE to develop a software product line (SPL). This research proposes a new MDSPLE framework based on DOP that covers problem and solution domains. The process in SPLE is divided into *domain engineering* and *application engineering* (Pohl et al., 2005). Apel et al. (2013) separate those two processes into two different perspectives: *problem space* and *solution space*. This separation produces four main tasks in product line development: *domain analysis*, *domain implementation*, *requirement analysis*, and *product derivation*.

We adopt the engineering process for SPLE defined by Apel et al. (2013) to design MDSPLE framework based on DOP. We combine the process in SPLE with model transformation and code generation, two key techniques in MDSE, to connect the problem and solution spaces. The MDSE techniques can be applied in SPLE to automate product generation from the variability model and specific artifacts. Therefore, combining MDSE and SPLE into MDSPLE can enhance the traceability and maintainability of SPLs by establishing clear relationships between the models and the generated code.

The design of MDSPLE based on DOP is shown in Figure 4.1. In *domain analysis*, we use a feature diagram and UML diagrams. These diagrams are transformed into core and delta modules in *domain implementation* since we use DOP to implement SPLE. In *application engineering*, a feature selection process is performed to generate a product variant. This stage is called *requirements analysis* because a new feature can be added to the *domain analysis*. In *product derivation*, a running application is composed based on selected features. The source code is generated from reusable components in *domain implementation*.

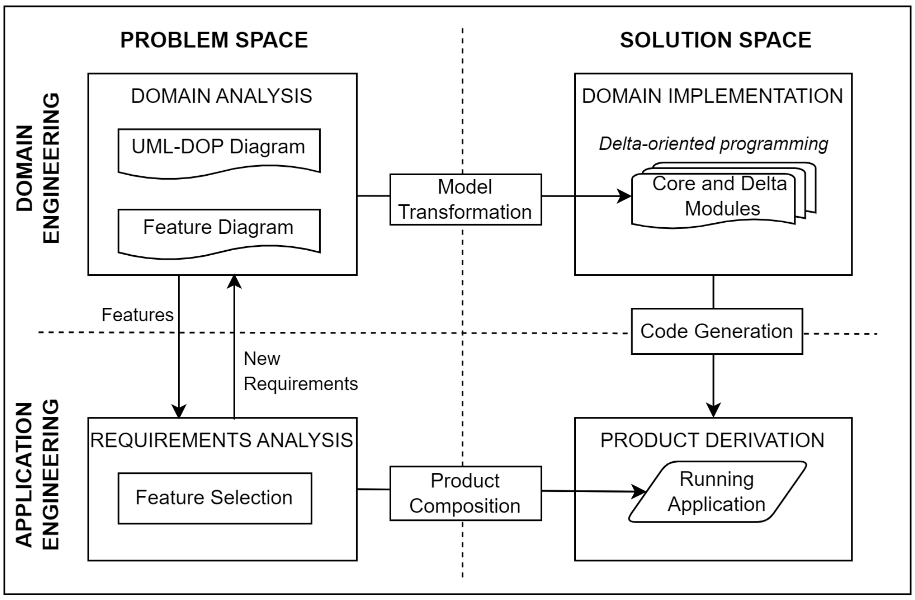


Figure 4.1: MDSPLE design based on DOP, adopted from Apel et al. (2013)

4.2 MDSPLE Framework for Web Development

The MDSPLE based on DOP explained in Section 4.1 can be used to develop SPLs in any domain. In this research, we aim to check the applicability of the MDSPLE approach for web-based product line development. A web application consists of various artifacts with different characteristics and granularities. These artifacts can include components such as user interface (UI) elements, database schemas, server configurations, or business logic modules.

To facilitate the development of web-based product lines, artifacts in web applications must be designed by considering the commonality and variability. However, modeling all artifacts as SPLs can indeed introduce complexity in the feature selection process. When multiple artifacts are treated as product lines, each with its own set of features and variations, the feature selection becomes more complicated and challenging. It requires careful consideration and management of the inter-dependencies, constraints, and interactions among the features across different artifacts.

Based on the complexity consideration, we focus on modeling *functional requirements* of web applications as SPLs. The *functional requirements* define the specific functionalities and capabilities that the application should exhibit to meet the users' needs. Web artifacts related to *non functional requirements*, such as *performance*, *security*, *scalability*, are not modeled as SPLs. Regarding the functional requirements, we need to analyze commonality and variability of the following artifacts:

- Business Logic Modules: web applications can be modularized into different components or modules, such as authentication, payment, and content management. These modules may have common functionality across different applications but also exhibit variations based on specific requirements. We can model the commonality and variability of business logic modules in a feature diagram and UML diagrams. Modeling these modules as an SPL allows for reusing common components and configuring variations based on the desired functionalities.
- UI Elements: UI elements, such as forms, buttons, and menus, can exhibit commonalities and variabilities. For example, different variations of a form may have different fields, validation rules, or styling options. Modeling these components as an SPL allows for systematically managing variations in the user interface. However, variations in UI elements can not be modeled in a feature diagram or UML diagram. We utilize an Interaction Flow Modeling Language (IFML) diagram to model variations in UI elements.
- Data Models: The data models (database schemas) used in web applications often have common entities and relationships but may also exhibit variability based on specific application requirements. Commonality and variability in data models can be represented in UML class diagrams. As we used UML-DOP to model variations in UML diagrams, an adaptive database schema can be generated based on selected features.

The MDSPLE approach for web-based product line development based on DOP is designed to model *functional requirements* as SPLs. It encompasses both the web back end, which includes business logic modules and data models, and the web front end, which includes UI elements. To capture commonality and variability in the web back end, a feature diagram and UML diagrams are utilized in the problem domain. On the other hand, the web front end is modeled using IFML diagrams in the solution domain. By refining the MDSPLE based on DOP in Figure 4.1, an MDSPLE framework for web-based product line development is designed to support web development, as illustrated in Figure 4.2.

The process of web-based product line development adheres to the MDSPLE process outlined in Sections 4.1, with some differences observed in *domain implementation* and

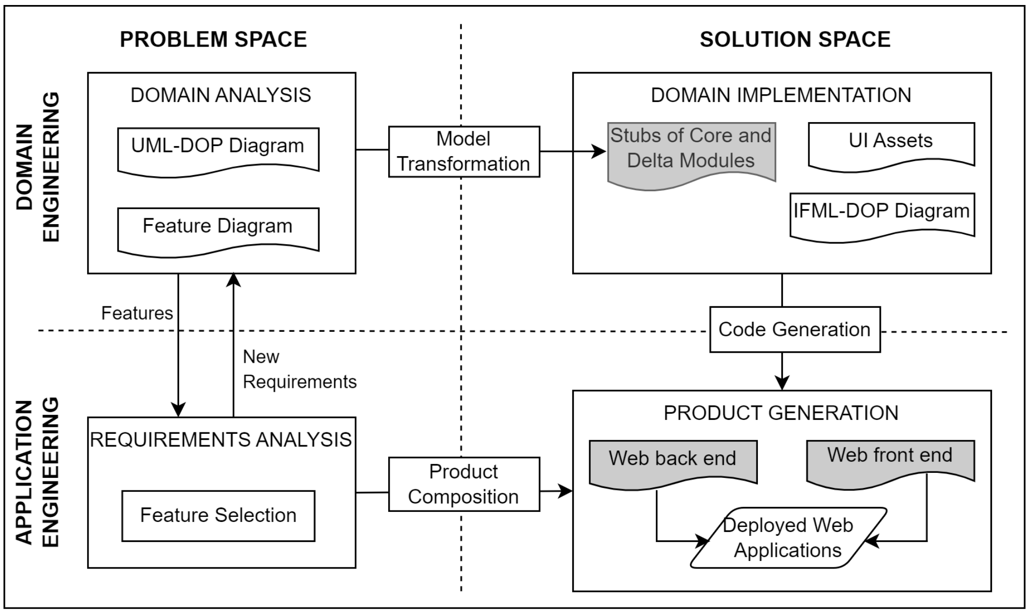


Figure 4.2: MDSPLE framework for web development

product derivation. In *domain implementation*, the development of reusable artifacts is extended to encompass the UI aspect. An IFML diagram is used to model abstract UIs that do not have any styling information, such as color, size, and layout. The UI assets are defined to complement the IFML diagram. In *product derivation* phase, the generated running applications are separated into the web back end and front end. The core and delta modules stubs are reused to generate the web back end, while the IFML diagram and the UI assets are combined to generate the web front end.

4.3 The UML-DOP Profile

UML has a mechanism to extend the metamodel by defining a UML profile as a standard modeling language. In this research, a UML profile is defined to model product line variations. As we use DOP to implement SPLE, the UML profile is defined based on DOP, called UML-DOP profile. The UML-DOP profile is created by defining UML stereotypes, constraints, and tagged values based on DOP elements. In this proposed MDSPLE approach, the UML-DOP profile is the main foundation because a UML diagram with the UML-DOP profile (UML-DOP diagram) is used in the domain analysis.

Initially, the UML profile is defined based on DOP language, abstract behavioral specification (ABS) in Setyautami (2013). The UML-ABS profile was defined specific to ABS language without considering DOP in general. Then, the UML-ABS profile is revised to a UML profile for delta-oriented programming (UML-DOP) in Setyautami et al. (2016).

The UML-DOP profile was extended to support multi-product lines (Setyautami et al., 2018; Setyautami & Hähnle, 2021) and microservice-based product lines (Setyautami et al., 2020). As a result, table 4.1 shows the mapping of DOP elements to UML stereotypes.

Table 4.1: Mapping of DOP elements to UML stereotypes

DOP Element Name	UML Base Class	Stereotype Name
Core Module	Package	`<<module>>`
Import	Dependency	`<<import>>`
Class Parameter	Property	`<<classParam>>`
Delta Module	Package	`<<delta>>`
Delta Parameter	Property	`<<deltaParam>>`
Module Access	Dependency	`<<uses>>`
Module Modifier	Association	`<<adds>>`
		`<<removes>>`
		`<<modifies>>`
	Class	`<<addedClass>>`
		`<<removedClass>>`
		`<<modifiedClass>>`
	Interface	`<<addedInterface>>`
		`<<removedInterface>>`
		`<<modifiedInterface>>`
Class Modifier	Operation	`<<adds>>`
		`<<removes>>`
		`<<modifies>>`
	Property	`<<adds>>`
		`<<removes>>`
Feature	Component	`<<feature>>`
Require	Dependency	`<<require>>`
Exclude	Dependency	`<<exclude>>`
Product	Component	`<<product>>`
When	Dependency	`<<when>>`
After	Dependency	`<<after>>`
Variability Module	Package	`<<vm>>`
Service	Component	`<<service>>`
Endpoint	Component	`<<endpoint>>`

Based on the UML base class (metaclass) from Table 4.1, DOP elements can be modeled in UML by using several UML diagrams, such as class diagram, package diagram, and component diagram. For example, metaclasses class, association, operation, and property are elements of UML class diagram. Thus, the stereotypes that extend those metaclass, such as `<<classParam>>`, `<<deltaParam>>`, `<<removes>>`, `<<modifies>>`, and `<<adds>>`, can be modeled in UML class diagram. Those stereotypes represent DOP elements that will exist in UML class diagram.

4.4 The IFML-DOP Extension

In this research, the MDSPLE approach is designed based on DOP to develop a web-based SPL. As defined in the MDSPLE process in Figure 4.2, Interaction Flow Modeling Language (IFML) is used to model the abstract user interface (UI). An IFML diagram supports the UI specification without considering the technological details of their implementation (Brambilla & Fraternali, 2015). The front end of web applications, such as the composition (structure) of the view, the content of the view, the interaction (event), the data flow (input-output), and the actions, can be modeled in the IFML diagram. The IFML metamodel is defined based on the OMG standards using the OMG Meta Object Facility (OMG, 2015).

An IFML diagram for feature *Income* is shown in Figure 4.3. Based on the requirements, a user can see a list of income on the website and also add new income data. In the IFML diagram, these requirements are modeled as two different `ViewContainers`: *Income* and *New Income*. A `ViewContainer` models a page or structure in the web application. In the `ViewContainer` *Income*, there is a `ViewComponent List` that shows a list of income data. The displayed attribute can be modeled in the `List` using `VisualizationAttribute`. The data in `VisualizationAttribute` can refer to fields in the UML class diagram or the database. This reference can be represented in the `DataBinding` component.

An `event` in IFML, represented as a circle, is an occurrence that affects the state of the UI. In Figure 4.3, there is an `event` *Create* in the `ViewContainer` *Income*. The effect of this event is represented by the arrow `InteractionFlow`. An `InteractionFlow` connects the *event* to the `ViewContainer` or `ViewComponent` affected by the *event*. In Figure 4.3, if the `event` *Create* is clicked, the event causes the display of `ViewContainer` *NewIncome* which consists of a `form` *AddIncome* to add new income data. An *event* can also trigger an *action*, represented by a hexagon. If a user click *event* `Submit`, an *action* `Save` is triggered to save the filled data.

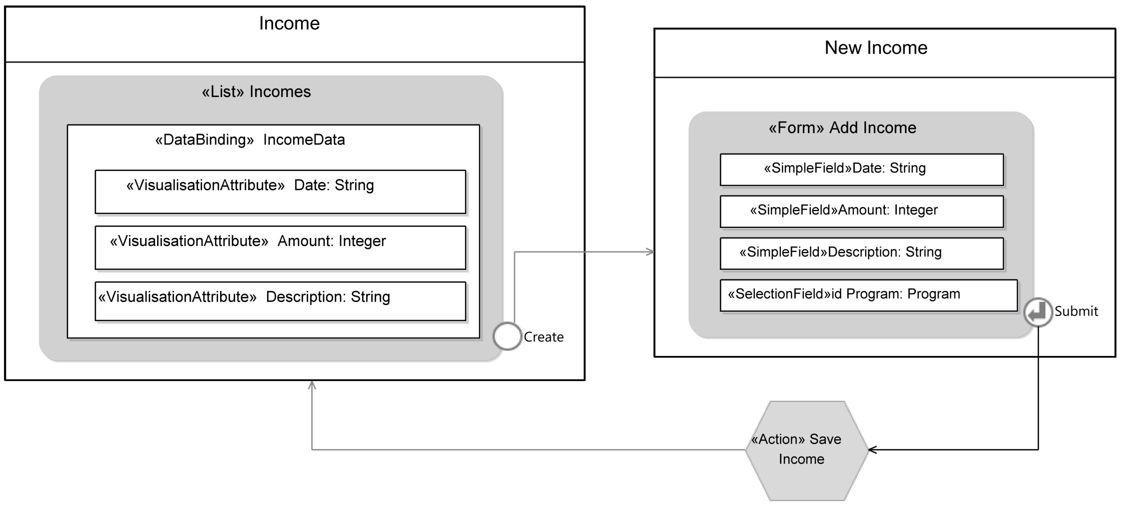
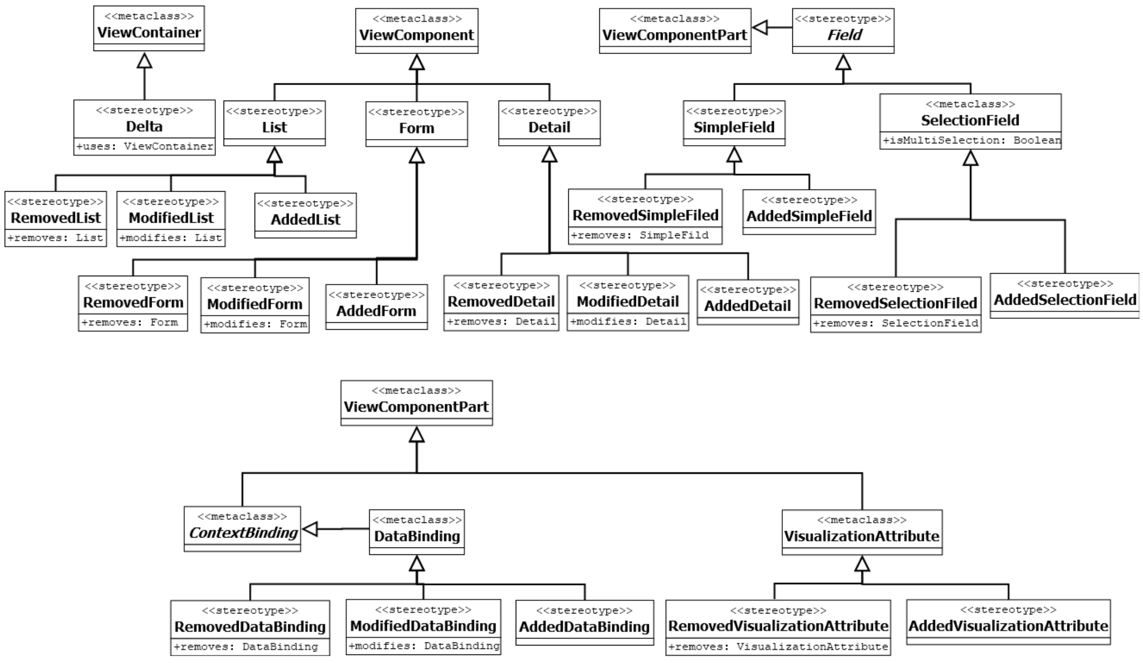

Income
«List» Incomes
«DataBinding» IncomeData
«VisualisationAttribute» Date: String
«VisualisationAttribute» Amount: Integer
«VisualisationAttribute» Description: String
Create
New Income
«Form» Add Income
«SimpleField»Date: String
«SimpleField»Amount: Integer
«SimpleField»Description: String
«SelectionField»id Program: Program
Submit
«Action» Save Income

Figure 4.3: IFML Diagram: Income

Modeling commonality and variability is an essential process in SPLE. In Section 4.3, the UML metamodel is extended, by defining the UML-DOP profile, to support variability modeling in the UML diagram. The UML-DOP profile is designed based on DOP by mapping the DOP elements to UML stereotypes. The DOP paradigm is not designed to develop the UI of applications. Thus, the UML diagram with the UML-DOP profile is only used to model the back end of a web-based SPL. Fadhlillah et al. (2018) proposed to use IFML to model abstract UI in web-based SPL development. Since the IFML is not intended to model UI variations, the IFML model is designed without considering commonality and variability.

Based on the IFML modeling process, some features might have similar UI but existing IFML model cannot be reused to model different features. For example, features *Expense* and *Income* requires the same IFML elements to model the web UI with different displayed data. As a result, we have two similar IFML models for *Income* and *Expense*. In DOP, the common parts are defined as core modules, and the variants are defined as delta modules. If the IFML diagram can be modeled as core and delta modules, the UI variations can be managed systematically.

IFML is allowed to be extended for a specific purpose. Existing studies by Brambilla et al. (2014); Hayat et al. (2021); Roubi et al. (2016); Shahin & Zamani (2021) extends IFML to model various domains. Brambilla et al. (2014) extends IFML to model mobile applications because IFML does not cover mobile-specific aspects and interactions. Hayat et al. (2021) extends IFML for modeling Geographical Information System, Shahin & Zamani (2021) extends IFML for Form Making System, and Roubi et al. (2016) extends IFML for Rich Internet Graphical UI. In this research, we extend the IFML model based on DOP to model UI variations. The following subsections explain the IFML-DOP extension based on `ViewContainer`, `ViewComponent`, and `ViewComponentPart`, as shown in Figure 4.4



<<metaclass>> ViewContainer
<<stereotype>> Delta +uses: ViewContainer
<<metaclass>> ViewComponent
<<stereotype>> List
<<stereotype>> Form
<<stereotype>> Detail
<<stereotype>> RemovedList +removes: List
<<stereotype>> ModifiedList +modifies: List
<<stereotype>> AddedList
<<stereotype>> RemovedForm +removes: Form
<<stereotype>> ModifiedForm +modifies: Form
<<stereotype>> AddedForm
<<stereotype>> RemovedDetail +removes: Detail
<<stereotype>> ModifiedDetail +modifies: Detail
<<stereotype>> AddedDetail
<<metaclass>> ViewComponentPart
<<stereotype>> Field
<<stereotype>> SimpleField
<<metaclass>> SelectionField +isMultiSelection: Boolean
<<stereotype>> RemovedSimpleFiled +removes: SimpleFild
<<stereotype>> AddedSimpleField
<<stereotype>> RemovedSelectionFiled +removes: SelectionField
<<stereotype>> AddedSelectionField
<<metaclass>> ViewComponentPart
<<metaclass>> ContextBinding
<<metaclass>> DataBinding
<<metaclass>> VisualizationAttribute
<<stereotype>> RemovedDataBinding +removes: DataBinding
<<stereotype>> ModifiedDataBinding +modifies: DataBinding
<<stereotype>> AddedDataBinding
<<stereotype>> RemovedVisualizationAttribute +removes: VisualizationAttribute
<<stereotype>> AddedVisualizationAttribute

Figure 4.4: IFML-DOP Extension

CHAPTER 5

VARIABILITY MODULES FOR JAVA

This chapter explains about Variability Modules for Java (VMJ), an architectural pattern to implement multi product lines (MPL) in JAVA.

5.1 Variability Module

Variability modules (VM) is an extension of a software module system that captures variability at the level of modules (Damiani et al., 2021). VM is designed to solve interoperability problems in product line variants. Multiple variants from a similar product line sometimes cannot co-exist 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.

Each VM represents a product line, and product lines can be considered as modules. A VM consists of a *module header*, a *core module*, and an optional *delta module*. A module header contains a module, optional import/export dependencies referring to other module cores, and optional feature declarations. A *core module* is a standard module with classes, interfaces, fields and methods. In addition, it allows to declare products.

Since in VM deltas are uniformly represented as class operations, the DOP aspects are straightforward: The *core module* contains *deltas* and *configuration knowledge*. The latter is as usual in DOP. Deltas may contain operations that specify their interfaces and classes by adding, removing, and modifying such elements. VM concept is realized in the ABS modeling language that supports SPLE (Damiani et al., 2021). The implementation VM for ABS is integrated into the ABS compiler tool chain[1].

5.2 Architectural Pattern in Java

In this research, we designe an architectural pattern to realize the VM concept in JAVA, called Variability Modules for Java (VMJ). The VM concept is combined with the JAVA

[1]https://github.com/abstools/abstools/tree/local_productlines

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 MPL. VMJ is more intuitive to use for anyone familiar with the JAVA programming language.

As VMJ is designed in the JAVA programming language, the problem is how to manage and implement variations in JAVA. Figure 5.1 illustrates the problem and solution mapping. We use the AMANAH case study as an illustration in the problem space. AMANAH consists of several features, such as *Program*, *FinancialReport*, and *Donation*. A rectangle denotes a mandatory feature (core module), and an ellipse denotes an optional feature (delta module). For example, three products are generated, i.e. *CharitySchool*, *Hilfuns*, and *YayasanPandhu*.

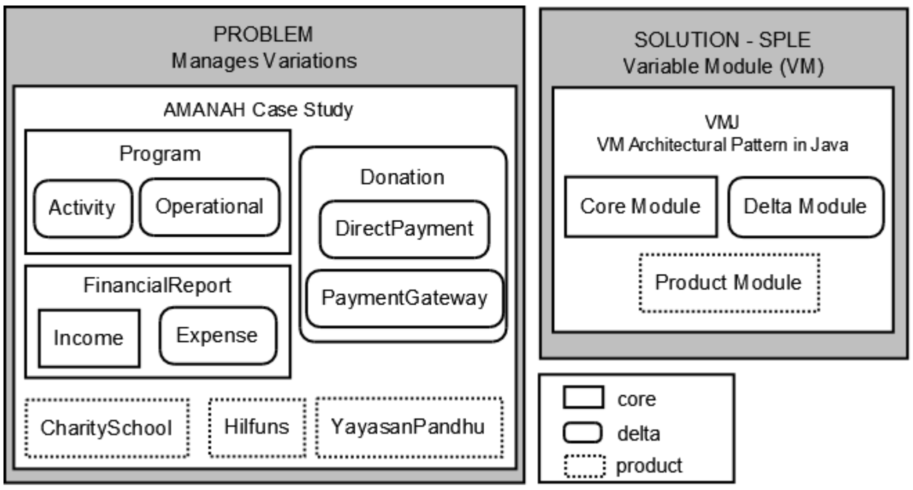


Figure 5.1: Problem Solution Mapping
Source: Setyautami & Hähnle (2021)

In the solution space, VMJ architectural pattern is designed to manage variation in JAVA. A product line in VMJ is represented as a JAVA project consisting of JAVA modules. As defined in VM, JAVA modules can be classified into the core, delta, and product modules. Each kind of module has a module declaration (header) specifying its name and dependencies. The structure of the VMJ is defined as follows:

- A VM is defined as a JAVA project that consists of several modules.
- A *core module* is defined as a JAVA module that consists of packages with common capabilities to be reused anywhere they are declared.
- A *delta module* is defined as a JAVA module that modifies a core module. A concrete decorator class is defined in the delta module.
- A *product* variant is defined as a JAVA module. A product consists of a list of selected features. So, the product module has a dependency on core modules and

delta modules.

The schema of the VMJ architectural pattern is shown in Figure 5.2. JAVA modules are managed using design patterns to derive product variants. A delta module decorates a core module by specifying the modification behavior. A product module configures objects based on the selected features. By defining java modules dependency, a product module can access functionality in the selected features. Objects in the core modules and delta modules are created using the factory pattern. The factory pattern also manages the delta application order if a feature is implemented by more than one delta.

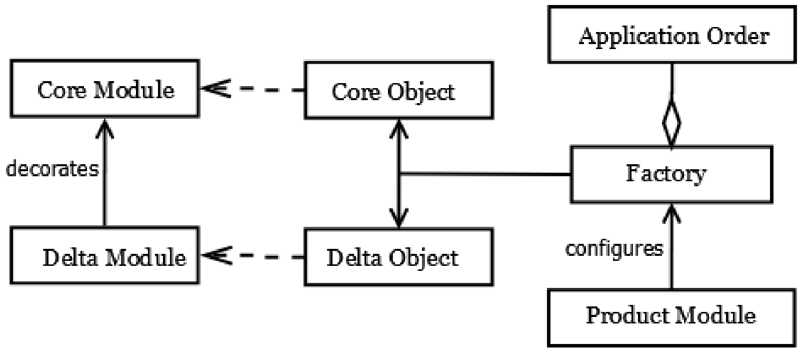


Figure 5.2: Schema of VMJ Architectural Pattern
Source: Setyautami & Hähnle (2021)

5.2.1 Decorator Pattern

In DOP, variability is implemented in the delta modules. Delta operations can add, remove, or modify existing classes, methods, or fields in the core modules. These operations are applied to the core modules without changing the source code directly. Based on the feature selection, the behavior in the core modules is changed by delta application. Managing variability in SPLE is not supported by JAVA languages. Inheritance is not suitable to model delta behavior because sub-classing is made statically. Therefore, in this research the decorator pattern is used to model delta's behavior.

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 can be maintained. For example, there are two kinds of *Financial Report*, *Income* and *Expense*. *Income* has additional a new field `<<paymentMethod>>`. Both *Income* and *Expense* modify method `total()` in the core module. Figure 5.3 shows the application of the decorator pattern to *FinancialReport* module.

To obtain variants of *FinancialReport*, the pattern is set up with interface `FinancialReport` and *abstract component class* `FinancialReportComponent`. The *concrete component class* `FinancialReportImpl` implements the *core module* by extending

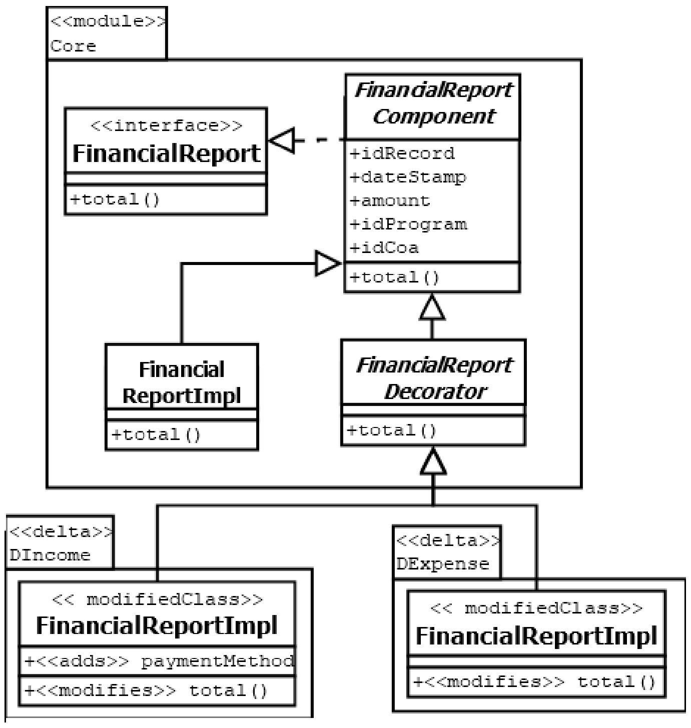


Figure 5.3: Applying the decorator pattern

the *abstract component class*. This *concrete* component class provides an implementation for all abstract methods from the *abstract* class. Variant behavior is handled by the abstract *decorator class* `FinancialReportDecorator` extending `FinancialReportComponent`. *Concrete* decorators are created to implement specific behavior of each delta. For example, class `FinancialReportImpl` in the `<<delta>>` package `income` implements feature *Income*.

The naming convention is used to model VM as JAVA modules. The module name consists of three parts: `<productline-name>.<feature-name>.<module-name>`. A core module name ends with `core`. For example, in Listing 5.1 module `aisco.financialreport.core` represents a *core module* of *FinancialReport*. A module `aisco.financialreport.income` in Listing 5.3 represents delta *income* that modifies *FinancialReport*.

```java
package aisco.financialreport.core;
public abstract class FinancialReportComponent implements FinancialReport {
  protected String idRecord;
  protected String dateStamp;
  protected int amount;
  ...
  public abstract int total(List<FinancialReport> records};
}
```

Listing 5.1: Financial Report: Component Class

Listing 5.1 is a snippet of the JAVA code of abstract class `FinancialReportComponent` implementing the `FinancialReport` interface (not shown). The *component class* consists of several fields and abstract method. Listing 5.2 shows class `FinancialReportImpl` that extends the *component class*, and implements the `total()` method. Class `FinancialReportComponent` and `FinancialReportImpl` form the *core module* representing commonality in *FinancialReport*.

```
package aisco.financialreport.core;
public class FinancialReportImpl extends FinancialReportComponent {

  public int total(List<FinancialReport> records) {
    int sum = 0;
    for(int i = 0; i < records.size(); i++) {
      int amount = (records.get(i)).getAmount();
      sum = sum + amount; }
    return sum;
}}
```

Listing 5.2: Financial Report: Implementing Class

The core module implementation of `total()` sums up the `FinancialReport`. Its different variants print either total income or expense. These features are realized in different delta modules. For example, feature *Income* prints the total income with fees. A delta module `DIncome` is created with additional fields and modified `total()`. As shown in Figure 5.3, this is achieved with a *decorator class*, see Listing 5.3. Another variant of *Financial Report*, with feature *Expense*, can be implemented as a further *decorator class* in analogous manner. Delta application is performed by creating a class instance with the *Factory* design pattern in Gamma et al. (1994).

```
package aisco.financialreport.income;
public class FinancialReportImpl extends FinancialReportDecorator { // decorator class
  private String paymentMethod;    // delta adds fields
  public FinancialReportImpl(FinancialReportComponent record, String paymentMethod) {
    super(record);
    this.paymentMethod = paymentMethod; }

  public int total(List<FinancialReport> incomes) {    // delta modifies method
    int sum = record.total(incomes); // original() call
    int fee =  (int) adminfee(incomes);
    System.out.println("Income: "+sum+"\n Fee: "+fee);
    return (sum-fee);
}}
```

Listing 5.3: Delta Income

5.2.2 Factory Pattern

Based on Figure 5.3, *core module* `FinancialReport` is modified by deltas `DIncome` and `DExpense`, which modify method `total()` in different ways. This condition raises ambiguity in standard DOP, because we cannot specify which modification should be applied. As a result, a product that consists of features *Income* and *Expense* raises a run time error. VM concepts provide a mechanism that both variants can co-exist without resulting in ambiguous calls to method `total()`.

In VMJ, the factory pattern is used to choose the appropriate variant during product generation. The pattern allows creating groups of related objects without specifying their concrete class Gamma et al. (1994). For example, in the `aisco.financialreport` package, a class `FinancialReportFactory` is created. This class is responsible to create objects from `FinancialReportImpl` in different packages: `core`, `income`, and `expense`. Therefore, the factory pattern encapsulates which FinancialReport's variant to create and let the user choose during the product generation.

```
1  package aisco.financialreport;
2  public class FinancialReportFactory {
3    ...
4    public static FinancialReport createFinancialReport(String fullyQualifiedName, Object... base) {
5      FinancialReport record = null;
6      ...
7      Class<?> clz = Class.forName(fullyQualifiedName);
8      Constructor<?> con=clz.getDeclaredConstructors()[0];
9      record = (FinancialReport)con.newInstance(base);
10     return record;
11 }}
```

Listing 5.4: Factory class `FinancialReport`

Listing 5.4 shows The *factory class* `FinancialReport`. In line 4, a *creator* method `createFinancialReport()` is declared with two parameters. The first parameter is the fully qualified name that specifies which class `FinancialReportImpl` is created. The second parameter is a *variable length argument* `Object... base` that can be filled by zero or more fields. Since delta can add or remove fields, a *variable length argument* allows flexibility of the number of field. In lines 7–9, JAVA's reflection API is used to determine the appropriate constructor method based on the fully qualified name.

5.3 UML-VM Profile

In Setyautami et al. (2016), a UML profile is defined based on DOP, called the UML-DOP profile. It realizes a one-to-one mapping between DOP elements to the UML model. The advantage is that a deterministic transformation from UML model elements to DOP languages (and vice versa) can be achieved (Muhammad & Setyautami, 2016; Setyautami et al., 2019). The UML-DOP is extended to the UML-VM profile for modeling VM, which supports MPL and interoperability in the product line. The UML-VM profile is defined by modeling VM elements from the metamodel using UML notation.

The UML profile can be represented as UML profile diagram. The profile diagram for UML-VM profile is shown in Figure 5.4. The profile diagram shows all stereotypes and their extending metaclasses. For example, VM, core module, delta, are represented as a UML package with different stereotypes `<<vm>>`,`<<module>>`, and `<<delta>>`. Specific attributes in the stereotypes can be defined in tagged values, see three tagged values in stereotypes `<<feature>>`.

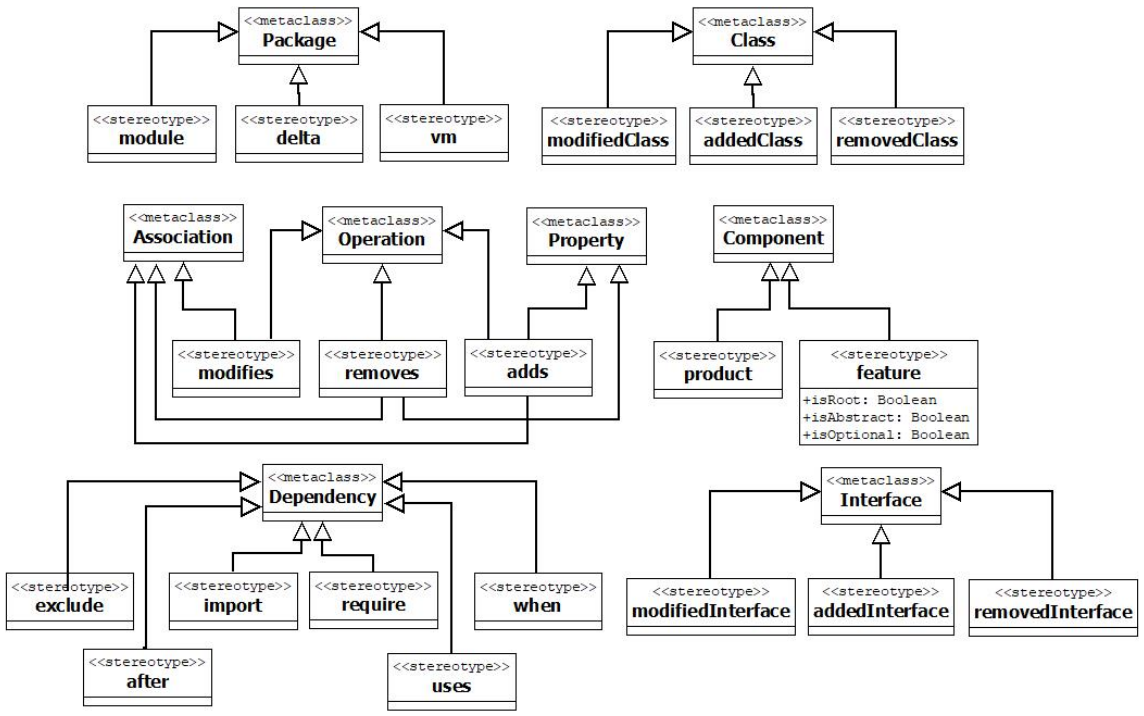


Figure 5.4: UML-VM profile diagram

CHAPTER 6

PRICES-IDE

In this chapter, we explain Prices-IDE, an integrated development environment (IDE) to develop a web-based software product line (SPL). This chapter also provides a realization and practical application of the MDSPLE approach. We describe the design of Prices-IDE in the first section. The following sections explain supporting tools in Prices-IDE that are deployed as Eclipse plugins. In the last section, we show a practical application of the MDSPLE process with Prices-IDE using a case study.

6.1 Prices-IDE Design

PRICES is a framework to develop a web application using software product line engineering (SPLE) (Setyautami et al., 2021). This framework is supported by several tools, such as model transformation tools, back end generator, and user interface generator. The ABS language is used to model the back end based on DOP. In this research, we change the ABS language to variability modules for Java (VMJ). The first reason is that VMJ solves interoperability problems in the SPL that exist in the ABS web framework. Second, VMJ supports multi-product line (MPL) development. Third, VMJ utilizes standard JAVA modules system and design patterns so that the developers do not need to learn new languages to develop a web-based SPL. As VMJ is designed based on VM and DOP, we can use the UML-VM profile in the UML diagram.

The MDSPLE process with VMJ is shown in Fig. 6.1. The process is started with the modeling phase with the diagram editor. A UML-VM diagram is used to model the domain design, a feature diagram is used to model the domain requirements (feature variations), and an IFML diagram is used to model the abstract UI. Those diagrams are transformed into the domain implementation using model transformation tools. As a result, the source codes of the product line back end and front end are generated. A running web application is derived from the user's feature selection.

Based on the MDSPLE process in Fig. 6.1, we redesign PRICES tools into an integrated development environment (IDE), called Prices-IDE. We choose Eclipse, an open-source platform, to develop interoperable PRICES tools for developers. We utilize Eclipse

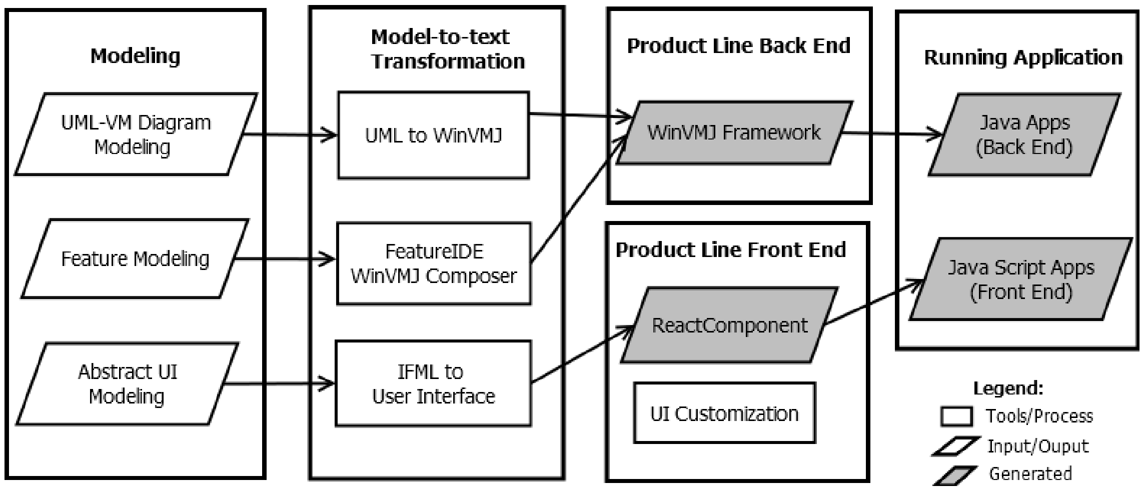


Figure 6.1: MDSPLE Process - VMJ

Modeling Framework (EMF) that provides modeling and code generation facility for building model-based applications.

In Eclipse, PRICES tools are deployed as Eclipse plugins and these plugins can be installed without dealing with infrastructure or integration issues. The structure of Prices-IDE framework is shown in Figure 6.2. This framework is designed based on the foundation in the yellow boxes. The foundation is a new conceptual contribution in this research, such as UML-VM profile that allows modeling variability modules (VM) in UML. The contributions are applied to practical implementation in Prices-IDE plugins and web framework. Dotted arrows show the dependencies from practical implementations to theoretical foundations. For example, the UML to WinVMJ tool is designed and developed based on UML-VM profile.

Prices-IDE plugins are developed in Eclipse Modeling tools, version 2020-12[1]. These plugins are deployed in the following update sites:

1. WinVMJ Composer: `https://amanah.cs.ui.ac.id/priceside/winvmj-composer/updatesite/`
2. UML to WinVMJ Tool: `https://amanah.cs.ui.ac.id/priceside/uml-to-winvmj/updatesite/`
3. IFML UI Generator: `https://amanah.cs.ui.ac.id/priceside/ifml-ui-generator/updatesite/`

[1]`https://www.eclipse.org/downloads/packages/release/2020-12/r/eclipse-modeling-tools`

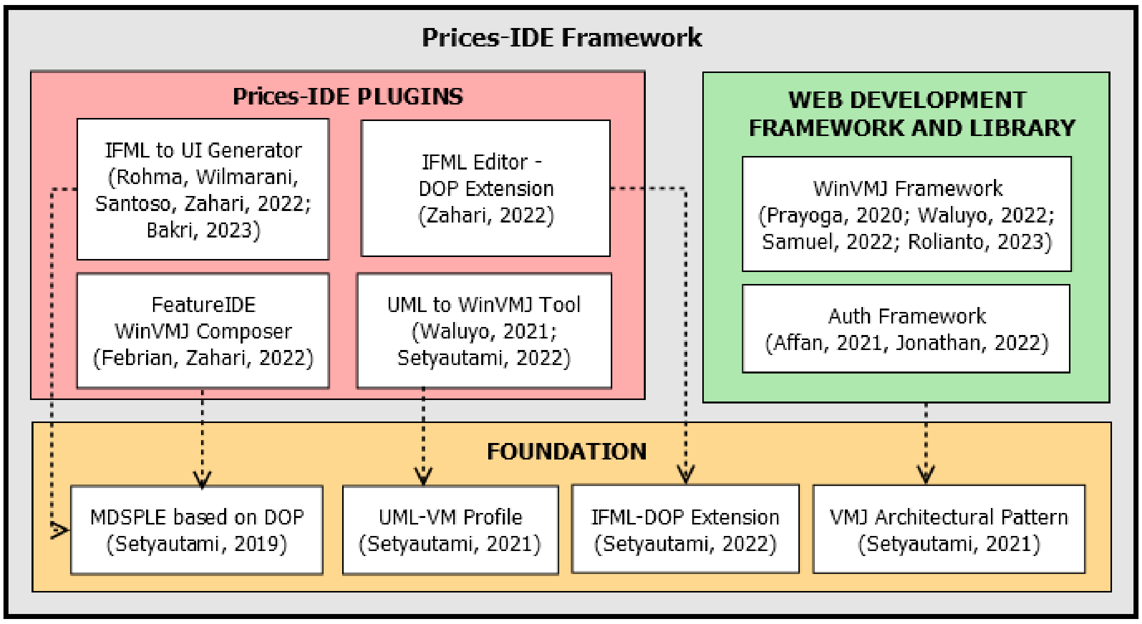

Prices-IDE Framework
Prices-IDE PLUGINS
IFML to UI Generator (Rohma, Wilmarani, Santoso, Zahari, 2022; Bakri, 2023)
IFML Editor - DOP Extension (Zahari, 2022)
FeatureIDE WinVMJ Composer (Febrian, Zahari, 2022)
UML to WinVMJ Tool (Waluyo, 2021; Setyautami, 2022)
WEB DEVELOPMENT FRAMEWORK AND LIBRARY
WinVMJ Framework (Prayoga, 2020; Waluyo, 2022; Samuel, 2022; Rolianto, 2023)
Auth Framework (Affan, 2021, Jonathan, 2022)
FOUNDATION
MDSPLE based on DOP (Setyautami, 2019)
UML-VM Profile (Setyautami, 2021)
IFML-DOP Extension (Setyautami, 2022)
VMJ Architectural Pattern (Setyautami, 2021)

Figure 6.2: Prices-IDE Plugins

4. IFML-DOP Editor: `https://amanah.cs.ui.ac.id/priceside/ifml-dop-editor/updatesite/`

6.2 UML to WinVMJ Tool

The UML diagram with UML-VM profile (UML-VM) diagram is used in the domain analysis to model the web-based product line back end. The UML diagram is designed with Eclipse Papyrus[2] modeling environment. Eclipse Papyrus is an integrated tool to model the UML diagram as defined in the OMG specification. The UML diagram in Papyrus has an XMI representation and a graphical notation. We can refer to a specific version of UML metamodel in Eclipse to design a UML diagram. Eclipse Papyrus also provides support to extend the UML diagram by defining the UML profile. We design the UML-VM profile in the Eclipse Papyrus and then apply the profile to the UML-VM diagram.

The UML to WinVMJ tool is developed to transform the UML-VM diagram into WinVMJ source code. WinVMJ is a web framework based on DOP VMJ and VMJ architectural pattern. As defined in VMJ, WinVMJ uses JAVA module systems and design patterns. Support libraries are also developed using JAVA modules to produce a running web-backend application. The architecture of the WinVMJ framework is separated into three layers to distinguish the development concerns (Prayoga, 2020; Waluyo, 2022).

[2]https://www.eclipse.org/papyrus/

The transformation rules for the UML to WinVMJ tool are defined based on the UML-VM profile. The UML-VM profile maps VM elements to the UML stereotypes so that the stereotypes are key information in the transformation process. The transformation rules are summarized in Table 6.1. Each product line is modeled as a UML package with stereotype `<<vm>>`. This UML package is transformed into a JAVA project, as defined in Rule R1.

As defined in the VMJ, there are three kinds of JAVA modules in WinVMJ: core modules, delta modules, and product modules. In UML these modules are represented as UML packages with different stereotypes, whereas in JAVA these modules are implemented as JAVA modules. In the UML to WinVMJ tool, we only generate implementation of core and delta modules because the product module is generated by WinVMJ Composer tool. In Table 6.1, we separate the rules for core and delta modules.

Table 6.1: Transformation Rules

Rule	**UML Base Class**	**Stereotype Name**	**VMJ Source Code**
R1	Package	`<<vm>>`	Java Project
	Core Module		
R2	Package	`<<module>>`	Java (core) module
R3	Interface	–	Java interface
R4	Class	–	Abstract component class Abstract decorator class Concrete component class Factory class
	Delta Module		
R5	Package	`<<delta>>`	Java (delta) module

R6	Interface	`<<addedInterface>>`
		`<<modifiedInterface>>`
			Java interface
			Java interface

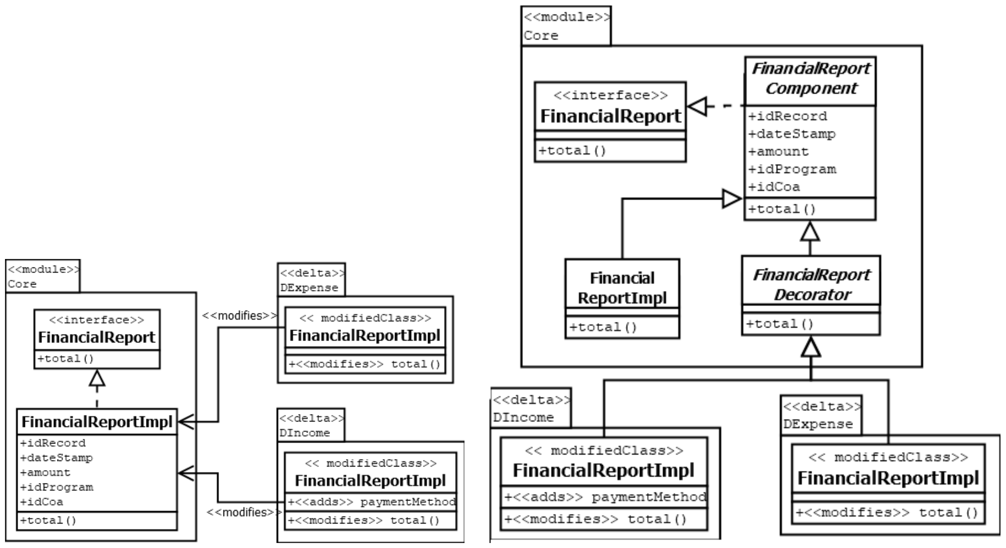
R7	Class	`<<addedClass>>`
		`<<modifiedClass>>`
			Concrete decorator class
			Concrete decorator class

R8	Attribute	`<<adds>>`	Field in concrete decorator class

R9	Operation	`<<adds>>`
		`<<modifies>>`
			Method in concrete decorator class

We illustrate the idea of transformation rules by using a UML diagram in Figure 6.3. The input of the transformation tool is the UML-VM diagram in Figure 6.3a, and the output is VMJ source code which is illustrated in the UML diagram decorator in Figure 6.3b.

As shown in Table 6.1, the transformation rules are categorized into a core module (Rule R2-R4) and a delta module (Rule R5-R9). A UML package with stereotype `<<module>>` is transformed into a JAVA core module (Rule R2) and a UML package with stereotype `<<delta>>` is transformed into a JAVA delta module (Rule R5).



(a) UML-VM diagram **(b)** UML diagram decorator

Figure 6.3: UML transformation

The UML to WinVMJ tool is implemented in Eclipse using Acceleo Model to Text Transformator (M2T)[3]. Acceleo is a template-based approach to develop custom code generators. Acceleo transforms input models and code templates into generated source code. The Acceleo language, namely Model to Text Language (MTL), conforms to the OMG's Meta-Object Facility (MOF), a standard for model-driven engineering. It uses any EMF based models such as UML, SysML, or domain-specific models. An Acceleo MTL module (.mtl file) consists of two main structures: templates and queries. The templates are used as code skeletons to generate code, and the queries extract information from the input models.

The input of UML to the WinVMJ tool is a UML-VM diagram, and the output is WinVMJ source code. Figure 6.4 shows the package diagram of the UML to WinVMJ tool implemented in Eclipse Acceleo. To improve the reusability, we create util and services packages for common functions. When the UML to WinVMJ tool runs, the plugin calls the main package. The main package transforms the UML model by calling the code generators in the other packages. As shown in Figure 6.4, the main package depends on the following packages:

1. Package `modulegenerator`

[3]https://www.eclipse.org/acceleo/

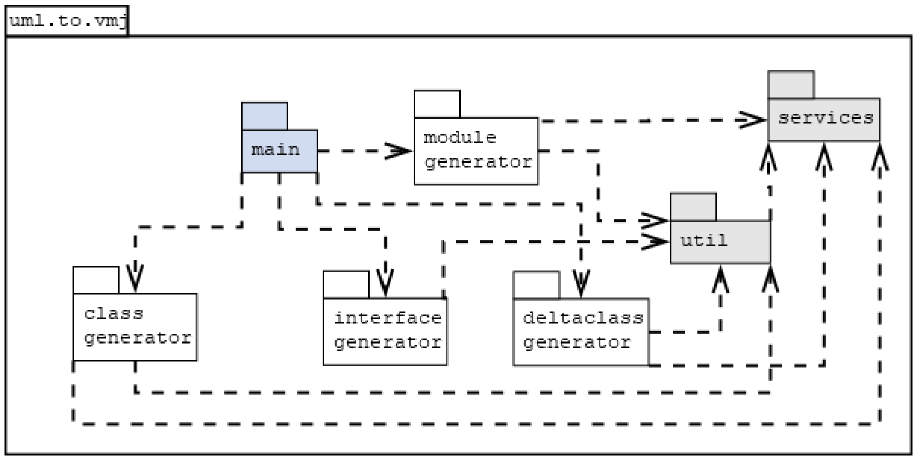


Figure 6.4: Package diagram - UML to WinVMJ tool

Package `modulegenerator` handles a transformation for UML packages in the UML-VM diagram. As defined in the transformation rules, a UML package with stereotype `<<module>>` is transformed into a *core module*, and a UML package with stereotype `<<delta>>` is transformed into a delta module.

2. Package `interfacegenerator`
   The transformation of UML interfaces in the UML-VM diagram is managed in package `interfacegenerator`. Interfaces in the UML-VM diagram are transformed to JAVA interface.

3. Package `classgenerator`
   Package `classgenerator` manages the transformation of UML classes in the core module. As illustrated in Figure 6.3, a class in the core module is transformed into *abstract component class*, *abstract decorator class*, and *concrete component class* in the model layer. The resource layer is also generated to implement CRUD methods.

4. Package `deltaclassgenerator`
   The transformation rules for classes in delta modules are defined in package `deltaclassgenerator`. If the UML class has stereotype `<<modifiedClass>>`, the module calls functions to generate a modified class in the model and resource layers. As defined in Rule R7 in Table 6.1, a UML class with stereotype is transformed into a concrete decorator class. The concrete decorator class extends the abstract decorator class in the core module.

6.3 FeatureIDE WinVMJ Composer

FeatureIDE is an integrated framework to support feature-oriented development of software product lines (Meinicke et al., 2017). FeatureIDE is designed based on Eclipse IDE as an eclipse plugin. The following SPLE processes can be conducted in FeatureIDE:

- Domain analysis: A feature diagram is modeled using Feature Modeling Tool (FMT) which has graphical and textual notation. In Prices-IDE, we use the latest format of feature model in FeatureIDE, namely Universal Variability Language (UVL) (Sundermann et al., 2021).
- Requirements analysis: We can perform a feature selection to define the product line configuration. FeatureIDE provides support to validate the configuration based on the feature model.
- Domain Implementation: FeatureIDE has several implementation approaches based on FOP, DOP, and AOP. We develop a new composer in FeatureIDE to support domain implementation using the WinVMJ framework.
- Product Generation: a product can be generated based on feature selection. FeatureIDE composer manages the product generation, as defined in the domain implementation.

Prices-IDE utilizes feature modeling and configuration tools in FeatureIDE by extending the domain implementation support. FeatureIDE has several composers that manage the product generation process, such as AHEAD, Munge, AspectJ, FeatureHouse, and FeatureC++. When we create a new FeatureIDE project, we can choose the composer based on the domain implementation preference. Febrian (2022) developed a new composer in FeatureIDE called WinVMJ composer. The WinVMJ composer supports web-based product line development based on DOP.

The development of the WinVMJ composer in FeatureIDE provides the possibility to enhance the WinVMJ framework with feature selection and product generation. The WinVMJ framework, which was developed by Prayoga (2020); Samuel (2022); Waluyo (2022), did not integrate directly with a feature model. The relation between features and implementation in WinVMJ is defined in the JAVA properties file, and the selected feature is defined in the module declaration. These definitions could not refer to constraints in the feature model, so product validation is a challenging process in WinVMJ. The product generation is managed by a build script that composes required JAVA modules based on module declaration. In the WinVMJ composer, the product generation is performed based on feature selection in FeatureIDE. This process produces a valid product module automatically and the product can be compiled and run using the WinVMJ composer.

6.4 IFML to UI Generator

As mentioned in Section 6.1, interaction flow modeling language (IFML) is used in the domain implementation to model the abstract UI. IFML to UI Generator is a plugin in FeatureIDE that generates the front end of web applications. Initially, the UI generator plugin was developed by Rohma (2022) and then enhanced by Wilmarani (2023); Santoso (2023). The UI generator flow is shown in Figure 6.5. The IFML to UI generator is implemented using the Eclipse Acceleo model to text transformation. Acceleo modules (`.mtl files`) consist of transformation rules and templates. In the IFML UI generator, Wilmarani (2023) develops ReactJS template with various styles and Santoso (2023) provides a library for static page management.

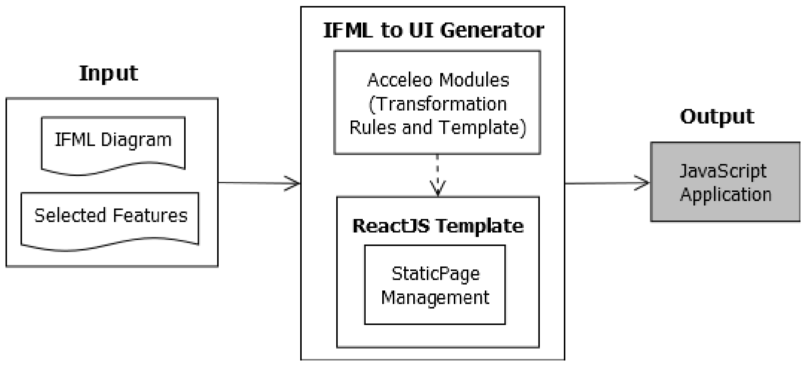


Figure 6.5: IFML to UI Generator Flow

The input of the IFML UI generator is an IFML diagram and a list of selected features. By executing the IFML UI Generator, a JavaScript (ReactJS) application for a specific product is generated. We have evaluated the IFML UI generator developed by Rohma (2022); Wilmarani (2023); Santoso (2023). We found duplication in the IFML diagram because the IFML diagram is not designed to model software product lines. For example, features *Income* and *Expense* have a similar business process. *Income* is modeled in a `ViewContainer Income`, as shown in Figure 4.3. *Expense* is modeled in a different `ViewContainer Expense` with the same IFML elements (as defined in `Income`).The IFML model could not be reused even though these features require the same UI flow and elements.

One of the contributions of this research is designing an extension of IFML to support variation modeling based on DOP, as explained in Section 4.4. The IFML UI generator, developed by Wilmarani (2023) and Santoso (2023), is not design for IFML-DOP model. Alisha (2023) extends the capability of the IFML UI generator to support IFML-DOP notations.The Acceleo modules are adjusted to recognize new elements in the IFML-DOP, such as *DeltaViewContainer*, *ModifiedList*, and *AddedVisualizationAttribute*. The transformation rules for those new elements in IFML-DOP are modified in Acceleo modules and templates. However, the ReactJS (JavaScript) template can be reused without modifications because the resulting (generated) front end remains the same.

6.5 Running Example: Charity Organization System

The proposed MDSPLE approach has been realized in Prices-IDE tools and plugins, as explained in Sections 6.1-6.4. In this part, we illustrate the practical application of Prices-IDE using a running example, an adaptive information system for charity organizations (AMANAH). Charity organizations have a typical business process: gather donations, develop programs, and distribute aid to society.

6.5.1 Domain Analysis

In the domain analysis, we use a feature diagram and UML diagrams to model the variability. Before modeling process, we analyze the requirements from charity organizations. We use an extractive approach in SPLE by exploring several charity organization websites to understand the requirements. We access features in the website to analyze the commonalities and variabilities of charity organization websites. The result is documented in the application-requirements (AR) matrix, that maps requirements to a list of applications. AR matrix is a simple way to perform commonality and variability analysis (Pohl et al., 2005) The domain analysis for charity organizations websites is conducted in three steps:

1. **Analysis with the AR Matrix**. The analysis process for charity organization websites with the AR Matrix is shown in Figure 6.6. The first stage is preparing the template of AR matrix, the column is for the name of charity organization and the row is for the features. Next, we select several charity organization websites with different scopes, sizes, and activities to analyze the commonality and variability. The chosen organizations are summarized in Table 6.2.

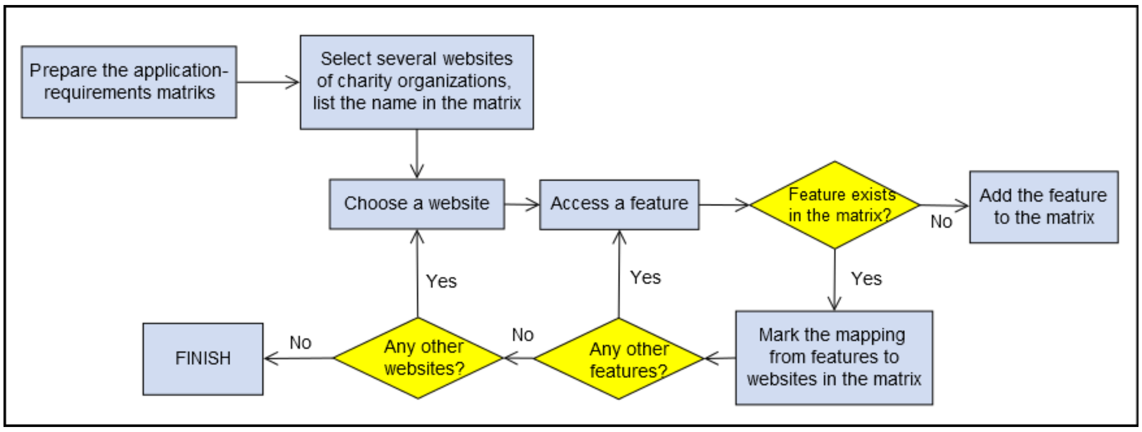

Prepare the application-requirements matriks
Select several websites of charity organizations, list the name in the matrix
Choose a website
Access a feature
Feature exists in the matrix?
No
Add the feature to the matrix
Yes
Yes
Yes
Mark the mapping from features to websites in the matrix
Any other features?
No
Any other websites?
No
FINISH

Figure 6.6: Analysis Process with AR Matrix

Table 6.2: Charity Organization Websites

Num	Organization	Description
1.	Human Initiative (HI)	HI is a charity organization that focuses on humanitarian programs. The programs are categorized into some aspects: education, health, economy, and disasters relief.
2.	Mizan Amanah (MA)	MA is an organization that supports orphans with charity programs, such as shopping with orphans and Eid donations for orphans. MA also gathers donation for *zakat* and *qurban*.
3.	Baitul Qu'ran (BQ)	BQ provides a free school for underprivileged children. BQ open donations for foster parents, *zakat, infaq, shadaqah, fidyah* and *qurban*.
4.	Aksi Cepat Tanggap (ACT)	ACT is a charity organization that works on humanitarian relief, post-disaster recovery, community empowerment, and also spiritual programs such as *qurban* and *zakat*.
5.	Titian Foundation (TF)	TF is a small organization that helps society to get education and training. TF provides funding for high school students, training for teachers, and microfinance for small entrepreneurs.
6.	LAZ Al-Azhar (LA)	LA is a philanthropic organization of Al-Azhar mosque in Jakarta. LA gathers donation to help disaster relief, infrastructure projects, and community empowerment. LA also receives donation for *zakat, infaq*, and *shadaqoh*.

We put the charity organization's name in the header column of the matrix. In the third stage, we access the feature in the website to understand the requirements. Each feature is placed on a single row in the AR matrix. If the feature does not exist, we add the feature to the AR matrix. Table 6.3 shows the AR matrix of charity organization websites. If the feature already exists in the AR matrix, we mark the mapping between features and websites. We try all the features from each website and then repeat the steps for all websites.

Table 6.3: Application-Requirements (AR) Matrix - Charity Organization Websites

Features	**HI**	**MA**	**BQ**	**ACT**	**TF**	**LA**
About Us	✓	✓	✓	✓	✓	✓
Bank Account	✓	✓		✓		✓
Chat-bot		✓		✓		
Contact Us	✓	✓	✓	✓	✓	
Online Donation	✓	✓	✓	✓		✓
Donation Confirmation	✓		✓	✓		
Donation Guide						✓
E-Magazine	✓					✓
FAQ	✓			✓		✓
Gallery	✓		✓			
Home	✓	✓	✓	✓	✓	✓
Language	✓			✓	✓	
Link Social Media	✓	✓	✓	✓	✓	✓
Login/Register	✓	✓				✓
News	✓	✓	✓	✓	✓	✓
Partner	✓	✓		✓	✓	✓
Program	✓	✓	✓	✓	✓	✓
Public Report	✓	✓		✓		✓
Search	✓	✓	✓	✓		
Services			✓			✓
Term and Condition	✓	✓		✓		✓
Testimonial					✓	
Video			✓			
Volunteer	✓			✓		
Zakat Calculator		✓				✓

2. **Feature Modeling**. The second step in domain analysis is feature modeling. In our research, the feature model is represented in a feature diagram. Before modeling the feature diagram, we perform the following tasks:

   - Commonality Analysis: requirements that are identical for all websites are good candidates for mandatory features in the feature diagram. Common features also can be chosen from basic requirements that every website for the domain must fulfill. Furthermore, organizational standards or national law could also enforce the chosen mandatory features.

- Variability Analysis: requirements that differ from each other are considered as variants of a feature. The difference can be viewed in terms of functionality, behavior, or user interface.

Based on the AR matrix, features *About us, Home, News, Program,* and *Link social media* exist in all websites. These features can be considered as mandatory features. The other features only exist in several organizations. For example, feature *Public Report* is only available for HI, MA, ACT, LA websites. This feature is thus optional for charity organization websites.

A feature diagram is a tree whose nodes represent features with unique names. The parent feature denotes a more general functionality. Variations can be defined as child features, as the child features represent a specialization. If a feature X1 is a child of feature X, X1 can be selected if feature X is also selected. Based on the parent-child perspective, we group features of charity organization websites by considering their functionalities. We also analyze commonality and variability for each group to design constraints in a feature diagram. A snippet of the feature diagram of AMANAH (charity organization product line) is shown in Figure 6.7. There are three mandatory features, *Activity*, *Financial Report* and *OrganizationInfo*; and an optional feature *Donation*.

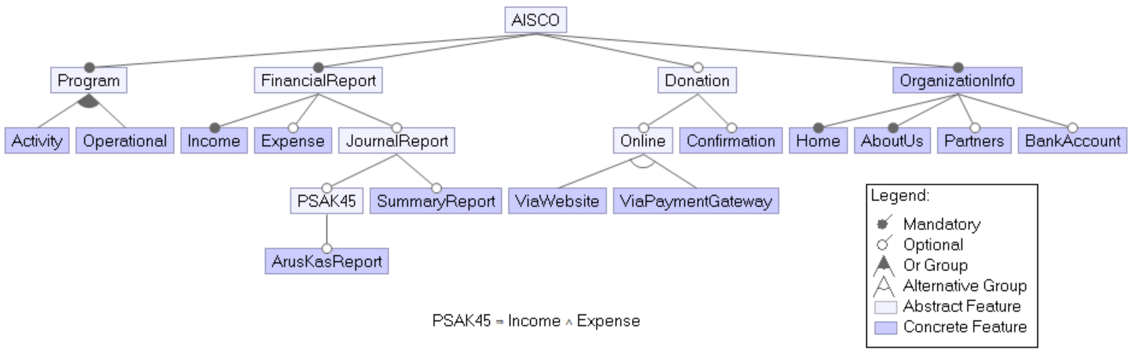


Figure 6.7: AMANAH Feature Diagram

3. **UML Modeling**. The third step in domain analysis is UML modeling. We use a UML-VM diagram to model the structural and architectural design in domain design. We have designed and implemented some features in AMANAH, such as *Program*, *FinancialReport*, *Donation*, and *OrganizationInfo*. Figure 6.8 shows a snippet of UML-VM diagram representing VM for the AMANAH product line.

Inside the package, stereotyped `<<vm>>` are the *core module* for *FinancialReport*, delta modules, and features they implement. The *core module* is modeled as a UML package stereotyped `<<module>>`. It consists of the interface `FinancialReport` and class `FinancialReportImpl`.

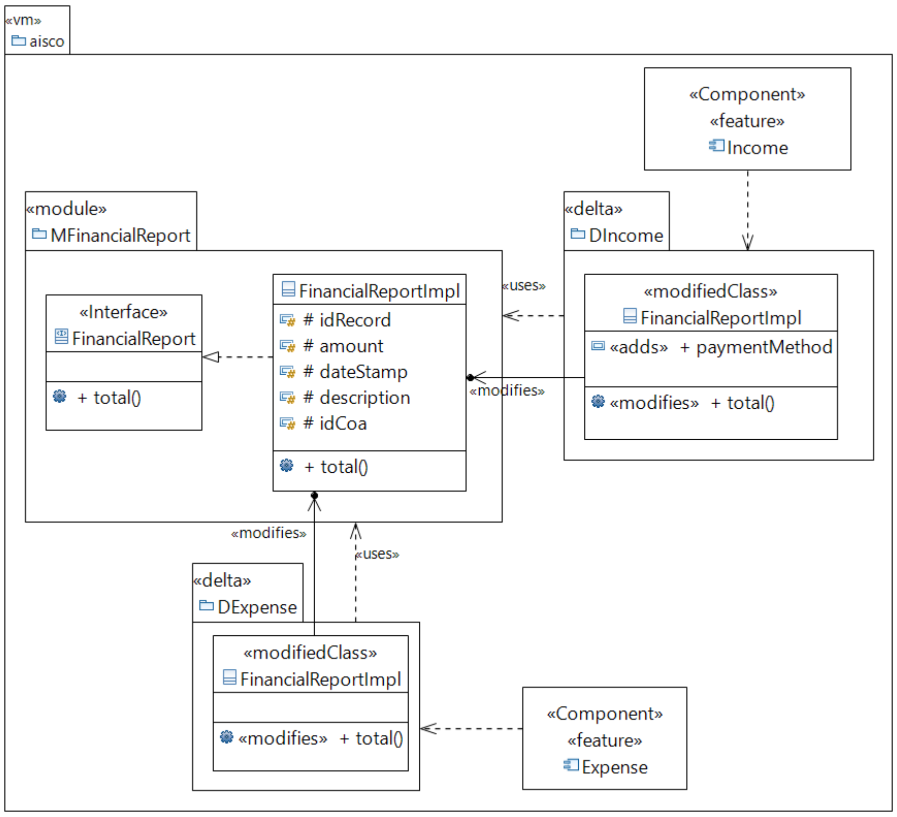


Figure 6.8: UML-VM diagram of `FinancialReport` module

The UML package `DIncome` with stereotype `<<delta>>` represents a delta module. This delta module modifies the core module `FinancialReport`, denoted by UML dependency with sterotype `<<uses>>`. A delta module may consists of several modification. Delta `DIncome` modifies class `FinancialReportImpl` in the core module by adding field `paymentMethod` and modifying method `total()`. The application of the delta operation is represented as a class `FinancialReportImpl` with stereotype `<<modifiedClass>>`. It contains the new field and modified method.

Another variant of feature *FinancialReport* is *Expense*. It is realized by delta `DExpense`, represented as the UML package with stereotype `<<delta>>`. Delta `DExpense` modifies class `FinancialReportImpl` by modifying method `total()`. This delta does not add new attributes or methods to the existing class. Deltas `DExpense` and `DIncome` modify the same class in the core module. Therefore, the application of these two deltas simultaneously must be managed in the domain implementation.

A delta module realizes feature's implementation. A feature is represented as a UML component with stereotype `<<feature>>`. The configuration knowledge that describes a delta application condition is modeled as a UML dependency stereotyped `<<when>>` or `<<after>>`. In Figure 6.8, there is a UML dependency `<<when>>` from delta `DIncome` to feature *Income* and from delta `DExpense` to feature *Expense*. It means that delta `DIncome` is applied when feature *Income* is selected and delta `DExpense` is applied when feature *Expense* is selected.

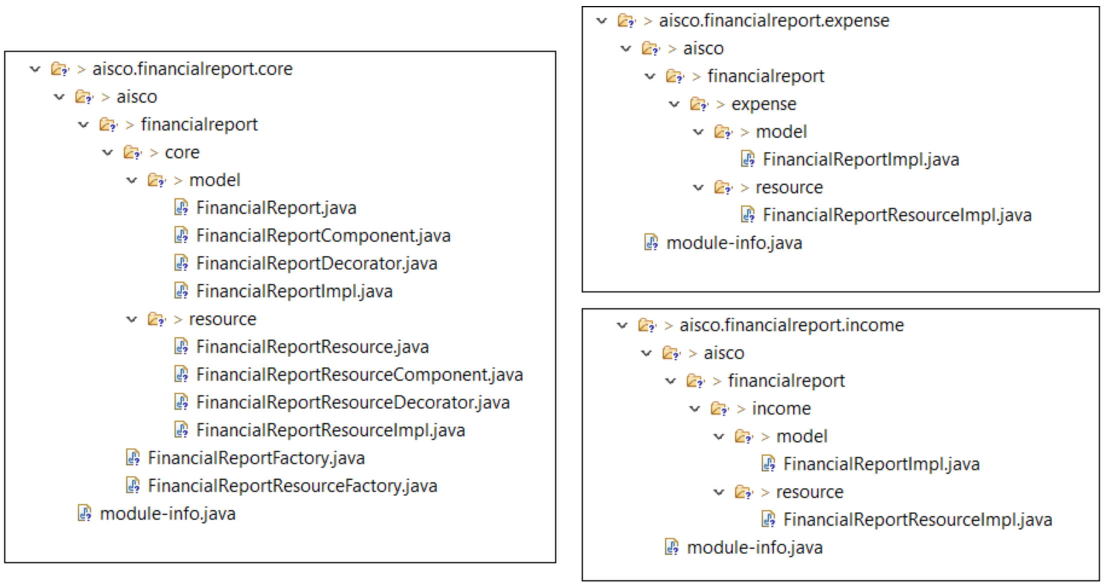
6.5.2 Domain Implementation

We use DOP approach with VMJ in the domain implementation. Since the case study is a web application, the WinVMJ framework is used in the web back-end development and the IFML diagram is used to model the abstract UI of the web front end. The skeleton of the WinVMJ source is generated from the UML-VM diagram and IFML diagram is used to generate JavaScript application during the product generation. We divide the domain implementation into four steps:

1. **Generate the Back End Implementation**. The output from domain analysis, *i.e.,* UML-VM diagrams, are transformed into WinVMJ source code. As defined in Section 6.2, the UML to WinVMJ tool is used to automate the transformation process. The structure of generated source code follows the decorator and factory pattern in VMJ modules. An example of generated WinVMJ modules for the UML-VM diagram (Figure 6.8) is shown in Figure 6.9. The format name for a core module is `[productlinename].[modulename].core` and for a delta module is `[productlinename].[coremodulename].[deltamodulename]`. For each module, there is a directory for model and resource layers.

2. **Code Implementation**. The UML to WinVMJ tool produces complete modules, packages, and classes of the WinVMJ project. However, the generated methods in the JAVA class do not have complete implementation. The UML-VM diagram only models the structural behavior, so the methods are transformed into a code skeleton. The standard methods for database operations, such as create, read, update, and delete, are completely generated by UML to WinVMJ tool.

   As defined in the transformation rules in Table 6.1, a UML class in the core module is transformed into three classes in the WinVMJ model layer and three classes in WinVMJ resource layer. Four of these six classes are abstract classes, and the other two classes are concrete classes. The four abstract classes are fully generated by UML to WinVMJ tool because all methods in abstract classes can be defined as



```
v > aisco.financialreport.core
  v > aisco
    v > financialreport
      v > core
        v > model
            FinancialReport.java
            FinancialReportComponent.java
            FinancialReportDecorator.java
            FinancialReportImpl.java
        v > resource
            FinancialReportResource.java
            FinancialReportResourceComponent.java
            FinancialReportResourceDecorator.java
            FinancialReportResourceImpl.java
        FinancialReportFactory.java
        FinancialReportResourceFactory.java
    module-info.java
```

```
v > aisco.financialreport.expense
  v > aisco
    v > financialreport
      v > expense
        v > model
            FinancialReportImpl.java
        v > resource
            FinancialReportResourceImpl.java
  module-info.java
```

```
v > aisco.financialreport.income
  v > aisco
    v > financialreport
      v > income
        v > model
            FinancialReportImpl.java
        v > resource
            FinancialReportResourceImpl.java
  module-info.java
```

Figure 6.9: Generated WinVMJ modules for FinancialReport variants

abstract methods. The developer only needs to complete a method's implementation for two concrete classes in each module.

3. **Integration with Feature Model**. Integration of the generated WinVMJ source code with feature model is conducted in the FeatureIDE project using WinVMJ composer, developed by Febrian (2022). We create a FeatureIDE project and choose `WinVMJ` in the composer selection. As a result, the FeatureIDE project has a structure that follows the WinVMJ framework. The generated source code is placed in directory `modules`, directory `external` is used to store external library and directory `src` is used to store the required modules for a product based on the feature selection. The feature selection it self is defined in directory `configs`.

   We use a feature model in UVL notation, that has visual and textual representations. The feature model is defined in file `model.uvl` and the source code of all JAVA modules is stored in directory `modules`. A feature is implemented by one or more delta modules (deltas) in DOP. The mapping between features in the feature module and deltas is defined in file `feature_to_module.json`, as shown in Listing 6.1. For example, feature *Income* is implemented by delta module `aisco.financialreport.income`.

```
{
  "Income": ["aisco.financialreport.income"],
  "Expense": ["aisco.financialreport.expense"],
}
```

Listing 6.1: Mapping Feature to Module

4. **UI Modeling**. An activity in the domain implementation for front-end development is modeling abstract UIs using IFML diagrams. An IFML diagram can be used to model the content, user interaction, and control behavior of the applications. IFML elements supports platform independent model because they only model the abstract UI without detail information about style, size, layout, or color. As explained in Section 4.4, we extend IFML elements with DOP notation to model commonality and variability in the abstract UI.

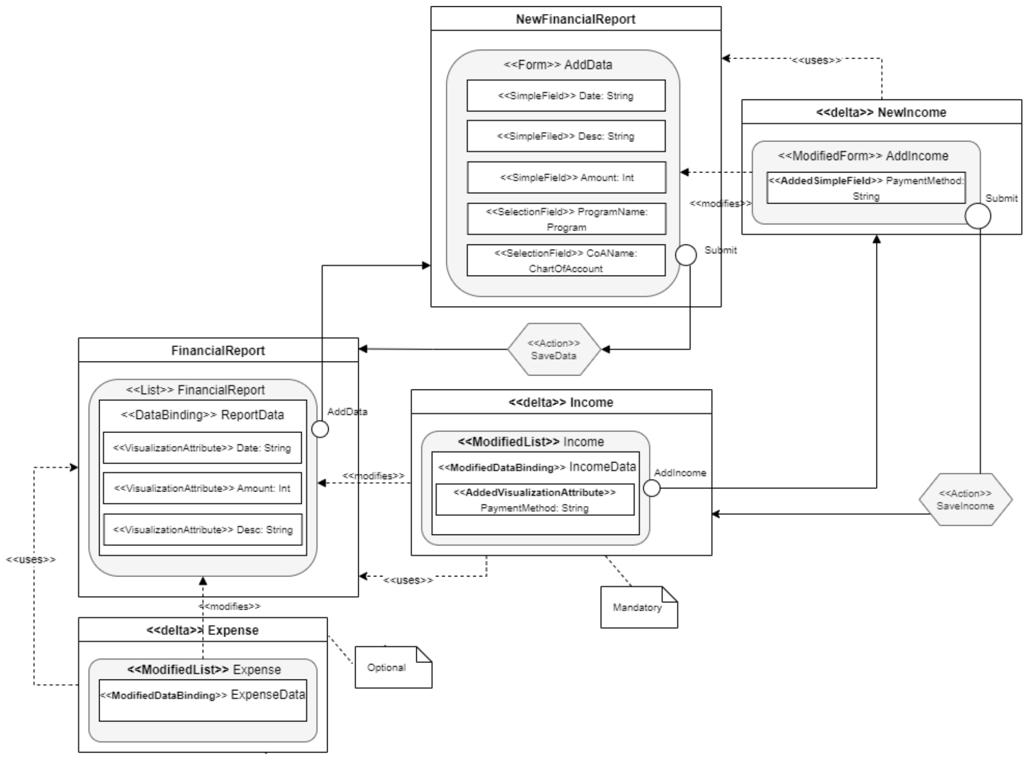

NewFinancialReport
<<Form>> AddData
<<SimpleField>> Date: String
<<SimpleFiled>> Desc: String
<<SimpleField>> Amount: Int
<<SelectionField>> ProgramName: Program
<<SelectionField>> CoAName: ChartOfAccount
Submit
<<uses>>
<<delta>> NewIncome
<<ModifiedForm>> AddIncome
<<AddedSimpleField>> PaymentMethod: String
Submit
<<modifies>>
<<Action>> SaveData
FinancialReport
<<List>> FinancialReport
<<DataBinding>> ReportData
<<VisualizationAttribute>> Date: String
<<VisualizationAttribute>> Amount: Int
<<VisualizationAttribute>> Desc: String
AddData
<<delta>> Income
<<ModifiedList>> Income
<<ModifiedDataBinding>> IncomeData
<<AddedVisualizationAttribute>> PaymentMethod: String
AddIncome
<<Action>> SaveIncome
<<modifies>>
<<uses>>
Mandatory
<<uses>>
<<modifies>>
<<delta>> Expense
<<ModifiedList>> Expense
<<ModifiedDataBinding>> ExpenseData
Optional

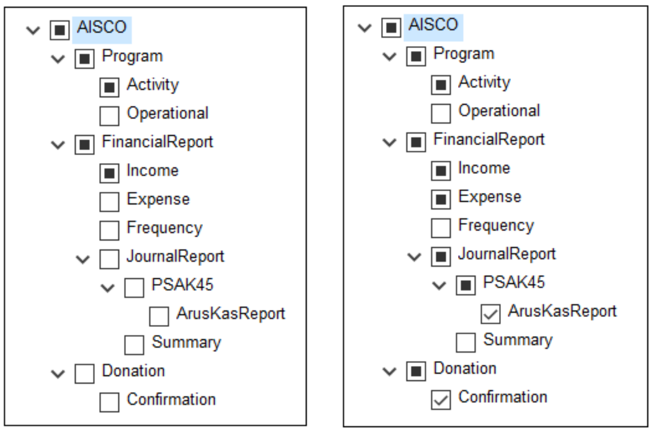
Figure 6.10: IFML-DOP diagram - FinancialReport

A feature in the feature diagram may have one or more web pages. A web page is modeled as a ViewContainer in the IFML diagram. An example of IFML-DOP diagram for feature *FinancialReport* in charity organization system is shown in Figure 6.10. A ViewContainer FinancialReport consists of List FinancialList to display the data. In DataBinding, IFML elements may have references to the domain model in the back end. For example, to display data from the database, DataBinding ReportData refers to the back-end API. We set an API address in a property of DataBinding called Uniform Resource Identifier. We can choose specific attributes to be displayed by defining VizualitionAttribute.

In the IFML-DOP diagram, existing ViewContainer can be modified by ViewContainerDelta. In Figure 6.10, ViewContainerDelta Income modifies ViewContainer FinancialReport by modifying List with a new DataBinding Income. A new VisualizationAttribute is also added to display a field that does not available in existing elements. ViewContainerDelta Income also has a different NavigationFlow from ViewContainer FinancialReport. The NavigationFlow connects to ViewContainerDelta NewIncome. It is triggered by Event AddIncome. ViewContainer FinancialReport is also modified by ViewContainerDelta Expense. This delta does not add a new IFML element but it modifies the DataBinding.

6.5.3 Product Generation

The *application engineering* in Figure 4.2 starts with feature selection. In Prices-IDE, the feature selection process is conducted in FeatureIDE by defining a configuration. FeatureIDE provides a configuration editor to select features, as shown in Figure 6.11. The left box is an initial configuration before any feature is selected. The mandatory features, such as *Activity* and *Income*, are automatically selected in the initial configuration. The right box in Figure 6.11 is a configuration for product `BisaKita`. Product `BisaKita` consists of features *Active, Income, Expense, ArusKasReport* and *Confirmation*.



- ■ AISCO
  - ■ Program
    - ■ Activity
    - ☐ Operational
  - ■ FinancialReport
    - ■ Income
    - ☐ Expense
    - ☐ Frequency
    - ☐ JournalReport
      - ☐ PSAK45
        - ☐ ArusKasReport
      - ☐ Summary
  - ☐ Donation
    - ☐ Confirmation

- ■ AISCO
  - ■ Program
    - ■ Activity
    - ☐ Operational
  - ■ FinancialReport
    - ■ Income
    - ■ Expense
    - ☐ Frequency
    - ■ JournalReport
      - ■ PSAK45
        - ✓ ArusKasReport
      - ☐ Summary
  - ■ Donation
    - ✓ Confirmation

Figure 6.11: Feature selection - BisaKita product

The product generation consists of two main processes:

1. **Generate Web Back End**
   A product line may have several product variants and a configuration file is defined for each product. All these configuration files are stored in directory `configs` in FeatureIDE project. The product generation for a web back-end in the FeatureIDE WinVMJ composer is conducted in the following steps:

   (a) Set configuration. We must choose a config file as a current configuration to generate a product based on selected features. Right-click on a selected config and choose FeatureIDE → `Set As Current Configuration`. When the active configuration file is modified and saved, the required JAVA modules from directory `module` are copied to directory `src`. The product module is also generated at this stage. It consists of a module declaration (`module-info.java`), a package, and a JAVA product class.

(b) Compile product. Based on the chosen configuration, required modules are generated in directory `src`. The next step is compiling the generated modules into a running application. Select directory `src`, right-click, and choose FeatureIDE → WinVMJ → Compile. The results are Jar files based on selected features and required WinVMJ libraries.

(c) Run back end. A script to run the generated product (back-end), namely `run.bat`, is generated from the previous step (*compile product*). The script contains *commands* to create tables, insert required data, and run the Java source code. It can be executed from *terminal* or Eclipse (*external tools configuration*).

2. **Generate Web Front End**
The front end of web applications is generated from the IFML diagrams. We have to ensure that the selected features in the front end are the same as the selected features in the back end. We use a configuration file in the FeatureIDE project as input for the IFML UI generator. The IFML-UI generator consists of transformation rules and ReactJS (JavaScript) templates. The following steps are performed to generate a web front end:

(a) Prepare the UI boilerplate. The UI boilerplate is available in the UI generator to provide common assets of generated front end. First, create a general project in Eclipse and then copy the templates to the project. Second, right-click on the Eclipse (empty) project and choose `IFML UI Generator` → `Copy Template Here`.

(b) Generate or define selected features. We reuse the configuration in FeatureIDE as an input in the UI generator to ensure consistency in feature selection. First, choose `Generate Selected Features` in FeatureIDE WinVMJ composer. Then, right click on the `configs` file, choose FeatureIDE → WinVMJ → Generate Selected Features. Choose the ReactJS project, defined in step (a), as a target project. A list of selected features is generated in plain text format.

(c) Generate the front end. The last step is generating the front end (JavaScript) from the IFML diagrams. Right-click on the IFML diagram (file `.core`), choose `IFML UI Generator` → `Generate UI`. Choose the ReactJS project, defined in step (a), as a target project.

(d) Run the product. The web front (ReactJS) is fully generated from the UI generator. Install the required dependencies by running `npm install` command and run the ReactJS application using `npm start` command. A new terminal is also required to run the static server (for static page management) using `npm run json:server` command.

After finishing step 1 (generate web back-end) and step 2 (generate web front-end), the product is ready to use. An example of the generated product (BisaKita website) is shown in Figure 6.12. The generated features are *(Program) Activity, Income, Expense, ArusKasReport, (Donation) Confirmation*, as defined in the feature selection for `BisaKita` (Figure 6.11). A script for *automatic deployment* is also available to deploy the generated website on the cloud (server).

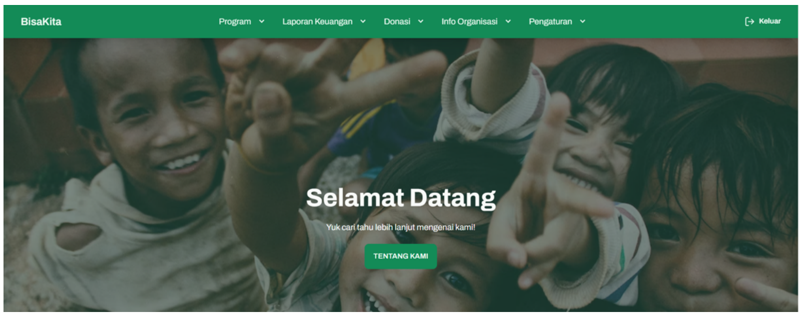


Figure 6.12: Product - BisaKita

The product generation steps defined, such as feature selection, generate web back end and generate web front end can be reused to generate other products. For instance, we want to create a website for `CharitySchool` organization. `CharitySchool` requires two features, *(Program (Activity) and Income)*. The process is started by defining a new configuration in FeatureIDE for `CharitySchool`. Select all required features and choose '`Set As Current Configuration`' to start the product generation for the web back-end, as defined in step 1(a). Continue to steps 1(b) and 1(c) to finish the generation process. To generate the web front end for `CharitySchool`, steps 2(a), (b), (c) and (d) in this section can be repeated. Since the selected features for `CharitySchool` product is not the same as `BisaKita`, the generated website is also different.

CHAPTER 7

EVALUATION

The first section of this chapter analyzes the usage of the UML-VM profile in variability modeling. Then, the UML-VM profile is evaluated using some quality criteria in SPLE. The second section evaluates the applicability of the UML-VM profile in the MDSPLE process. In the third section, the degree of automation is measured to evaluate the improvement quantitatively. We also evaluate the process improvement in the MDSPLE by comparing the standard (clown-and-own) and this MDSPLE approach in the fouth sections. In the last sections, we discuss the threats to validity of this research.

7.1 Analysis of Variability Modeling

Variability management is a crucial concern in software product line engineering (SPLE), an emerging approach to develop software with variations. The domain analysis process is started with modeling commonality and variability. Common features are implemented in reusable components, and variations are designed to support mass customization. In domain implementation, many product variants can be managed in a single development. Based on the empirical study from Echeverría et al. (2021), SPLE is more efficient than *clone-and-own* approach, but the developers spend more time checking actions.

In SPLE, the variability can be modeled using several approaches, such as feature model, decision model, and orthogonal variability model (OVM). The first research question analyzes which application artifacts can be modeled as software product lines. We model the commonality and variability for *functional requirements* without considering the variability of non-functional requirements. We use a feature model combined with UML to model the problem domain and interaction flow modeling language (IFML) to model the abstract user interface.

Based on a systematic review of evaluation of variability management by Chen & Ali Babar (2011), most approaches use *example application* to evaluate their research. Chen & Ali Babar (2011) found that *rigorous analysis* is usually applied when formal methods are used in variability management approaches. The UML-VM profile is defined based on the VM concept. We rely on the VM formal semantic, which is explained by Damiani et al.

(2023). In this section, we use quality criteria for SPL implmentation techniques defined in Apel et al. (2013) to evaluate the approach: *preplanning effort*, *feature traceability*, *separation of concerns*, *information hiding*, *granularity*, and *uniformity*. The definition of these criteria are explained in Section 3.1. We also discuss and compare our approach to the other MDSPLE approaches for web-based product line development that are explained in Section 2.4. The results are summarized in Table 7.1.

Table 7.1: Comparison of MDSPLE approach for web-based SPLs

Quality Criteria	**Laguna et al. (2009)**	**Alferez & Pelechano (2011a)**	**Vranic & Taborsky (2016)**	**Horcas et al. (2018)**	**Horcas et al. (2022)**	**This Approach**
Preplanning effort	new artifacts and models	new artifacts and models	new artifacts and models	new artifacts and models	new artifacts and models	new models and generate artifacts
Feature traceability	package merge mechanism	weaving model	feature and transforma-tion	variation point	binding and references	UML profile and configuration knowledge
Separation of concerns	intended by impl. paradigm	impl. dependent	intended by impl. paradigm	impl. dependent	impl. dependent	intended by impl. paradigm
Information hiding	impl. paradigm dependent	impl. paradigm dependent	impl. paradigm dependent	comp. mechanism dependent	support at architectural level	support at architectural and imple-mentation levels
Granularity	coarse-grained	coarse- and medium-grained	coarse-grained	all levels	all levels	coarse- and medium-grained
Uniformity	enforce common style	different styles	different styles	common style	common style	common style

- ***Preplanning Effort***. A product variant in an SPL may require a new feature that does not exist in the existing artifacts. Preplanning should be conducted to address additional requirements of the new variant. In standard development, a new UML diagram is created to model a new product variant, and the new features are modeled in the new UML diagram.

  The UML-VM profile is designed based on VM that supports DOP. A new requirement or feature in DOP is implemented by defining a delta module. In the UML-VM diagram, we use the delta-oriented programming paradigm. We do not need to create a new diagram for each product variant, but we can update existing models by adding a new delta module. A delta is modeled as a UML package that can add, modify, or remove existing elements without changing the original model.

The MDSPLE framework in this research is supported by model transformation tools. The implementation artifacts for a new feature can be generated from the models. It does not require changes in the code base. Compared to other approaches, the amount of *preplanning* in this research is relatively low. Other approaches require creating new models, changing the code base, or adding new artifacts because they use different approaches in problem and solution domains.

- ***Feature Traceability***. In SPLE, a feature in the feature model can be implemented in one or more implementation artifacts. *Feature traceability* is the ability to trace the relationship between features in the variability model and their implementation. Most approaches in Table 7.1 provide a mechanism to maintain traceability, such as a package merge mechanism in standard Java or mapping features and artifacts in weaving models, model transformation, or variation points. The mapping mechanism in these approaches is complicated because the variability is modeled in the problem domain, but the solution domain uses standard programming languages that do not expose variability at the implementation level. Herefore, the traceability in the existing approach could be more challenging to maintain.

  This approach uses a dedicated UML profile, namely the UML-VM profile, that supports good traceability between models and implementation artifacts because there is a one-to-one mapping between UML-VM stereotypes and VM elements. The mapping is also implemented as transformation rules in the UML to WinVMJ tool. The relationship between features and delta modules is also defined in the *configuration knowledge*. Therefore, the traceability is well preserved in the problem and solution domains with support from the UML-VM profile and *configuration knowledge*.

- ***Separation of Concerns***. Separation of concerns refers to the strategy of separating features into several models and implementations based on functionalities. In existing MDPSPLE approaches for web-based development, the separation of concerns is managed at the implementation level. Some approaches depend on the programming paradigm to manage the concerns. In DOP, each delta is designed to implement specific functionalities. Several delta modules can be composed to realize the implementation of specific features. Thus, in this approach, the separation of concerns is intended by the implementation paradigm.

  The separation of concerns in this research is also supported at the architectural level. UML has many kinds of diagrams to model a system from different views and we use component, class, and package diagrams in the UML-VM profile. Each diagram has different levels of abstraction to distinguish the concerns. Component diagrams can be used to model features and product variation. The dependency between components is modeled in the provided or required interfaces. The UML class diagram is used in domain design to model variability in more detail. Classes in the UML-VM diagram are contained in a package with stereotype `<<module>>` or `<delta>>`. The UML package diagram can be used to model the higher level of abstraction, and each package denotes different concerns in the UML.

- ***Information Hiding***. Information hiding emphasizes the separation of the internal and external parts of the system by decomposing the artifacts into modules or components. The internal part is hidden from other modules, and the external part describes a contract with other modules. As shown in Table 7.1, information hiding in web-based product line development may depend on the implementation paradigm or composition mechanism. An approach by Horcas et al. (2022) supports better information hiding (implementation independent) than others because of the modeling mechanism. However, this approach does not support information hiding at the implementation level.

  In this research, information hiding is not only supported at the architectural level but also the implementation level. VM is designed by module concept, so information hiding is fully supported by specifying which elements should be hidden and which should be exposed. Modules do not export anything by default, so their elements are not accessible from other modules. Elements in a module are visible to other modules when the module exports the name, and then another module can import the elements. Since the UML-VM profile is defined based on VM, the design of the UML diagram using the UML-VM profile follows the VM concept.

- ***Granularity***. Granularity specifies the level of variability that can be modeled or implemented. Annotation based approach usually supports fine-grained granularity and composition-based approach supports coarse-grained granularity. Approaches by Horcas et al. (2018, 2022) support all levels of granularity because they combine annotation and composition approaches. Other approaches only support medium- or coarse-grained variability.

  In our research, the UML-VM profile and DOP implementation support coarse- and medium-grained variability because delta modules can add, remove, or modify interfaces, classes, methods, or fields. For example, coarse-grained variability is managed when a delta module adds a new class. Adding new elements to the UML class can be considered medium-grained variabilities, such as adding new methods or fields. The UML diagram in the UML-VM diagram does not model the method's implementation. Therefore, the UML-VM profile does not support fine-grained variability, such as adding a new statement to a method body.

- ***Uniformity***. A general approach should represent all kinds of artifacts based on the principle of *uniformity*. Approaches in Table 7.1 use different artifacts representation. Laguna & Crespo (2013) enforces a common style using UML diagrams for modeling artifacts. However, Alferez & Pelechano (2011a) and Vranic & Taborsky (2016) use different styles for representing web-based artifacts. As web applications have various kinds of artifacts, a generic model is required to represent artifacts in a similar manner or style. Horcas et al. (2018, 2022) propose a mechanism to model various web artifacts in a single model.

  In this research, we use the UML-VM profile, which models variability in different levels of abstraction. The variability of features is modeled in the UML component diagram, the dependency is modeled in the UML package diagram, and the detail of variability is modeled in the UML class diagram. All these diagrams form the product line architecture capable of modeling commonality and variability. The modeling level also supports language-independent implementation. Based on the UML-VM diagram, any delta-oriented programming techniques can be used in domain implementation. The IFML diagram that is used to model the abstract UI also supports uniformity because it represents platform-independent UI artifacts.

7.2 Applicability of the UML-VM Profile

The second objective of this research is *defining a unified modeling mechanism for web-based product line applications*. This aim is achieved by defining the UML-VM profile because several product variants in software product lines can be modeled in a single UML diagram. One UML diagram with a VM profile (UML-VM diagram) can represent all variants. Thus, the design for a specific product variant can be derived from the UML-VM diagram. The second research question, *how to model the problem domain of a web-based product line*, is answered by the UML-VM diagram.

Based on the results of this research, the UML-VM profile is applied in the following processes:

1. ***Designing the Domain Implementation***. We design the domain implementation based on VM, namely Variability Modules for Java (VMJ). VM is defined based on

the module concept and DOP, which support variability, whereas VMJ is defined on top of JAVA, which supports object-oriented programming (OOP). The problem is implementing variability based on DOP in JAVA programming language. VM elements are already mapped to UML stereotypes in the UML-VM profile. Thus, we have a bridge between DOP and OOP in the UML-VM profile. The UML-VM profile is used as a reference in the domain implementation because UML and JAVA are designed to develop the object-oriented system. Once we have a UML diagram with VM notation, mapping from the UML-VM diagram to JAVA is more straightforward. The UML-VM profile contributes to answering the third research question, *how to implement the product line based on the model in the problem domain*.

2. ***Model Transformation Rules***. The input for UML to WinVMJ tool is a UML-VM diagram, and the output is JAVA source code, which follows the structure of the WinVMJ framework. As described in Section 6.2, the UML-VM profile is used as a reference in the transformation rules. The model transformation tool produces stubs of core and delta modules. The method's implementation is completed manually because the UML-VM diagram only provides structural aspects without dynamic behavior. The UML-VM profile is a main ingredient in the model transformation tool. One-to-one mapping between VM elements and UML stereotypes brings determinism in transformation rules. The fourth objective in this research, *design model transformation tools to bridge the problem and solution domains*, is also supported by the UML-VM profile.

7.3 Automated Code Generation

The MDSPLE framework is realized in the Prices-IDE tools, as explained in Section 6.1. Prices-IDE consists of several tools (plugins) to automate the development process, such as UML to WinVMJ tool (Section 6.2), WinVMJ Composer (Section 6.3), IFML to UI Generator (Section 6.4). In this section, we measure process improvement by analyzing the usage of automated code-generation tools. The automation tools may generate partial or complete source codes. Therefore, we compare the line of codes (LoC) between the generated source code and the running (complete) source code.

In this experiment, we use the AMANAH case study, as explained in Section 6.5, to analyze the automation process. We include the implementation of several features for the measurement, such as *Activity*, *Operational*, *Income*, *Expense*, *FinancialPosition*, *ActivityReport*, and *DonationConfirmation*. The measurement is conducted in the following steps:

1. Generate a WinVMJ (Java) code from the UML-VM diagram using UML to WinVMJ Tool. Copy the generated source to a new workspace and complete the source code (methods' implementation). Count the total line of codes in the generated source code and fully implement (complete) the source code.

2. Generate a specific product using WinVMJ composer by selecting the required features. For example, we generate product `CharitySchool` that requires features *Activity, Income, Expense,* and *DonationConfirmation*. A new JAVA module representing product `CharitySchool` is generated.

3. Generate a JavaScript (React) application from the IFML diagram for product `CharitySchool` using IFML UI Generator. We do not complete anything in the JavaScript application because the IFML UI Generator produces a complete source code.

Table 7.2 shows the comparison between generated and complete source code. As mentioned in the steps above, we count the total line of code produced by automated code generation. The UML to WinWMJ tool generates partial JAVA source code, around 60% lines of code are produced. The developer must complete the method's implementation because the UML-VM diagram only models the structural (static) behavior of the problem domain. The WinVMJ composer is able to generate a product module based on selected features. A running backend application can be derived from the reusable core and delta modules during product generation. Based on our experiment, the IFML to UI generator produces a complete JavaScript application from the IFML diagram. Therefore, 100% JavaScript source code is produced by our tool support.

Table 7.2: Comparison of Generated and Complete Source Code

Tool	Input	Output	LoC Generated	LoC Complete
UML to WinVMJ	UML-DOP Diagram	Java (WinVMJ)	3.089	5.093
IFML to UI Generator (*BisaKita* Product)	IFML Diagram	JavaScript JSON CSS HTML	6178 31.746 14 20	6178 31.746 14 20

7.4 Process Improvement

We design a scenario for requirements changes in the AMANAH case study to evaluate improvement in the development process. Assume that an organization, namely

`HeroFoundation`, requires a new feature, which does not exist in AMANAH product line. `HeroFoundation` needs feature *AnnualReports* that provides an automatic financial report for a specific year. First, we add *AnnualReports* as a new feature in the feature diagram. Figure 7.1 shows the modification of AMANAH feature diagram (a new feature is denoted by a red rectangle). Then, we implement the feature using two different approaches, *clone and own* and SPLE.

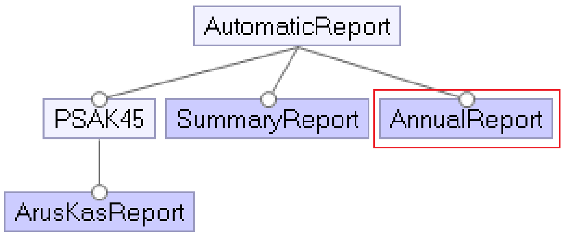


Figure 7.1: A new feature in AMANAH feature diagram

For *clone and own* approach, we use JAVA web framework (Sring Boot). We implement the AMANAH case study using Spring Boot and compare the implementation to WinVMJ to evaluate the process improvement. Then, we also compare how to implement a new feature in WinVMJ 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 (Apel et al., 2013).

The comparison of implementing a new feature in WinVMJ and Spring Boot is summarized in Table 7.3. 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 7.3, 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 SpringBoot, 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 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

Table 7.3: Comparison of Requirements Changes in WinVMJ and Spring Boot

Process	WinVMJ (MDSPLE)	Spring Boot (*Clone and Own*)
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*

feature in WinVMJ is lower than the effort in Spring Boot.

7.5 Threats to Validity

In this section, we discuss threats that could affect the validity of this work. We evaluate the MDSPLE approach based on two categories of validity threats defined by Wohlin et al. (2012). A case study is used to demonstrate the practical application of MDSPLE using Prices-IDE. One of the main concerns when using case studies is the validity of the results and their applicability to other contexts.

7.5.1 Internal Validity

Internal validity concerns the reliability of the results within the experiment environment. This research results in a running web application generated based on feature selection. The generated web is validated automatically using tools in the Prices-IDE. The WinVMJ composer handles the validation of the web back end. If the selected features violate the constraints, the configuration is failed to save, and the product is not generated. However, the current UI generator does not have a mechanism for validating the generated front end. To mitigate this threat, we take the input for generating the UI from product configuration in FeatureIDE.

Threats to internal validity are related to uncontrolled aspects that may affect the experiment results (Wohlin et al., 2012). We define the uncontrolled aspects based on web developers' and users' perspectives. For the developers, the uncontrolled aspect relates to the development environment. We develop Prices-IDE on top of Eclipse Modeling Tools. The operating system, Eclipse's version, Eclipse plugins, and Java's version are uncontrolled aspects of the developer's environmental setting. We prepare an Eclipse application with all required plugins in a single Eclipse to prevent problems during development.

From the web user's perspective, the uncontrolled aspect relates to the running application. The generated web is standard JAVA and JavaScript applications. However, we cannot ensure that the web application runs successfully in any web server or cloud provider. The uncontrolled aspects are the web application's environment, such as database connectivity, deployment, and server. To mitigate this threat, we have to define a deployment architecture or automated deployment script for software product lines. Therefore, the problem during deployment can be prevented.

7.5.2 External validity

External validity refers to the generalizability of experiment conditions (Wohlin et al., 2012). We must ensure that the experiment outcome from one environment can be applied to other environments. Threats to external validity concern the ability to generalize experiment results to other environments or outside the scope. We analyze the threats from two perspectives: the case studies and the organizational culture.

First, from the case study perspective, we applied the MDSPLE approach to two running examples. In Section 6.5, we show the applicability of the MDSPLE approach using a case study charity organization (AMANAH) web application. We also use the MDSPLE approach to develop another case study, the Bank Account product line. Bank Account is a generic case for SPLE, not a web application, used by Hähnle (2013) and Thüm et al. (2012). We also develop the Bank Account product line using the proposed MDSPLE approach. The approach can be applied to various domains, not only a web application. Generalization in other domains does not require additional experimental activities.

The MDSPLE approach is also used to develop other case studies, such as manufacturing software in SCM (Komarudin et al., 2021), e-Shop microservices applications (Setyautami et al., 2020), payment gateway (Koesnadi et al., 2022). Komarudin et al. (2021) and

Setyautami et al. (2020) use ABS language in the domain implementation, while this research and (Koesnadi et al., 2022) use DOP implementation with VMJ. Therefore, different DOP implementations can be used in the MDSPLE approach.

Second, external validity also relates to organizational culture. As the proposed MDSPLE is a novel approach to developing web applications, we cannot control the developers' preferences. Clone and own is still profitable in the industry, and sometimes, the setup cost of SPLE is higher upfront. Although the MDSPLE approach provides a framework and tools for developing various web applications in a single development, utilization of this framework in industrial practice could not be generalized.

CHAPTER 8
CONCLUSION AND FUTURE WORK

This research presents a model-driven software product line engineering (MDSPLE) framework based on delta-oriented programming (DOP). In this chapter, we revisit the research questions and conclude to what extent our research results have answered the questions. We also discuss and propose potential directions for future work.

8.1 Conclusion

Pohl et al. (2005) emphasize that the maturity of the software development process plays a crucial role in the successful adoption of SPLE. Therefore, it is essential to have a well-defined and structured software development process that is clear for developers. In this research, we have presented the MDSPLE framework based on DOP, as detailed in Chapter 4. This framework encompasses both the problem and solution domains and is supported by modeling extensions and automation tools.

In addressing the first research question regarding which artifacts of a web application can be modeled as a product line, we have analyzed that variability can exist not only in the web backend and frontend, but also in other aspects such as database systems, deployment servers, and operating systems. However, modeling all aspects of web applications as variability raises complexities in the feature selection process. Multi-stage configuration might be required if a product line models variability in multiple aspects. In our research, we decide to model variability related to user requirements by modeling the web backend and frontend as SPL.

To address the second research question concerning the modeling of the problem domain, we have introduced the UML-VM profile as an extension of UML diagrams. By standardizing the modeling notations, developers can effectively model variability in UML. The UML-VM profile maps variability model (VM) elements to UML stereotypes. A single UML diagram can capture the variability of multiple products in the SPL, and specific product variants can be derived based on selected features. The second research objective, *define a unified modeling mechanism for delta-oriented software product lines*, is achieved.

The third research question, focused on implementing the product line based on the model in the problem domain, is addressed through the variability modules for Java (VMJ). VMJ serves as an architectural pattern in JAVA for the domain implementation based on DOP. The FeatureIDE tool is extended to manages product configuration based on selected features. The third objective of this research is achieved because VMJ is used in the domain implementation to generate a running application.

To address the fourth research objective of designing a model transformation that bridges the problem and solution domains, we have developed a model transformation tool from UML-VM diagrams into WinVMJ source code. The UML-VM profile is used as a reference for transformation rules. The transformation rules are implemented within the model transformation tool. Our experiments have demonstrated the successful generation of source code (solution domain) from the UML diagrams (problem domain).

The MDSPLE framework is supported by various tools such as diagram editors, model transformation tools, and product line generators. To achieve the fifth research objective of integrating code generation and product derivation processes, we have designed Prices-IDE, an integrated development environment for web-based product lines. The theoretical concept of the MDSPLE framework can be applied to any domain, but with the current tool support, we have illustrated the practical application of Prices-IDE through a case study (web application).

8.2 Future Work

The following areas of future work can be explored based on the research conducted in this study:

1. Extending the UML-VM profile and transformation tools to include modeling of dynamic behavior in the UML-VM diagram would enhance the completeness of the generated source code. It should be possible to model the dynamic behavior using UML activity diagrams, UML sequence diagrams, or BPMN.
2. Integrating automated feature extraction and variability identification into the MD-SPLE approach would streamline the requirements engineering process. We have designed a UML-VM profile in this research, so the feature extraction result from various artifacts could be represented as a unified model.
3. Incorporating software analysis and verification techniques, such as type checking, model checking theorem proving, or static analysis, into the MDSPLE process would ensure the correctness and reliability of generated products.

REFERENCES

Alferez, G. H., & Pelechano, V. (2011a). Context-aware autonomous web services in software product lines. In *2011 15th international software product line conference* (p. 100-109). doi: 10.1109/SPLC.2011.21

Alferez, G. H., & Pelechano, V. (2011b). Systematic reuse of web services through software product line engineering. In *2011 ieee ninth european conference on web services* (p. 192-199). doi: 10.1109/ECOWS.2011.13

Apel, S., Batory, D., Kästner, C., & Saake, G. (2013). *Feature-oriented software product lines*. Berlin: Springer-Verlag. doi: 10.1007/978-3-642-37521-7

Arboleda, H., & Royer, J. C. (2012). *Model-Driven and Software Product Line Engineering*. London: Wiley-ISTE.

Aziz, A., Setyautami, M. R. A., & Azurat, A. (2019, Oct). A web-based software product line engineering framework. In *2019 international conference on advanced computer science and information systems (icacsis)* (p. 21-26). IEEE.

Brambilla, M., Cabot, J., & Wimmer, M. (2012). *Model-Driven Software Engineering in Practice*. Morgan & Claypool. doi: 10.2200/S00441ED1V01Y201208SWE001

Brambilla, M., & Fraternali, P. (2015). *Interaction Flow Modeling Language Model-Driven UI Engineering Web and Mobile Apps with IFML.* USA: Elsevier.

Brambilla, M., Mauri, A., & Umuhoza, E. (2014). Extending the interaction flow modeling language (IFML) for model driven development of mobile applications front end. In I. Awan, M. Younas, X. Franch, & C. Quer (Eds.), *Mobile web information systems* (pp. 176–191). Cham: Springer International Publishing.

Chen, L., & Ali Babar, M. (2011). A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology*, *53*(4), 344-362. (Special section: Software Engineering track of the 24th Annual Symposium on Applied Computing) doi: https://doi.org/10.1016/j.infsof.2010.12.006

Clements, P., & Northrop, L. M. (2002). *Software Product Lines: Practices and Patterns*. Boston, MA: Addison-Wesley.

Czarnecki, K., Antkiewicz, M., Kim, C. H. P., Lau, S., & Pietroszek, K. (2005). Model-driven software product lines. In *Companion to the 20th annual acm sigplan conference on object-oriented programming, systems, languages, and applications* (pp. 126–127). New York, NY, USA: ACM. doi: 10.1145/1094855.1094896

Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M., & Paolini, L. (2021). Variability Modules for Java-like Languages. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A* (p. 1–12). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3461001.3471143

Damiani, F., Hähnle, R., Kamburjan, E., Lienhardt, M., & Paolini, L. (2023). Variability Modules. *Journal of Systems and Software*, *195*, 111510. doi: https://doi.org/10.1016/j.jss.2022.111510

Echeverría, J., Pérez, F., Panach, J. I., & Cetina, C. (2021). An empirical study of performance using clone & own and software product lines in an industrial context. *Information and Software Technology*, *130*, 106444. doi: https://doi.org/10.1016/j.infsof.2020.106444

Fadhlillah, H. S., Adianto, D., Azurat, A., & Sakinah, S. I. (2018). Generating Adaptable User Interface in SPLE: Using Delta-Oriented Programming and Interaction Flow Modeling Language. In *Proceedings of the 22nd international systems and software product line conference - volume 2* (p. 52–55). New York, NY, USA: ACM.

Febrian, S. T. (2022). *Integrasi FeatureIDE dan WinVMJ Framework untuk Pengembangan Perangkat Lunak dengan Software Product Line Engineering* (Master Thesis). Universitas Indonesia, Depok, Indonesia.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design Patterns: "Elements of Reusable Object-Oriented Software"*. Addison-Wesley.

Groher, I., & Völter, M. (2009). Aspect-oriented model-driven software product line engineering. *LNCS Trans. Aspect Oriented Softw. Dev.*, *6*, 111–152. doi: 10.1007/978-3-642-03764-1_4

Hähnle, R. (2013). The abstract behavioral specification language: A tutorial introduction. In E. Giachino, R. Hähnle, F. S. de Boer, & M. M. Bonsangue (Eds.), *Formal methods for components and objects: 11th international symposium, fmco 2012, bertinoro, italy, september 24-28, 2012, revised lectures* (pp. 1–37). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-40615-7_1

Hayat, Z., Rashid, M., Azam, F., Rasheed, Y., & Waseem Anwar, M. (2021). Extension of Interaction Flow Modeling Language for Geographical Information Systems. In *2021 10th international conference on software and computer applications* (p. 186–192). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3457784.3457814

Hernández-López, J.-M., Juaréz-Martínez, U., & Sergio-David, I.-D. (2018). Automated software generation process with SPL. In J. Mejia, M. Muñoz, Á. Rocha, Y. Quiñonez, & J. Calvo-Manzano (Eds.), *Trends and applications in software engineering* (pp. 127–136). Cham: Springer International Publishing. doi: 10.1007/978-3-319-69341-5_12

Horcas, J. M., Cortiñas, A., Fuentes, L., & Luaces, M. R. (2018). Integrating the common variability language with multilanguage annotations for web engineering. In *Proceeedings of the 22nd international systems and software product line conference - volume 1, SPLC 2018, gothenburg, sweden, september 10-14, 2018* (pp. 196–207). ACM. doi: 10.1145/3233027.3233049

Horcas, J. M., Cortiñas, A., Fuentes, L., & Luaces, M. R. (2022). Combining multiple granularity variability in a software product line approach for web engineering. *Inf. Softw. Technol.*, *148*, 106910. doi: 10.1016/j.infsof.2022.106910

Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., & Steffen, M. (2012). ABS: A core language for abstract behavioral specification. In B. K. Aichernig, F. S. de Boer, & M. M. Bonsangue (Eds.), *Formal methods for components and objects.* Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-25271-6_8

Kastner, C., Thum, T., Saake, G., Feigenspan, J., Leich, T., Wielgorz, F., & Apel, S. (2009). FeatureIDE: A tool framework for feature-oriented software development. In *Proceedings of the 31st international conference on software engineering* (pp. 611–614). Washington, DC, USA: IEEE Computer Society. doi: 10.1109/ICSE.2009.5070568

Koesnadi, E. S., Setyautami, M. R. A., & Azurat, A. (2022). Domain Analysis of Payment Gateway Product Line. In *2022 International Conference on Information Technology Systems and Innovation, ICITSI 2022 - Proceedings.* IEEE.

Komarudin, O., Arrumaisha, H., & Azurat, A. (2021, mar). Design and Realisation of Reusable Artefacts for Internal Supply Chain Management in Manufacturing Company. *Journal of Physics: Conference Series*, *1811*(1), 012091. doi: 10.1088/1742-6596/1811/1/012091

Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., & Damiani, F. (2014). 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* (pp. 63–74). New York, NY, USA: ACM. doi: 10.1145/2647508.2647512

Krueger, C. W. (2002). Easing the transition to software mass customization. In F. van der Linden (Ed.), *Software product-family engineering* (pp. 282–293). Berlin, Heidelberg: Springer Berlin Heidelberg.

Laguna, M. A., & Crespo, Y. (2013). A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming*, *78*(8), 1010–1034. doi: 10.1016/j.scico.2012.05.003

Laguna, M. A., González-Baixauli, B., & Hernández, C. (2009). Product line development of web systems with conventional tools. In *Web engineering, 9th international conference, ICWE 2009, san sebastián, spain, june 24-26, 2009, proceedings* (Vol. 5648, pp. 205–212). Springer. doi: 10.1007/978-3-642-02818-2_16

Martinez, J., Lopez, C., Ulacia, E., & del Hierro, M. (2009). Towards a model-driven product line for web systems. In *5th international workshop on model-driven web engineering (mdwe).* CEUR-WS.org.

Martinez, J., Ziadi, T., Bissyandé, T. F., Klein, J., & l. Traon, Y. (2015, Nov). Automating the extraction of model-based software product lines from model variants (t). In *2015 30th ieee/acm international conference on automated software engineering (ase)* (p. 396-406). doi: 10.1109/ASE.2015.44

Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., & Saake, G. (2017). *Mastering software variability with featureide*. Springer. doi: 10.1007/978-3-319-61443-4

Muhammad, R., & Setyautami, M. R. (2016, Oct). 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)* (pp. 605–610). IEEE.

Naily, M. A., Setyautami, M. R. A., Muschevici, R., & Azurat, A. (2018). A framework for modelling variable microservices as software product lines. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 10729 LNCS). doi: 10.1007/978-3-319-74781-1_18

Nerome, T., & Numao, M. (2014). A product domain model based software product line engineering for web application. In *Second international symposium on computing and* *networking, CANDAR 2014, shizuoka, japan, december 10-12, 2014* (pp. 572–576). IEEE Computer Society. doi: 10.1109/CANDAR.2014.105

OMG. (2015). Interaction Flow Modeling Language [Computer software manual]. Retrieved from `http://www.omg.org/spec/IFML/1.0/`

OMG. (2017). OMG Unified Modeling Language (OMG UML) Version 2.5.1 [Computer software manual]. Retrieved from `https://www.omg.org/spec/UML/2.5.1/PDF`

Ouali, S., Kraïem, N., Al-Khanjari, Z., & Baghdadi, Y. (2013). A model driven software product line process for developing applications. In X. Franch & P. Soffer (Eds.), *Advanced information systems engineering workshops - caise 2013 international workshops, valencia, spain, june 17-21, 2013. proceedings* (Vol. 148, pp. 447–454). Springer. doi: 10.1007/978-3-642-38490-5_40

Pohl, K., Bockle, G., & van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles, and Techniques*. Berlin: Springer-Verlag. doi: 10.1007/3-540-28901-1

Prayoga, H. A. (2020). *"WinVMJ: Web framework berbasis variability modules for java untuk software product line engineering"* (Bachelor Thesis). Universitas Indonesia, Depok, Indonesia.

Rohma, I. A. (2022). *"pengembangan eclipse plug-in untuk ui generator pada software product line engineering"* (Bachelor Thesis). Universitas Indonesia, Depok, Indonesia.

Roubi, S., Erramdani, M., & Mbarki, S. (2016). Extending graphical part of the Interaction Flow Modeling Language to Generate Rich Internet Graphical User Interfaces. In *2016 4th international conference on model-driven engineering and software development (modelsward)* (p. 161-167).

Samuel, C. (2022). *Penyempurnaan fitur dan library software product line engineering framework winvmj* (Bachelor Thesis). Universitas Indonesia, Depok, Indonesia.

Santoso, C. L. (2023). *Static page management dan perbaikan interaction flow modeling language untuk memodelkan software product line* (Bachelor Thesis). Universitas Indonesia, Depok, Indonesia.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., & Tanzarella, N. (2010). Delta-oriented programming of software product lines. In J. Bosch & J. Lee (Eds.), *Software product lines: Going beyond* (pp. 77–91). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-642-15579-6_6

Setyautami, M. R. A. (2013). *Applying UML profile and refactoring rules to create software product line from UML class diagram* (Master Thesis). Fakultas Ilmu Komputer Universitas Indonesia.

Setyautami, M. R. A., Adianto, D., & Azurat, A. (2018). Modeling Multi Software Product Lines Using UML. In *Proceedings of the 22nd international systems and software product line conference - volume 1* (p. 274–278). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3233027.3236400

Setyautami, M. R. A., Fadhlillah, H. S., Adianto, D., Affan, I., & Azurat, A. (2020). Variability Management: Re-Engineering Microservices with Delta-Oriented Software Product Lines. In *Proceedings of the 24th acm conference on systems and software product line: Volume a - volume a.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3382025.3414981

Setyautami, M. R. A., Fadhlillah, H. S., & Azurat, A. (2021). Prices: Towards web-based product lines generator. In *Proceedings of the 25th acm international systems and software product line conference - volume a* (p. 209). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3461001.3472734`

Setyautami, M. R. A., & Hähnle, R. (2021). An Architectural Pattern to Realize Multi Software Product Lines in Java. In *15th international working conference on variability modelling of software-intensive systems.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3442391.3442401

Setyautami, M. R. A., Hähnle, R., Muschevici, R., & Azurat, A. (2016). 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* (p. 45–49). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2934466.2934479

Setyautami, M. R. A., Rubiantoro, R. R., & Azurat, A. (2019). Model-Driven Engineering for Delta-Oriented Software Product Lines. In *26th asia-pacific software engineering conference, APSEC 2019, putrajaya, malaysia, december 2-5, 2019* (pp. 371–377). Washington, DC, USA: IEEE. doi: 10.1109/APSEC48747.2019.00057

Shahin, G., & Zamani, B. (2021). Extending Interaction Flow Modeling Language as a Profile for Form-making Systems. In *2021 12th international conference on information and knowledge technology (ikt)* (p. 169-173). doi: 10.1109/IKT54664.2021.9685877

Sundermann, C., Heß, T., Engelhardt, D., Arens, R., Herschel, J., Jedelhauser, K., … Schaefer, I. (2021). Integration of UVL in FeatureIDE. In *Proceedings of the 25th acm international systems and software product line conference - volume b* (p. 73–79). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3461002.3473940

Thüm, T., Schaefer, I., Apel, S., & Hentschel, M. (2012, sep). Family-Based Deductive Verification of Software Product Lines. *SIGPLAN Not.*, *48*(3), 11–20. doi: 10.1145/2480361.2371404

Vranic, V., & Taborsky, R. (2016). Features as transformations: A generative approach to software development. *Comput. Sci. Inf. Syst.*, *13*(3), 759–778. doi: 10.2298/CSIS160128027V

Waluyo, F. P. (2022). *Integrasi ORM Hibernate Dengan Framework WinVMJ Untuk Pengembangan Aplikasi Web Berbasis Software Product Line Engineering (SPLE)* (Master Thesis). Universitas Indonesia, Depok, Indonesia.

Wilmarani, A. (2023). *Variasi dan kustomisasi antarmuka untuk front-end software product line pada adaptive information system for charity organization (aisco)* (Bachelor Thesis). Universitas Indonesia, Depok, Indonesia.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., & Wessln, A. (2012). *Experimentation in Software Engineering*. New York: Springer Publishing Company, Incorporated.

GLOSSARY

Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artifacts and exploiting the product line variability.

Configuration knowledge is a mapping between feature model and delta modules.

Core module is a module in DOP that consists of common implementation of a product line.

Delta module is a module in DOP that implements feature's variations by adding, removing, or modifying elements in the core module.

Delta oriented programming (DOP) is a paradigm to implement SPLE by defining a core module and a set of delta modules.

Domain is an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

Domain analysis is a process for capturing and representing information about applications in a domain, specifically common characteristics, variations, and reasons for variation.

Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realized.

Domain implementation is the process of developing reusable artifacts that correspond to the features identified in domain analysis.

Feature is an end-user visible characteristic of a system, represents typically domain abstraction.

Feature diagram is a graphical notation to specify a feature model.

Feature model is a list of features and an enumeration of all valid feature combinations.

Feature selection is a process to select required features in the product derivation.

Framework is an incomplete set of collaborating classes to be extended for specific use cases, such as plug-ins for browsers. Components and services are units of composition with well-defined interfaces which can be composed to build a specific product.

Mass customization is the large-scale production of goods tailored to individual customers' needs.

Multi product line (MPL) is a set of software product lines that shares commonality and variability or depends each other.

Platform is any base of technologies (reusable parts) on which other technologies or processes are built.

Product is deployed software application that is derived from a software product line.

Product derivation is a production step in application engineering, where reusable artifacts are combined based on selected features.

Software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment.

Software product line engineering (SPLE) is a paradigm to develop applications (software products) using platforms (for commonality) and mass customization (variability).

UML profile is a mechanism to extend UML notation for specific purposes.

Variability module (VM) is an extension of a software module system that captures variability at the level of modules.

Variability module for Java (VM) is an architectural pattern to realize the variability modules (VM) in Java.