# PhaDOP: A Pharo Framework for Implementing Software Product Lines using Delta-Oriented Programming and Model-Based Engineering

Boubou Thiam Niang, Giacomo Kahn, Yacine Ouzrout, Mustapha Derras, Jannik Laval

# Highlights

## PhaDOP: A Pharo Framework for Implementing Software Product Lines using Delta-Oriented Programming and Model-Based Engineering

Boubou Thiam Niang,Giacomo Kahn,Yacine Ouzrout,Mustapha Derras,Jannik Laval

- Introduction of Pharo libraries tailored for Delta-Oriented Programming.
- Combination of Model-Driven Engineering with Delta-Oriented Programming methodologies.
- Development of a truth table-based methodology for identifying potential Delta modules at both entity and model levels.
- Creation of a metamodel designed to capture reusable artifacts associated with method implementation.
- Provision of comprehensive functionalities and user-friendly graphical interfaces for efficient Delta project management.

# PhaDOP: A Pharo Framework for Implementing Software Product Lines using Delta-Oriented Programming and Model-Based Engineering

Boubou Thiam Niang[a,b,*], Giacomo Kahn[b], Yacine Ouzrout[b], Mustapha Derras[a] and Jannik Laval[b]

[a]Berger-Levrault, 361 All. des Noisetiers, Limonest, 69760, France
[b]Université Lumière Lyon 2, INSA Lyon, Universite Claude Bernard Lyon 1, Université Jean Monnet Saint-Etienne, DISP UR4570, Lyon, 69007, France

ARTICLE INFO

*Keywords*:
Code generation
Delta-Oriented Programming
Model-Based Engineering
Software Product Line

ABSTRACT

Delta-Oriented Programming is a modular, and flexible paradigm for implementing a Software Product Line. Delta-Oriented Programming involves implementing a core software product alongside a collection of Delta Modules, which encapsulate modifications that can be applied to the core software to obtain the desired product. The applicable Delta modules are activated through product configuration. The core product in Delta-Oriented Programming is mainly an object-oriented program. While Delta-Oriented Programming has the potential to enhance productivity by reusing modifications, its limited availability of tools poses a challenge in managing large-scale software systems, making it difficult for non-experts to use. Model-based engineering offers a viable solution to this problem by utilizing model artifacts instead of code snippets, simplifying product line management based on Delta-Oriented Programming. This paper presents PhaDOP, a framework for implementing Delta-Oriented Programming at the model level in the Pharo environment. The framework provides the necessary structures to construct Software Product Line architecture, organize reusable artifacts, and prepare the Delta Module through a graphical user interface. PhaDOP was evaluated through a case study.

## 1. Introduction

Today, software development projects rarely start from scratch. Instead, they often leverage legacy systems or start with basic code structures. One example that relies on foundational code structure is *Spring Initializr* [1], a tool that generates minimal code templates for the SpringBoot projects [11]. Whether leveraging legacy systems or starting from scratch, the overarching goal is accelerating development productivity by building on existing foundations. However, the need for productivity extends beyond initial development to ongoing maintenance and evolution. To address this, developers generally resort to the manual Clone-and-Own approach. As described in [6], Clone-and-Own (C&O) involves forking and adapting existing software to create desired products. While initially expedient, the C&O approach introduces long-term challenges. It results in the proliferation of similar software versions, each requiring separate maintenance efforts and testing resources. The software products are tested individually and usually require as many tests as there are available products. In addition to this duplication, which is a bad practice, the case becomes more critical when the product is distributed among different teams.

The analysis effort during a software bug occurrence is not shared. Each software instance demands dedicated analysis, tailored bug correction, and separate testing. While this method might suffice for systems with rare alterations, it becomes impractical for those undergoing constant evolution. This poses a notable challenge today, with systems swiftly adapting to emerging trends like migrating to new languages or versions or transitioning from on-premise to cloud solutions. Here is where methodologies like software product line engineering prove invaluable.

Software Product Line Engineering (SPLE) [20] is a methodology for developing a set of software products that share a common basis. This approach allows adding specific features to tailor software products for different use cases. One advantage of SPLE is that shared functionalities are developed once and can be reused across a wide range of similar software products. The product line features are developed and tested before applying to the software products for which they are required. Each software product includes mandatory features, while optional features are included based on the specifications.

Implementing a software product line involves several stages, and several tools have been proposed to support these stages of implementation for different approaches. These tools include feature localization, feature analysis, or product creation. However, there need to be more tools to implement product lines, i.e., tools that cover product derivation for industrial context, with frequent changes and large-scale management. This is even more evident in the DOP [21] paradigm as a recent paradigm. This paper presents a framework for implementing a software product line using Model-Driven Engineering (MDE) [22].

This paper extends the first idea about a model-based SPL framework introduced in [18], with significant changes. Unlike the previous version, which relied on calculating model differences from different versions suggested by an

*Boubou Thiam Niang
bouboutHiam.niang@berger-levrault.com (B.T. Niang)
ORCID(s): 0000-0002-8618-1740 (B.T. Niang)

[1] https://start.spring.io/

expert to define the Delta modules, here we use a single version of the software product model. Users can now directly create and apply Delta modules to derive the desired product, simplifying the process. The possible Delta modules are stored in a database. The paper also explains how reusable software artifacts at different granularity levels, namely classes, methods, and attributes, are organized. A metamodel for organizing the reusable artifact is proposed to consider the implementation function that can not be captured by the core product model represented by a UML class diagram [25]. In addition, this paper addresses the generation of Java code from the resulting model through the common use case used in related work, which is the *Expression product Line (EPL)* [13, 19].

The paper is organized as follows: First, we provide background information on Delta-Oriented Programming (DOP) in section 2. Section 3 introduces the proposed framework. The case study is presented in section 4. In section 5, we propose several methods to facilitate delta module identification. Implementation details of the framework are given in section 6. Section 7 evaluates the framework through a use case. Related work is discussed in section 8. Threats to validity are listed in section 9. The framework's source code is available on GitHub[2]. In addition, a demonstration video of the framework is available in the shared repository[3].

## 2. Background

This section aims to introduce the main concept around the software product line.

### 2.1. Software Product Line Engineering

A software product line is a set of software with common functionalities present in each product and is customized using variable functionalities that are reusable if needed. The methodology for developing an SPL is called software product line engineering. SPLE approach comprises two subprocesses [20]: domain engineering (DE) and application engineering (AE). DE focuses on analysis, modeling functionality, defining reusable artifacts, and creating a generic SPL architecture. In AE, the product line is configured to create a specific product. Figure 1 gives an overview of SPLE steps.

Both DE and AE are divided into two spaces. The problem space focuses on analysis and modeling, while the solution space takes on concrete aspects: implementation and product creation. The PhaDOP framework proposed in this document focuses on realizing the solution space.

However, the solution space shown in figure 1 is a general representation. The implementation in the solution space depends on the paradigm chosen. Let us go into more detail about DOP, which is the paradigm taken into account by the PhaDOP framework.

[2] https://github.com/boubouthiamniang/tool-spl-dop-mde/tree/master

[3] https://drive.google.com/drive/folders/1uiCPWTq9N0FBJJqK0u9RZ2kGD9E8CtjI?usp=sharing

### 2.2. Paradigms for Implementing Software Product Lines

Software product line engineering offers various approaches for implementing a product line. Therefore, selecting the implementation paradigms that dictate how we will execute and structure the product line implementation is essential as a preliminary step.

The literature proposes several approaches for implementing a software product line. These fall into two categories: annotative and compositional approaches. Annotative approaches are based on conditional compilation [15], one of the earliest ideas for implementing a software product line, as it is based on the well-known preprocessor technique. Compositional approaches include feature-oriented programming (FOP) [8], aspect-oriented programming [12], Delta -oriented programming [21], the most recent approach known today, the Trait-oriented approach (TOP) [4]. Although these two classifications are the most widely used in the literature, we propose to move away from this traditional classification by adding a third, transformation-based approach that includes DOP and TOP.

FOP is a programming paradigm focusing on code organization around features rather than classes or objects. It involves using a feature model, which is a high-level representation of the desired features of a software system. AOP is a programming paradigm that aims to increase modularity by enabling the separation of concerns. It identifies behavior common to several program parts and extracts it into reusable modules called aspects. These aspects can then be integrated into the application to add behavior in the appropriate places. The AOP is not deliberately created to implement SPL. This is especially true of the DOP approach, which is one of the most recent. DOP is considered an extension of FOP, as the latter concentrates on feature combinations and cannot modify existing products. DOP is a programming paradigm that enables incremental, modular modification of software systems. DOP is based on modifying a system by expressing changes in the form of deltas. Trait-Oriented Approach (TOP) [4] is a principle that combines methods independent of any class hierarchy to create a specific product.

### 2.3. Delta-Oriented Programming principle

Implementing a software product line following the DOP principle implies implementing a core product and set of changes called Delta modules that can be activated for a given configuration and applied to create a customized product. Figure 2 gives the overview of the DOP principle.

Implementing a software product line implies several terminologies. The following notions are redundant for SPL:

*Feature and feature model* feature is a configurable element of a product line. A feature model is a model that represents a set of valid feature combinations to create a product from a product line.

*Artifacts* are tangible or intangible elements of software. Artifacts include documents, diagrams, code, executables,

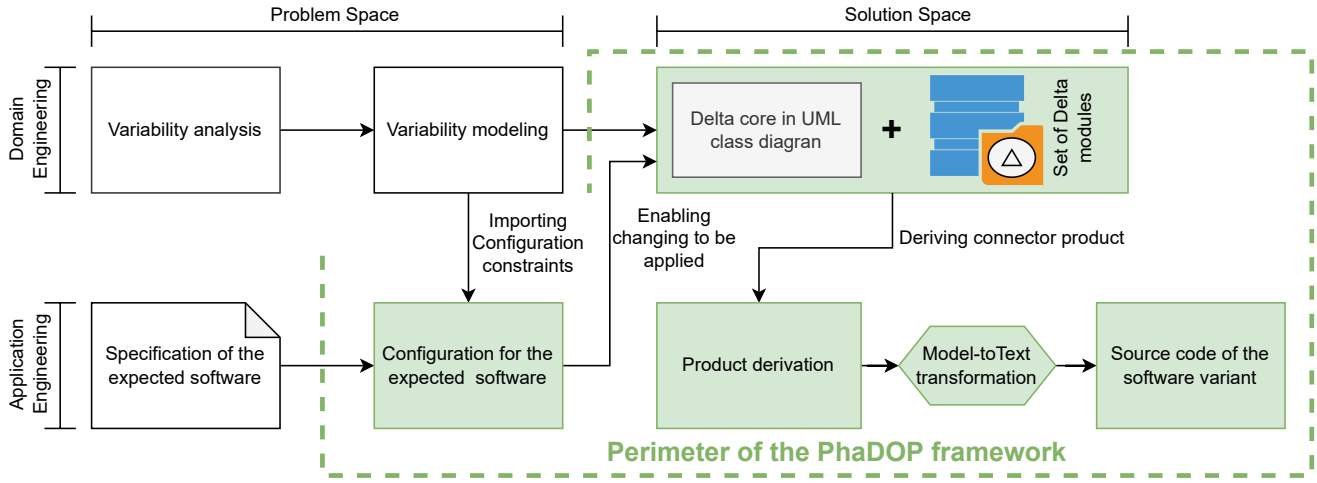


**Figure 1:** Overview of the Software Product Line Engineering (SPLE) methodology

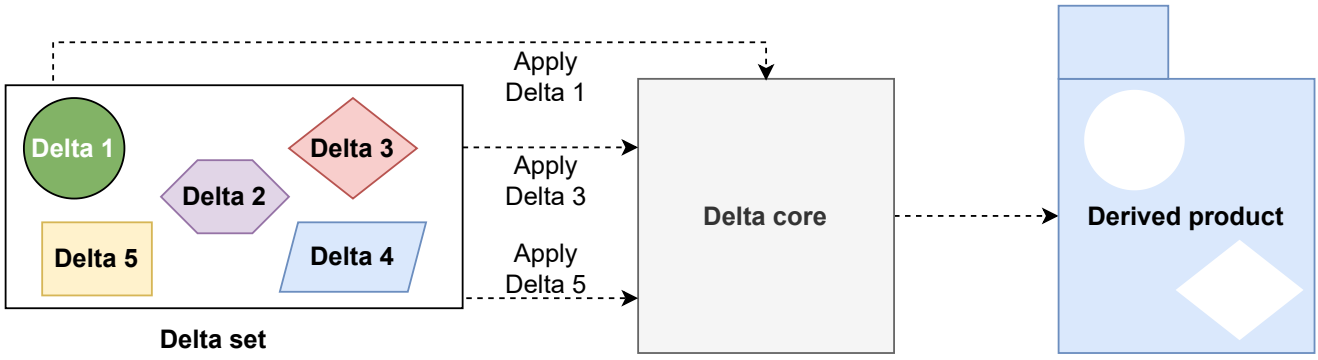


**Figure 2:** Overview of the Delta-Oriented Programming (DOP) principle

tests, databases, and others, depending on our positioning. Furthermore, classes, methods, and attributes can be regarded as code artifacts.

Specifically to DOP programming, we have several terms related to the notions of delta [19]:

*Delta action* is a set of one or more modification actions applicable to an original product to have a resulting product.

*Delta module* is a container of modification that can be applied to a product version to have another. The *Delta module* contains a set of Delta actions.

*Delta set* is a group of available delta modules. It outlines the dependencies between Delta modules and establishes potential sequences for their application when needed. For example, it can specify that one Delta module must be applied after another. The arrangement of Delta modules within the Delta set makes it easier to create new Delta modules and allows for more complex modifications that may not have been anticipated through the creation of dedicated Delta modules. This is accomplished by combining and reusing existing Delta modules.

*Delta core* The delta core is the initial product representing the SPL following the DOP paradigm. The Delta module is applied to the Delta core to create a software product variant.

*Delta core implementation strategies* Two strategies are applicable for implementing a Delta Core [21]: starting from a *Complex Core* and starting from a *Simple Core*. The *Complex Core* strategy involves creating product variants from complete products, primarily by removing features. The *Simple Core* strategy involves creating product variants from the most basic products, with only mandatory features present.

## 3. Overview of the PhaDOP framework

Despite its potential benefits, Software Product Line (SPL) adoption remains limited [23], primarily due to the lack of adequate technological support for its concrete implementation. This implementation involves articulating requirements for the product line, deriving product variants, and obtaining source code for resulting software variants. Consequently, many projects stall at the product line modeling stage, where experts use features to create specifications and configuration guides, and stakeholders rarely progress

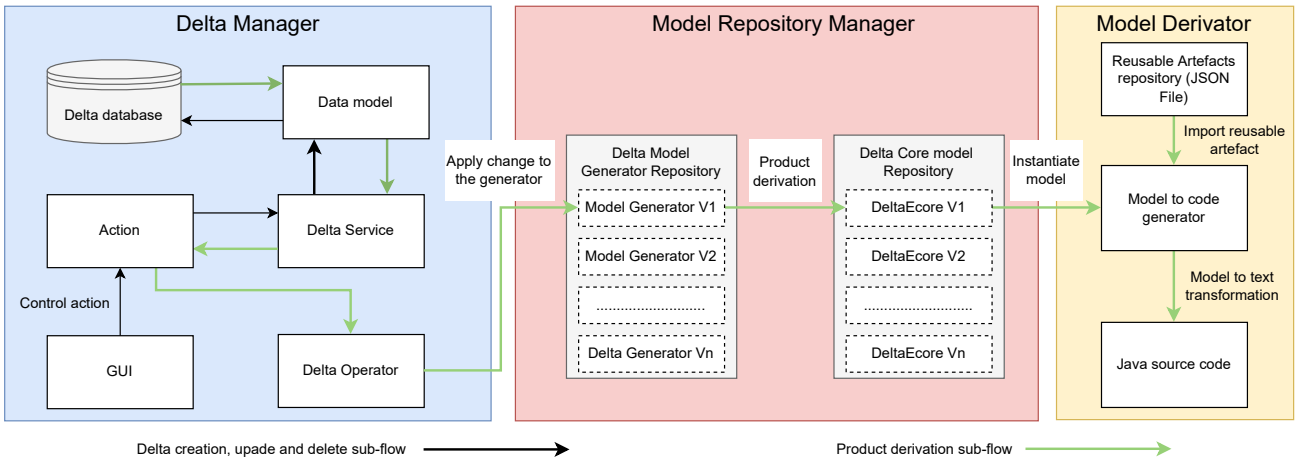


**Figure 3:** Overview of the PhaDOP framework

beyond domain engineering. Even though Delta-Oriented Programming (DOP) represents a promising approach to SPL, it faces similar technological support challenges.

This section provides an overview of prominent tools categorized by their primary focus. The literature introduces two main tools for implementing DOP in software product lines: DeltaJ [13] and SiPL [19]. DeltaJ supports essential DOP actions such as adding, removing, and updating classes and methods. However, users often find themselves manually writing substantial amounts of code for Delta operations, especially in large systems. SiPL takes a step towards automation by implementing SPL at the model level. With SiPL, Delta actions are automatically generated by analyzing differences between different versions of the core model. Nevertheless, users must manually create various versions of the model from which Delta actions are derived. This manual process can become cumbersome as it may be challenging to anticipate and create all possible versions of the core model in advance, potentially resulting in gaps in Delta module coverage.

This section introduces the PhaDOP framework for Delta-Oriented Programming at the model level. PhaDOP aims to address the shortcomings identified in existing solutions, providing a more straightforward approach to managing projects involving DOP and Delta modules, thereby enabling the use of DOP for large-scale systems. PhaDOP simplifies Delta module management by utilizing models at high levels of abstraction. This approach facilitates handling models, such as UML class diagrams, at high levels of abstraction, thereby sparing users from directly managing extensive source code. Delta operations are abstracted from users, who only need to provide information through a set of GUIs to execute fundamental DOP operations. Figure 3 provides an overview of the main components of the framework.

The PhaDOP framework consists of three primary components: the Delta Manager, the Model Repository Manager, and the Model Derivator. Each component plays a crucial role in facilitating different aspects of Delta-Oriented Programming.

## 3.1. Delta Manager

The Delta Manager consists of modules responsible for managing delta projects and delta modules, encompassing tasks such as creating, updating, and deleting delta modules.

*The PhaDOP Graphical User Interfaces (GUIs)* Managing delta projects and Delta modules can be complex when dealing with large amounts of source code or many Delta modules. The main challenge is maintaining usability when working with Delta modules. It is important to hide potentially tedious tasks such as program execution or source code manipulation. This is one of the limitations we pointed out with DeltaJ [13]. To overcome this challenge, the *PhaDOP* framework introduces a set of GUIs that allow users to interact with Delta projects and modules. These interactions include creating, updating, and deleting Delta modules or delta projects, creating model variants, and generating source code from the model.

Figure 4 depicts the initial presenter of the PhaDOP framework. The presenter on the left side of the image acts as the gateway to the framework, providing users with a sub-menu of seven GUIs, each offering different options for navigating to the desired presenter. These GUIs enable users to perform various operations for Delta-Oriented Programming at the model level. These operations involve initializing a new Delta project, creating and executing Delta modules, modifying a Delta Module, generating a model variant, generating Java code from the model, and canceling operations when necessary. On the right side is an example of a GUI to which we are redirected after selecting an option. This graphical user interface demonstrates code generation from a model variant and displays the necessary input information for the model-to-text (M2T) transformation required by the PhaDOP framework.

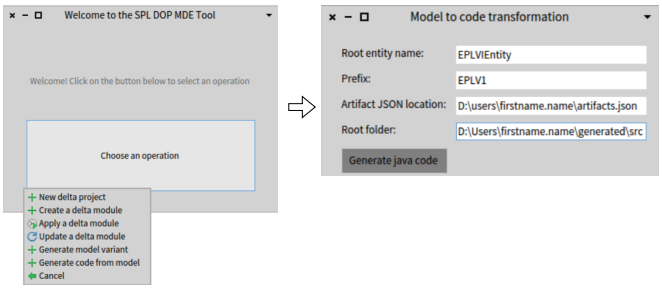


**Figure 4:** Example of graphical user interfaces in PhaDOP. The interface on the left displays all available tool options, while the code generation interface appears on the right when the corresponding option is selected.

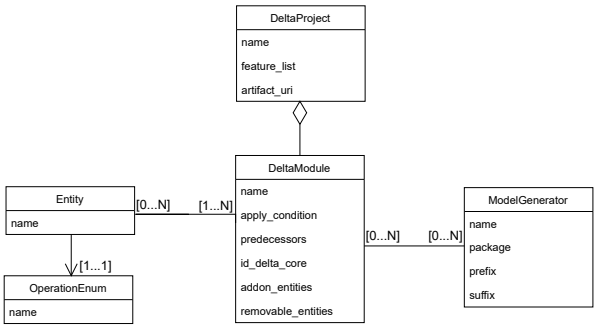


**Figure 5:** Delta data model depicting the organization of artifacts within a delta project

*Action controller* sub-component is activated by user actions within the graphical user interface (GUI). It receives requests from the GUI and directs tasks to specific Delta services or Delta actions. Delta services handle tasks related to database access, while Delta actions manage code modifications.

*Delta service* sub-component is invoked by the Action Controller sub-component to manage data access. They utilize data transfer objects to insert data into the internal database and retrieve information from it [17]. The retrieved data is then passed to the Delta Operator sub-component for code modification.

*Delta operator* sub-component enables the manipulation and application of created Delta modules for product derivation. Whenever a user initiates an action in the GUI to use a Delta module, this sub-component retrieves the relevant Delta modules and applies them to the target model generator. To ensure data integrity, the Delta Manager operates on a copy of the original model generator, meaning the original Delta Core remains unchanged.

*Delta Database* is a repository for storing essential data within the framework. The database facilitates the storage and retrieval of information for the purposes of creation, update, deletion, and application. The database contains tables designated for Delta Project, Delta Module, model generator, Entity, and their respective links. These tables offer insights into the interdependencies of project information. The Delta database reflects the Delta data model, as shown in figure 5.

A delta project comprises a set of delta modules, each of which can be applied to one or more entities. Indeed, creating a delta module implies that at least one entity is affected by an addition, update, or deletion. An entity can be influenced by zero or more delta modules. Mandatory feature entities are always present in the system and cannot be affected by any delta module except for updates. Each delta module can have one and only one operation for each entity. For example, a delta module cannot add or delete an entity. A variant of the model generator can be obtained by applying zero or more delta modules. A typical example of zero delta modules is the initial model generator, which has not yet been modified by any delta module. Conversely, a delta module can be used in zero or more model generator variants. Indeed, a delta module may not have been used for any product derivation.

## 3.2. Model Repository Manager

The Model Repository Manager component stores various versions of the model generator and the resulting models. Notably, the model version obtained after derivation, facilitated by applying delta modules, serves as a model that can be utilized for further derivations.

*Delta model generator repository* contains each model generator, which is a Pharo class with a set of methods facilitating the generation of concrete models in the form of class diagrams when needed. The initial model generator is manually created by the user within the tool. Subsequent versions of the model generator are derived from the existing model generator by applying Delta modules to the initial model generation or another variant. Each model generator variant can be the basis for creating another variant. However, the user must manually create a copy of the desired version before the derivation to prevent overwriting versions. The name of each version is stored in the delta database to track every Model generator, as depicted in Figure 5. For further details on the process of generating a model from a model generator, please refer to the link provided in footnote [4].

*Delta Core model repository* stores the UML class diagram models of concrete software product variants generated using model generator versions. Instead of directly modifying the model, our approach focuses on the generator, facilitating the generation of corresponding models as needed. The model generators maintain uniform structures, incorporating packages, classes, properties, and methods. This uniformity allows for the development and application of common methods, with specific functions utilized for delta management on a case-by-case basis. For example, it may be necessary to iterate over a model before removing an entity or its properties.

[4] https://modularmoose.org/posts/2021-02-04-coasters

### 3.3. Model Derivator

The model derivation component enables the creation of concrete-derived models.

*Reusable artifacts repository* This repository consists of a JSON file containing reusable artifacts related to method implementation. It establishes links between model entities and methods and their reusable source code. While the Delta Core captures the structural aspects of the system through a class diagram, it does not include behavioral details, limiting reusability to architectural commonalities. Consequently, derived products can only generate source code based on the project's structure, such as classes, attributes, inheritance, interfaces, and method signatures. The absence of behavioral commonalities necessitates manual code rewriting. By linking the core and reusable classes to the source code of reusable artifacts via JSON, the source code of reusable methods is centralized, facilitating their utilization across multiple products.

Figure 6 illustrates the interaction among the Delta cores, represented as a class diagram, the available Delta modules, and the JSON file representing reusable method implementations.*

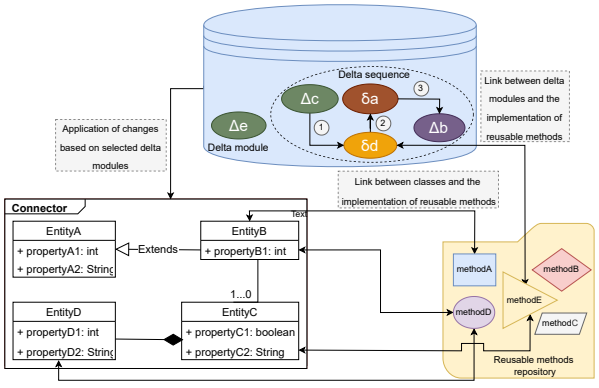


**Figure 6:** Illustrates the interaction among the Delta cores, depicted as a class diagram, and the available Delta modules.

*The Model-to-Code generator* The Model-to-Code Generator facilitates the instantiation of the product variant, which represents a model obtained through a model generator resulting from derivation. Instantiation involves assigning values to the model properties. The core method is extracted from the Reusable Artifact JSON. Additionally, we need to store the property values in the JSON file. The organization of reusable artifacts is outlined by the metamodel proposed in Figure 8. Once the connector model is instantiated, the software product's source code can be generated. The code generation from the instantiated model relies on the Famix2Java project [5] for code generation.

[5] https://github.com/moosetechnology/FAMIX2Java

## 4. Case Study

This section serves as a case study to evaluate the practicality and usability of the PhaDOP framework. The aim is to demonstrate the framework's effectiveness through a well-known use case, the Expression Product Line (*EPL*). The *EPL* has been previously studied in related research, notably in works such as DeltaJ [13] and SiPL [19], providing a solid foundation for comparison and analysis.

Figure 7 depicts the feature model of the *EPL*. This model represents all possible combinations of features that can be used to create different *EPL* variants. The model comprises two principal features, *Data*, and *Operations*. *Data* has three features *Lit*, *Add*, and *Neg*. *Lit* is mandatory, while *Add* and *Neg* are optional features. The *Operations* feature has two child features. *Print* feature is mandatory where *Eval* feature is optional.

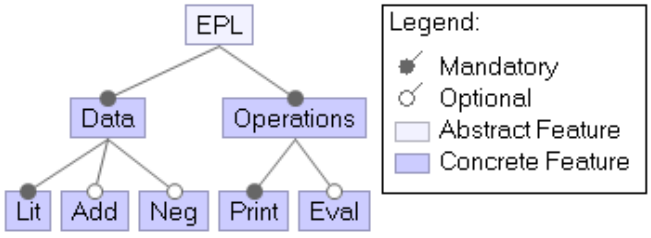


**Figure 7:** Feature Model showing the configuration capabilities of the Expression Product Line (EPL)

In this case study, we assume the product line is constructed using a reactive approach, leveraging the existing EPL legacy system implemented in Java. Segments of this implementation are presented in Listings 1, 2, 3, and 4.

Listing 1: Java implementation of the *Exp* class within the EPL legacy system

```
public class Exp {
    void print(){ }
    int eval(){
        return 0;
    }
}
```

Listing 2: Java implementation of the *Lit* class within the EPL legacy system

```
public class Lit extends Exp {
    int value;
    Lit(int n){
        this.value = n;
    }
    void print(){
        System.out.println(this.value);
    }
    int eval(){
        return this.value;
    }
}
```

Listing 3: Java implementation of the *Add* class within the EPL legacy system

```
public class Add extends Exp {
    Exp expr1;
    Exp expr2;
    Add(Exp expr1 , Exp expr2){
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    void print(){
        this.expr1.print();
```

```
        System.out.print( " + " );
        this.expr2.print();
    }
    int eval(){
        return this.expr1.eval() + this.expr2.eval();
    }
}
```

Listing 4: Java implementation of the *Neg* class within the EPL legacy system

```
public class Neg extends Exp {
    Exp expr;
    Neg(Exp expr){
        this.expr=expr;
    }
    void print(){
        System.out.print ( " ( - " );
        this.expr.print();
        System.out.println ( " ) " );
    }
    int eval(){
        return (-1) * this.expr.eval();
    }
}
```

This study aims to comprehensively evaluate the PhaDOP framework in practical software engineering contexts, covering various topics, including product line construction and code generation.

## 5. Methodologies for Improving the Identification and Management of Delta Modules

Managing Delta modules in large systems is a significant challenge when adopting the Delta-Oriented Programming (DOP) paradigm, as highlighted in the related work discussed in this article. In Pietsch et al. [19], Delta actions are automatically generated by computing the differences between different versions of the core model. However, users must manually create different versions of the model from which Delta actions are derived. This manual process becomes cumbersome because it can be difficult to anticipate and create all possible versions of the core model in advance, resulting in potential gaps in Delta module coverage.

Similarly, DeltaJ [13] supports essential DOP actions such as removing, adding, and updating classes and methods. However, users are burdened with manually writing significant amounts of code for Delta operations, especially in large systems.

This section presents a methodology that utilizes truth tables to identify Delta modules at the entity and method granularity levels.

Additionally, using a class diagram representation of the model allows for capturing features, attributes, and method signatures but not implementation details. The PhaDOP framework addresses the aforementioned limitation by introducing reusable artifacts at the method implementation level, referred to as model-level granularity reusable artifacts. To capture these artifacts, we adopt a JSON representation. Additionally, we propose a metamodel to outline the organization of the JSON file repository for model-level reusable artifacts.

| Lit | Print | Add | Neg | Eval |
|---|---|---|---|---|
| + | + | + | + | + |
| + | + | + | + | - |
| + | + | + | - | + |
| + | + | + | - | - |
| + | + | - | + | + |
| + | + | - | + | - |
| + | + | - | - | + |
| + | + | - | - | - |

**Table 1**
Truth table illustrates the presence or absence of all EPL features. A '+' indicates the feature is present, while a '-' indicates its absence.

### 5.1. Truth tables for Identifying Delta Modules at Entity and Method-Level Granularity

Delta Modules at the entity and method-level granularity refer to changes made at the class level, such as adding or removing classes and attributes. A Delta Core and a list of features and reusable artifacts are required to initialize the Delta Project. The next step involves preparing Delta Modules that apply to the Delta Core. Predicting future additions, such as new entities or methods, can be challenging, so forecasting all potential Delta Modules in advance requires effort. Identifying foreseeable changes can improve the overall process, especially when using model-based engineering. This increases productivity and makes the task more accessible to non-experts.

We propose a methodology for identifying Delta Modules that apply to a Delta Core according to the feature model. This process relies on a known truth table, where each feature has two possible states for a valid target software product: present or absent. Each feature presence test results in a true or false value. If the corresponding artifact does not exist, a Delta Module that performs an 'Add' operation indicates a feature's presence. Conversely, if the entity exists, the Delta Module that executes the removal operation indicates the absence of features. This truth table proposition does not consider the granularity of features at the properties level. It is important to note that the Delta Core, represented by a UML class diagram, only captures reusable artifacts at the entity and attribute level. At the same time, methods are managed directly from the reusable artifacts repository.

For a truth table with $n$ inputs, where $n$ is the number, there are $2^n$ possible outputs. This can be applied to our context by designating features as inputs and valid combinations to create products as outputs.

Table 1 displays the truth table for the EPL. The '+' sign indicates the presence of a feature for a given combination of table truth inputs, while the '-' sign indicates its absence. The table shows all possible configurations related to entities and methods.

It is important to note that there are mandatory features that are always present and do not change. These features are not considered variable inputs in our methodology. Therefore, the total number of inputs equals the number of optional features. The optional features include optional, group, and alternative group features since each element is occasionally optional. For 'b,' representing the number of optional features, there are a total of $2^b$ possible Delta modules. The

| Print | Add | Neg | Eval |
|---|---|---|---|
| Delta 1 | + | + | + |
| Delta 2 | + | + | - |
| Delta 3 | + | - | + |
| Delta 4 | + | - | - |
| Delta 5 | - | + | + |
| Delta 6 | - | + | - |
| Delta 7 | - | - | + |
| Delta 8 | - | - | - |

**Table 2**
Truth table illustrating potential Delta modules for adding and removing entities and methods in the EPL.

| Delta Module | Possible Delta Action | Possible Delta Action for *Complex Core* strategy |
|---|---|---|
| Delta 1 | add *Add* entity, add *Neg* entity, add *Eval* method | N.C. |
| Delta 2 | add *Add* entity, add *Neg* entity, remove *Eval* method | remove *Eval* method |
| Delta 3 | add *Add* entity, remove *Neg* entity, add *Eval* method | remove *Neg* entity |
| Delta 4 | add *Add* entity, remove *Neg* entity, remove *Eval* method | remove *Neg* entity, remove *Eval* method |
| Delta 5 | remove *Add* entity, add *Neg* entity, add *Eval* method | remove *Add* entity |
| Delta 6 | remove *Add* entity, add *Neg* entity, remove*Eval* method | remove *Add* entity, remove *Eval* method |
| Delta 7 | remove *Add* entity, remove *Neg* entity, add *Eval* method | remove *Add* entity, remove Neg entity |
| Delta 8 | remove *Add* entity, remove *Neg* entity, Remove *Eval* method | remove *Add* entity, remove *Neg* entity, remove *Eval* method |

**Table 3**
Truth table depicting potential entity-level Delta modules *Add* and *Remove* operation that are applicable to the EPL.

set of possible Delta modules has different granularities: entity and function. In this context, we focus on the entity granularity.

Table 2 depicts the truth table of the EPL, which focuses on variable features. Each row in the truth table represents a valid configuration to create an EPL variant. This enables us to identify Delta Modules.

We have chosen the implementation of DOP following the *Complex Core* strategy, focusing primarily on removal operations. This simplifies both the number and size of the Delta Modules. Table 3 illustrates the list of possible Delta actions.

We note a simplification using the *Complex Core*. Indeed, the Delta Module size is reduced, **i.e.,** the number of possible Delta Action, because we do not have operations related to mandatory features. For example, Delta has just one operation, removing the *Eval* method instead of three, as presented in Table 3. Once identified, all we have to do is implement the Delta Modules we want.

Table 2 is insufficient to show the impact of operation embedded in Delta action and difficulties caused by the number of combinations related to the Delta module. Indeed, when a configuration output gives several Delta Modules with zero or more Delta Action, this does not mean all these Delta Action operations must be applied. Remove actions are accomplished if the entity or method does not already exist. Converting the Add operations is performed if the feature is absent. This means each Delta module is itself a set of Delta Action combinations. We can see that the Delta Module itself is a truth table where the Delta Action is input. So, a Delta Module was evaluated thanks to the possible entity-level add and remove Delta Module truth table, Table 4.

| Delta Module | Possible Delta Action |
|---|---|
| Delta 2-1 | add *Add* entity |
| Delta 2-2 | add *Neg* entity |
| Delta 2-3 | remove *Eval* method |
| Delta 2-4 | add *Add* entity, add *Neg* entity |
| Delta 2-5 | add *Add* entity, remove *Eval* method |
| Delta 2-6 | add *Neg* entity, remove *Eval* method |
| Delta 2-7 | add *Add* entity, add *Neg* entity, remove *Eval* method |
| Delta 2-8 | N.C |

**Table 4**
Example of possible entity-level *Add* and *Remove* for a single Delta Module example applicable to the EPL.

If we zoom in on a Delta Module in Table 4, for example, *Delta 2*, the possible Delta Actions in the second column give $2^n - 1$ versions of the Delta module, where $n$ is the number of Delta actions. For *Delta 2*, $n = 3$ because we have three Delta actions. The $-1$ is included because a combination remains unused. For example, the composition of Delta Actions*add Add entity, add Neg entity, and remove Eval method* corresponds to the configuration of the feature model in where entities *Add* and *Neg* are present, and method *Eval* is absent.

## 5.2. A Metamodel for Managing Repositories of Reusable Artifacts

Method-level reusable artifacts are crucial in Delta-Oriented Programming (DOP) at the model level. Without these artifacts, the generated code resulting from Model-to-Text transformations would be limited to the class structure, attribute declaration, and method signature. We can generate method implementation by considering reusable artifacts at the method level. To maintain consistency within entities and Delta modules, it is essential to establish a connection between the UML class diagram representing the delta core and the actual method implementations.

To improve the management of reusable artifacts at the method level, we propose a metamodel structured as a JSON file containing an array of JSON objects. This metamodel helps to organize and link method-level reusable artifacts with the delta core entities, ensuring coherence within the system.

Figure 8 illustrates the proposed metamodel for organizing method-level reusable artifacts.

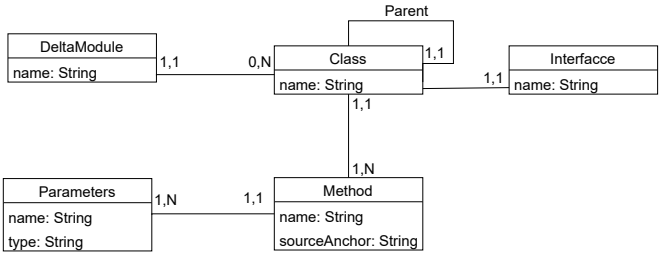


**Figure 8:** Metamodel structuring the repository for capturing reusable artifacts related to method implementation.

The JSON structure for a Delta module includes a unique identifier, referred to as the *name*, and a field called *predecessors*. The *name* serves as a distinct label for the

Delta module. In contrast, the *predecessors* field contains a comma-separated list of other Delta module names that must be applied before the current one. This list determines the order in which the modules should be applied. Each Delta module may have zero or multiple predecessor Delta modules. A Delta module comprises one or more classes corresponding to the core Delta module entity. Each class has a name and includes multiple methods. The methods within a class have their name and a source anchor that represents the source code of the methods and pertains to the reusable artifacts. If a method is present in multiple predecessor Delta modules, it will inherit the method from the last predecessor Delta module in the order. The method is redefined if a Delta module contains existing methods in any of its predecessor Delta modules. The method implementation of the Delta module itself is considered.

## 6. Implementation

The PhaDOP functionalities were realized using the Pharo language[6]. Moose [1], an open-source software and data analysis platform within Pharo, facilitated model management, aligning with the framework's utilization of Model-Driven Engineering principles. Following the precepts of Model-Driven Engineering [5], the product line was implemented utilizing platform-independent models. Delta modules are utilized to apply changes to abstract models, producing different versions through a derivation process. The source code is then generated from these derived models using model-to-model (M2M) transformation techniques [14]. The graphical user interface (GUI) development was facilitated by Spec2 [7], a specialized framework designed for GUI construction in Pharo. The framework encompasses a set of functionalities that are precisely defined. All of the code that has been implemented for PhaDOP is accessible on the project's Github repository [7].

### 6.1. Initializing the Delta Project

During the initiation of the Delta Project, the first step is to gather all necessary prerequisites for the project. This includes creating the Delta database and configuring the required tables based on the data structure according to the delta data model presented in Figure 5. During this phase, detailed specifications for the Delta Core must be completed. This includes the name of the initial Delta Core, the list of features for configuration inclusion or exclusion, and the location of method-level reusable granular artifacts that store method implementation crucial to the project. To provide this essential information, a graphical user interface is used to create the Delta Project. The GUI can be accessed by launching the tool from the starting presenter shown in Figure 4, right side and selecting the *New Delta Project* option from the sub-menu, as illustrated in Figure 9. This action guides users to the project initialization interface, called the *SpCreateDeltaProjectPresenter*. Here, users can enter the required information, such as the database name, Delta Core name, and feature list, into the presented input text fields. Separating each string value with a comma is important to ensure proper persistence in the delta database. Note that, for the Delta Project initialization, the tool provides all technical details, and the user only needs to fill in the required information in the GUI.

Figure 9 provides a visual depiction of the GUI for initializing the Delta Project with User-Provided Data.

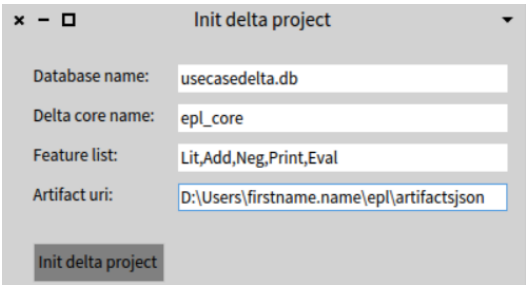


**Figure 9:** Graphical User Interface (GUI) displaying an example of User-Provided Data for initializing the Delta Project in PhaDOP.

The initialization stage concludes with creating the Delta Project database, encompassing all necessary tables storing project information. The framework utilizes SQLite as the local relational database [9]. Various functions have been implemented to store relevant Delta Core information in corresponding tables. This is made possible through invoked actions and associated services.

Figure 10 presents the sequence diagram of the initialization of a new Delta Project. The corresponding source code for

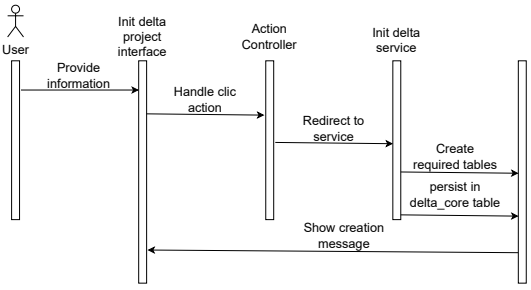


**Figure 10:** Sequence diagram depicting the interactions among PhaDOP internal components during the Delta project initialization process.

### 6.2. Creating Delta Modules

Delta Modules are modifications that can be applied to a Core Delta Module to generate valid products. The Delta Core is the initial product without modification and contains all required attributes. The entities are initially present in the Delta Core following the *Complex Core* strategy, and the cardinality between entities is defined using the feature

[6] https://pharo.org/
[7] https://github.com/bouboutthiamniang/tool-spl-dop-mde/tree/master

model. Entities with mandatory features must have a minimum cardinality of 1, while entities with optional features may have zero.

Reusable artifacts related to method implementation are organized using JSON objects. When applying the Delta Module, method implementations are linked to the corresponding entities, allowing for the addition and removal of methods as specified. Class source codes are grouped as key-value pairs.

An extract of the reusable artifacts repository for the EPL use case is presented in Listing 5. The JSON file structure adheres to the metamodel depicted in Figure 8.

Listing 5: Illustrative example of a JSON file serving as a Delta repository for managing reusable method implementations.

```
{
    "Add": {
    "targetSourceLocation":"D:\\project\\src",
        "methods":
            [
            {
            "name": "Add",
            "sourceEnchor": "{ this.expr1 = a ; this.expr2 = b ; }",
            "parameters": [
                {
                    "name": "expr1",
                "type":"Exp"
                },
                {
                    "name": "expr2",
                "type":"Exp"
                }
                ]
            },
            {
            "name": "print",
            "sourceEnchor": "{  this.expr1.print();System.out.print( \" + \" );
                    this.expr2.print(); }",
            "parameters": [
                {}
                ]
            }
            ],
        "parent": {
            "name":"Exp"
        },
        "interface": [
            {}
        ]
    }
}
```

The reusable artifacts for this Delta module are stored in the artifacts repository, as shown in Figure 5.

The Delta project is initially established by initializing the Delta Core module in conjunction with reusable artifacts. Subsequently, a Delta module is created to implement the changes made to the Delta Core module, generating a software variant.

The process involves multiple steps, including creating a Delta Module using the *SpCreateDeltaModulePresenter*. This can be done by selecting the *Create new Delta Module* option from the sub-menu on the left side of the starting presenter, as shown in Figure 4. Users provide details about the relevant database and the name of the Delta Module while specifying its application condition based on the feature list within the Delta Core table. Additionally, users specify the list of entities to be added or removed. The Delta Core ID precisely identifies the involved Delta Project. Please refer to Figure 11 for a visual representation of the textitDEvalLitAdd Delta Module creation interface.

Creating a Delta Module involves adding user-provided information to the corresponding table in the embedded Delta database. Users are relieved of technically demanding

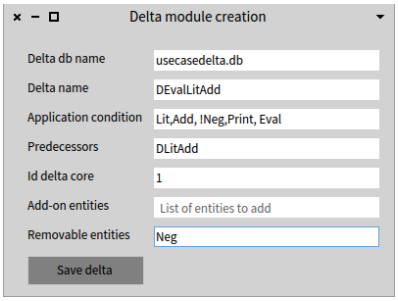


**Figure 11:** Graphical User Interface (GUI) displaying an example of User-Provided Data for creating a Delta module in PhaDOP.

tasks already implemented in the tool's source code. The code for the *SaveDeltaModule* action is provided in Listing 6.

Listing 6: Code implemented in PhaDOP for creating a Delta module using user-provided data from the GUI.

```
saveDeltaModule

  | connection dbName name applyCondition predecessors idDeltaCore addonEntities removableEntities
          idDeltaModule tabAddonEntities tabRemovableEntities|
  "Field values will become dto"
  dbName := fieldDbName text.
  name := fieldDeltaName text.
  applyCondition := fieldApplyCondition text.
  predecessors := fieldPredecessors text.
  idDeltaCore := fieldIdDeltaCore text.
  addonEntities := fieldAddonEntities text.
  removableEntities := fieldRemovableEntities text.

  "database"
  connection := SQLite3Connection memory.
  connection := SQLite3Connection on:
                  (Smalltalk imageDirectory / dbName) fullName.

  connection open.

  connection
    execute:
    'INSERT INTO delta_module(name, apply_condition, predecessors, id_delta_core,
            addon_entities, removable_entities) VALUES (?1, ?2, ?3, ?4, ?5, ?6);'
    with: {
          name.
        applyCondition.
        predecessors.
        idDeltaCore.
        addonEntities.
        removableEntities
        }.

  "Create entity and delta link"

  idDeltaModule := ((connection execute: 'Select id from delta_module where name=?' with: {name})
          next) at:'id'.
   "Todo create function for the two case duplication"
  tabAddonEntities := addonEntities splitOn: ','.
  tabAddonEntities do: [ :each |
    |entity|

    entity := (connection execute: 'Select name_entity from entity where name_entity=?' with: {
            each}) next.
    "if entity not alredy exist"
   entity ifNil: [
    each ifNotEmpty: [
      connection
        execute:
          'INSERT INTO entity(name_entity) VALUES (?1);'
           with: {
                  each
              }.
              ]

    ].
     "Create link between Delta Module and entity"
  each ifNotEmpty: [
        connection
        execute:
          'INSERT INTO delta_entity_link(id_delta, name_entity, operation) VALUES (?1, ?2, ?3
                  );'
           with: {
                  idDeltaModule.
                each.
                'ADD'
                }.
                ]
].

"Todo create function for the two case duplication"
  tabRemovableEntities := removableEntities splitOn: ','.
  tabRemovableEntities do: [ :each |
    |entity|
```

```
    entity := (connection execute: 'Select name_entity from entity where name_entity=?' with: {
            each}) next.
    "if entity not alredy exist"
   entity ifNil:  [
    each ifNotEmpty: [
      connection
        execute:
          'INSERT INTO entity(name_entity) VALUES (?1);'
          with: {
                each
              }.
              ]

    ].

     "Create link between Delta Module and entity"
     each ifNotEmpty: [
        connection
        execute:
          'INSERT INTO delta_entity_link(id_delta, name_entity, operation) VALUES (?1, ?2, ?3
              );'
          with: {
                idDeltaModule.
              each.
              'REMOVE'
              }.
              ]
].

  connection close.

  self inform: 'Delta Module ', name, ' succefully created'
```

Figure 12 presents the sequence diagram depicting the interactions among the pertinent components during the creation of a Delta module.

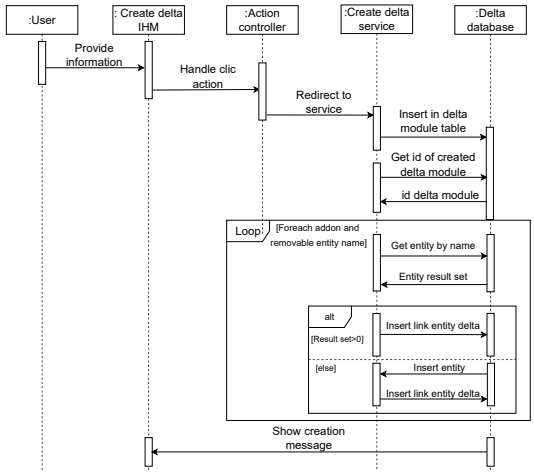


**Figure 12:** Sequence diagram depicting the interactions among PhaDOP internal components during the Delta module creation process.

## 6.3. Visualize Delta Modules

The number of Delta Modules in a large-scale system could increase to hundreds. Table 2 presents the Delta modules with only three optional features. It is important to note that the number of delta-modules related to entity and method can culminate at $2^n$. However, using Model-Driven engineering simplifies the handling of Delta Modules compared to code-level implementation. However, managing at such a scale may require advanced functionalities, particularly for users with limited expertise.

This section highlights the functionality available in PhaDOP for visualizing Delta Modules, enabling us to elucidate their inter-dependencies. Leveraging Roassal [3], a Pharo library, the visualization feature facilitates a clear depiction of the Delta Module table, showcasing the execution sequence between the Delta modules.

Figure 13 shows the visualization of the Delta Module table, which depicts the execution sequence between the Delta modules. In this fundamental example, each Delta module is represented by a gray circle. An arrow between two Delta modules indicates the order of execution, with the source of the arrow indicating the Delta module that must be applied first. The color of the arrow indicates the type of operation associated with the Delta module corresponding to the circle at the end of the arrow. Blue arrows represent addition operations, while red arrows represent removal operations.

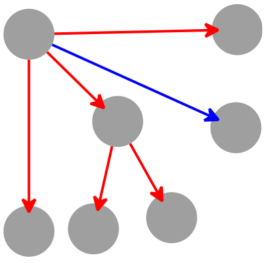

**Figure 13:** Illustrative example showcasing the visualization functionality for the execution sequence of Delta modules in PhaDOP.

## 6.4. Applying Delta Modules - Product derivation

To apply a Delta Module, users access the interface *SpApplyDeltaModulePresenter* after selecting the option *Apply Delta Module* from the sub-menu of the tool's starting GUI, as depicted in Figure 4 on the left side. Subsequently, users input the required information, including the database name, the name of the Delta Module to be applied, the name of the target model generator to be modified, the package name of the original model generator, the prefix for distinguishing entity names, and the suffix appended to the original package name. An overview of the delta application user interface is provided in Figure 14.

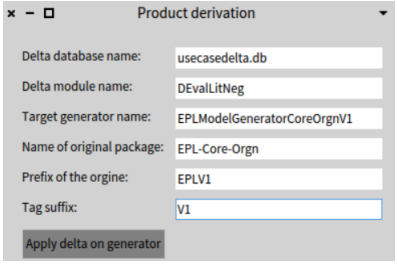


**Figure 14:** Graphical User Interface (GUI) displaying an example of User-Provided Data for applying a Delta module in PhaDOP.

After the presenter submits the data, the system retrieves information from the database and applies the necessary modifications to the target model generator. It is crucial to emphasize that users must ensure they have previously created the target model generator where the name is indicated when applying the Delta module. This model generator

is created by manually copying an existing version of the model generator, refraining from modifying the original version. The modifications are exclusively applied to the duplicate copy, ensuring the original version remains unchanged.

Listing 7 presents the method implemented in PhaDOP for creating the variant of the Core Module. Users only need to fill in the information in the dedicated GUI presented in Figure 14.

Listing 7: Code implemented in PhaDOP for applying a Delta module using user-provided data from the GUI.

```
applyDeltaModule

    | connection  dbName deltaName generatorName generatorClassName generatorClass entitiesOperationsLink
          packageName prefix suffix deltaModule deltaActionManager sourceNameDico retrievedGenerator|

  dbName := fieldDbName text.
  deltaName := fieldDeltaName text.
  generatorName := fieldGeneratorName text.
  packageName := fieldPackageName text.
  prefix := fieldPrefix text.
  suffix := fieldSuffix text.

  connection  := SQLite3Connection memory.
  connection := SQLite3Connection on:
                  (Smalltalk imageDirectory / dbName) fullName.

  connection open.

  deltaModule := (connection execute: 'Select id from delta_module where name=?' with: {deltaName})
          next.
  entitiesOperationsLink := (connection execute: 'Select name_entity, operation from
          delta_entity_link where id_delta=?' with: {(deltaModule at:'id')})"next".

  generatorClass := Smalltalk classNamed: generatorName.

  deltaActionManager := DeltaActionManager new.

  "Each entity delta link"
  (entitiesOperationsLink rows) do:[:row |
    deltaActionManager modifyGeneratorInstanceSideForDelta: generatorClass entitiesOperationsLink: row.
     ].

  sourceNameDico := Dictionary new.
  sourceNameDico at: 'packageName' put: packageName.
  sourceNameDico at: 'prefix' put: prefix.

  deltaActionManager modifyGeneratorClassSideForDelta: generatorClass sourceNameDico: sourceNameDico
          varianteSufixe:  suffix.

  "After applying the deltas modules, we save a link with the generator in db"
  connection
    execute:
    'INSERT INTO variant_generator(name, package, prefix, sufixe) VALUES (?1, ?2, ?3, ?4);'
    with: {
          generatorName.
        packageName.
        prefix.
        suffix
        }.

   "Create link between the generator and Delta Module"
   retrievedGenerator := (connection execute: 'Select id from variant_generator where name=?' with:
            {generatorName}) next.
  connection
    execute:
    'INSERT INTO delta_variant_link(id_delta, id_variant) VALUES (?1, ?2);'
    with: {
          (deltaModule at:'id').
        (retrievedGenerator at:'id')
        }.

  connection close.


  self inform: 'Delta Module', deltaName, ' successfully applied'.
```

The program passes through several framework components, as shown in the sequence diagram in Figure 15.

The Delta Module modifies the model generator methods that represent the Delta Core. The name of the variant of the model generator is stored in the database, and a relationship between the Delta Module and the model generator is established in the corresponding table. This allows tracking which Delta Modules have been applied to the model generator and which model generators are impacted by a particular Delta Module. Effective system management requires knowledge of the impact of removing a Delta Module. In PhaDOP, a variant of the model generator

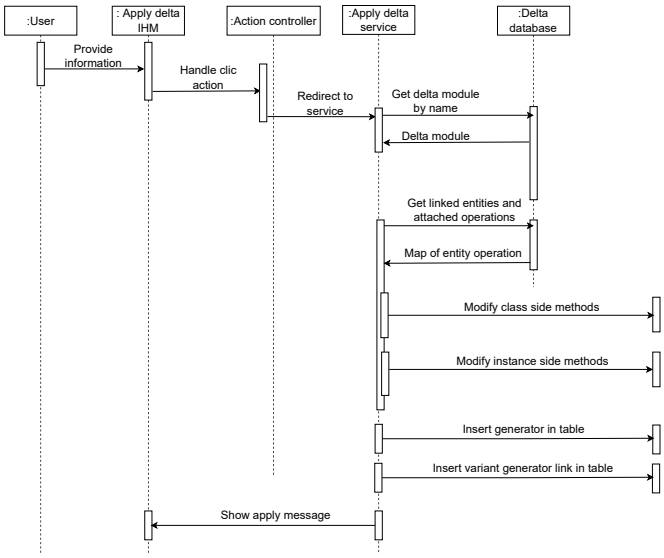


**Figure 15:** Sequence diagram depicting the interactions among PhaDOP internal components during the Delta module application process.

is created instead of the generated model. This is because the model generator always has the same structure, consisting of modified Pharo methods. This approach avoids difficulties if changes were made directly to the generated models, such as adding or removing entities and their attributes. This may require a more elaborate algorithm to be adapted to each model structure.

After applying the Delta Module, the next step is to use the dedicated GUI provided by the framework, the *SpModelGeneratorPresenter*, to generate the model from the model generator variant. Figure 16 shows the GUI for generating a model from a model generator.

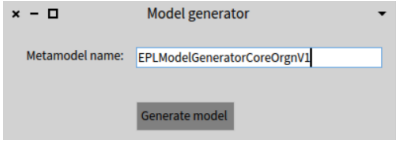


**Figure 16:** Graphical User Interface (GUI) displaying an example of User-Provided Data for generating a model from a model generator in PhaDOP.

Listing 8 displays the source code executed by PhaDOP when the user fills in the required information in the user interface for generating the model from the model generator variant and submits the data, as shown in Figure 16.

Listing 8: Code implemented in PhaDOP for generating a variant model using user-provided data from the GUI.

```
generateModel: modelName
    |class|
  class := (Smalltalk classNamed: modelName text).
  class generate.
  self inform: 'Model successfully generated'.
```

### 6.5. Generating of Product Source Code - Model-to-Text transformation

This section describes instantiating the software variant model and generating source code using PhaDOP. The tool offers Java code generation functionalities accessible through the dedicated GUI called the *SpModelToCodePresenter* interface. This interface can be accessed after making a selection in the starting interface. Figure 4 on the left side presents an overview of the tool's graphical user interface (GUI) and provides the necessary information for generating Java source code from the model variant.

The program uses data the user provides, such as the entity name, location of reusable artifacts, and root folder where the source code will be generated. It then iterates through model entities and creates a class for each entity based on the provided information. Please see 5 for more information. The program utilizes a JSON file to capture reusable artifacts associated with method implementation. It is, however, important to note that the class diagram does not encompass artifacts about method implementation. Its scope is limited to depicting the static structure of the system. The dictionary establishes connections between entities and their corresponding artifacts using the metamodel presented in Figure 8. An iterative process is employed to extract methods for each class.

The program designates the root location for generating the respective class in the source code for every class in the model. It iterates through the methods, setting the method parameters by further iterating through each method's parameters. Each method contains a source code segment that encapsulates the method's body.

Listing 9 provides the function implemented in PhaDOP that enables the generation of Java source code from the software model expressed as a class diagram. Users have to file the GUIs with the needed information.

Listing 9: Code implemented in PhaDOP for generating the Java source code from a variant model using user-provided data from the GUI.

```
generateJavaFromDeltaCore
    | rootEntityName prefix artifactsLocaton rootFolder entityList artefactsDictionary visitor|

  rootEntityName := fieldRootEntityName text.
  prefix := fieldPrefix text.
  artifactsLocaton := fieldArtifactsLocaton text.
  rootFolder := fieldRootFolder text.

entityList := (Smalltalk classNamed: rootEntityName) allSubclasses.

artefactsDictionary:= artifactsLocaton asFileReference
  readStreamDo: [ :readStream |
    (NeoJSONReader on: readStream) next ].

"Classes attribut"
entityList do: [ :class |
  |st c m package componentAnnotation componentAnnotationInstance getAnnotation getAnnotationInstance
          parentClass methodArray targetSourceLocation|
  st := FamixJavaClass new.
  st name: 'String'.
  c := FamixJavaClass new.
  c name: (class name copyFrom:prefix size + 1 to:class name size).

  class instVarNames do: [ :var |
    |currentAttribut|
    currentAttribut := FamixJavaAttribute new.
    currentAttribut name: var.
    currentAttribut declaredType: st.

    c addAttribute: currentAttribut.
    ].

  "Method"
  m := FamixJavaMethod new.
  "Get linked artefacts for current class"
  methodArray := (artefactsDictionary includesKey: c name) ifTrue: [ (artefactsDictionary at: c name) at:'
          methods']
  ifFalse: [ OrderedCollection  new.].
  methodArray do:[ :method |
          |arrayParam paramTmp|

      m := FamixJavaMethod new.
        m name: (method at:'name').
        m sourceAnchor:
          (FamixJavaSourceTextAnchor new source:
             (method at: 'sourceEnchor')).
        m parentType: c.
        m declaredType: st.
        arrayParam := OrderedCollection  new.
        (method at:'parameters') do: [ :p |
          |param paramType|
          "1 halt."
          p ifNotEmpty: [
            param := FamixJavaParameter new.
             paramType := FamixJavaClass new.
                  paramType name: (p at:'name').
             param declaredType: paramType.
             arrayParam add:param.
          ]
        ].
        "Annotations"
          getAnnotation := FamixJavaAnnotationType new name: 'MethodAnnotation'.
          getAnnotationInstance := FamixJavaAnnotationInstance new annotationType: getAnnotation.

          m annotationInstances add: getAnnotationInstance.
      c addMethod:  m.

      ].
  componentAnnotation := FamixJavaAnnotationType new name: 'ComponentAnnotation'.
  componentAnnotationInstance := FamixJavaAnnotationInstance new annotationType: componentAnnotation.
  c annotationInstances add: componentAnnotationInstance.

  "Inheritance - Parent class"
  parentClass := FamixJavaClass new
     name: 'ParentClass';
    parentPackage: package;
    yourself.

  c
    addSuperInheritance:
      (FamixJavaInheritance new
        subclass: c;
        superclass: parentClass).

  targetSourceLocation := (artefactsDictionary at:c name) at:'targetSourceLocation'.

  visitor := FAMIX2JavaVisitor new.
        "Create file"
  visitor rootFolder:'D:\Users\bouboutHiam.niang\workspace\epl_legacy_dop_tool_demo\
          ExpressionProductLineGeneratedNew\src' asFileReference.
  c accept: visitor.
    ].
```

When generating code, we utilize Famix2Java [8], a visitor designed to export FamixJava [24] models to Java code. The functionality implemented in PhaDOP focuses on model instantiation to generate Java source code.

## 7. Evaluation

This section assesses the feasibility of implementing Software Product Lines using the Delta-Oriented Programming paradigm with the PhaDOP framework. The evaluation is based on the case study in Section 4. The following research question has been formulated to assess the framework.

RQ 1: what is the amount of effort done by users comparing the PhaDOP framework

RQ 2: Does the PhaDOP framework support all steps of Solution Space implementation at the model level for a concrete use case, from the software product line implementation to source code generation?

RQ 3: Does PhaDOP generate code quality sufficient for direct use without rewriting?

RQ 4: Is the quantity of generated code sufficient to make the framework worthwhile?

### 7.1. Creating the Delta core of the EPL use case

In our case study, the software product line is constructed from the existing legacy system EPL using an approach

[8] https://github.com/moosetechnology/FAMIX2Java

known as extractive strategy [2]. Specifically, we employ a *Complex Core* strategy to implement the EPL, which involves generating product variants from the most comprehensive and valid products. In particular, this strategy emphasizes removal operations.

The PhaDOP framework uses Model-Driven Engineering to abstract the Delta Core at the model level. We create a UML class diagram using the Moose platform in the Pharo environment and language to represent the Delta Core of EPL. Currently, our focus is on developing a model generator to produce a model of the Delta Core, which is made possible by the Moose platform.

Creating Delta Core in Pharo and Moose, a model The creation of the Delta Core in Pharo and Moose, using a model generator, involves implementing several methods. In Pharo, there are class-side and instance-side methods. The class side manages behaviors and states shared among all class instances, while the instance side handles behaviors and states specific to individual instances.

Class-side methods comprise two functions: *packageName* and *prefix*. The former returns the package name where the model will be generated, while the latter provides a string value to avoid ambiguity in entity names.

The Delta Core's instance side includes the method *defineClasses* for declaring entities. In the EPL use case, the entities *Exp*, *Lit*, *Add*, and *Neg* are required. The method *defineHierarchy* establishes inheritance relationships, with *Lit*, *Add*, and *Neg* designated as subclasses of *Exp*. The method *defineProperties* specifies the necessary properties for each entity. For example, the entity *Lit* has the property *value*, while the entity *Add* has the properties *expr1* and *expr2*. The method *defineRelations* indicates the relations between entities. However, in the EPL system, entities such as *Exp*, *Add*, *Lit*, and *Neg* do not have multiplicity relations. The complete source code of the use case Delta Core is available on the provided Github repository [9]

Figure 17 shows the EPL Delta Core, which displays the necessary entities, method signatures, and attributes.

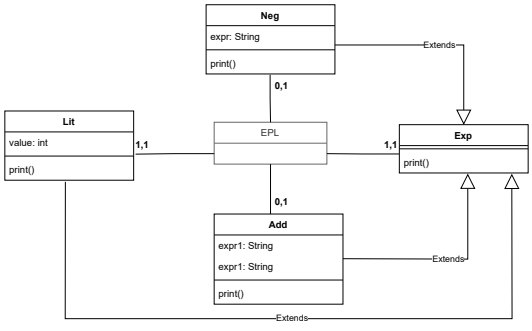


**Figure 17:** Initial Delta core of the EPL.

[9] https://github.com/boubouthiamniang/tool-spl-dop-mde/tree/master

## 7.2. Initialization the Delta Project and Delta Modules Implementation for the EPL Use Case

To begin the Delta Project for the EPL, we will use the provided GUI interface, as shown in Figure 9.

We will identify potential Delta modules using the methodology outlined in Section 5.1. As the methodology uses the *EPL* use case as an example, we will directly select the Delta modules for implementation and define the scenario accordingly.

*Delta module - DLitAdd:* In this scenario, the initial product contains entities *Lit* and *Add*, each featuring a *Print* method. We introduce the *DLitAdd Delta module* to modify the core module. Following the *Complex Core* strategy, this Delta Module evolves to remove optional features such as *Neg* and *Eval*. Referring to *Delta 4* in the truth table, Table 2). Figure 18 depicts the expected, resulting model.

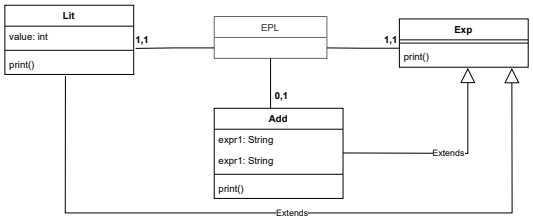


**Figure 18:** Expected variant model after applying the *DLitAdd* Delta module to the *EPL* Delta core.

*Delta module - DEvalLitAdd:* The following version aims to improve the modifications made by *DLitAdd* by adding the *Eval* method to both Lit and Add entities. Therefore, we introduce the *DEvalLitAdd* Delta Module, which depends on *DLitAdd*. It is important to note that applying *DEvalLitAdd* before *DLitAdd* would revert to the original core model. Figure 19 illustrates the resulting model.

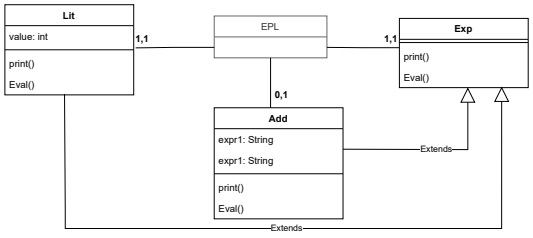


**Figure 19:** Expected variant model after applying the *DEvalLitAdd* Delta module to the EPL Core Module.

Regarding the initial Delta Modules, the reusable artifacts linked to the *DEvalLitAdd* Delta Module are also archived in the artifacts repository, as depicted in 5.

A single Delta Module with dependencies can be utilized to achieve the desired product. One of the *Neg* entities can be removed while simultaneously introducing the *Eval* method in both *Lit* and *Add* classes. This creates the *DBigEvalLitAdd* Delta Module, which encompasses the changes introduced

by *DLitAdd* along with adding the *Eval* method. This emphasizes the reusability of Delta Modules. The management of the Delta Module could be made more convenient by reusing Delta. This highlights a potential area for improvement in DeltaJ, where code-level management without MDE could be time-consuming.

In summary, for the desired product, it is recommended to use the second Delta Core and either the *DBigEvalLitAdd* Delta Module or a combination of the *DLitAdd* Delta Module with the smaller *DEvalLitAdd* Delta Module. It is important to specify the delta relation as *DLitAdd* after *DEval*, even if the latter choice is preferred. The Delta Module for the first case is presented. After initializing the Delta Project with the Delta Core and reusable artifacts, we create a Delta Module to modify the core model version for derived product creation. The desired Delta Module is determined using the scenario based on application conditions and their application order.

*Delta module - DLitNeg:* The third *Delta module* aims to construct a product that includes the *Neg* entity with both the *Print* and *Eval* methods while excluding the *Add* entity. Meanwhile, the *Lit* entity should retain both the *Print* and *Eval* methods. To achieve this, we introduce the *DLitNeg* Delta Module, which removes the *Add* entity from the Delta Core.

While the original Delta Core module already includes the *Neg* entity, additional adjustments are necessary to attain the desired product. Removing the *Add* entity from the original core generates a model variant with the *Neg* entity but without the *Add* entity. However, this results in the loss of the *Eval* method in the *Lit* entity. We propose applying the *DEvalLitAdd* Delta Module before *DEvalLitNeg* to address this. This sequence introduces the *Eval* method in both the *Add* and *Lit* entities before removing the *Add* entity.

The challenge arises from the fact that the *DEvalLitNeg* module does not contain a *Neg* entity, as it depends on the *DLitAdd* module, which removes the *Neg* entity. Therefore, the only viable solution is to transition from the *DEvalLitAdd* module by adding the *Neg* entity. Preferring a *Complex Core* strategy where removal takes precedence over addition, we select a Delta Module that removes the *Add* entity from the original core module while adding evaluation methods to both the *Lit* and *Neg* entities. This module is named *DEvalLitNeg* and is akin to *DBigEvalLitAdd* but with *Add* replaced by *Neg*. Alternatively, another approach involves first implementing a Delta Module that removes *Add* (*DLitNeg*) and subsequently introducing *DEvalLitNeg*, which relies on *DLitNeg* and adds evaluation methods to the *Lit* and *Neg* entities. Figure 20, the UML diagram of the EPL Delta core after applying *DEvalLitNeg* Delta module.

Like the previous Delta module, the repository includes reusable source code relevant to this Delta Module, as referenced in 5.

The user can easily create the described Delta Module through the dedicated user interface shown in Figure 11.

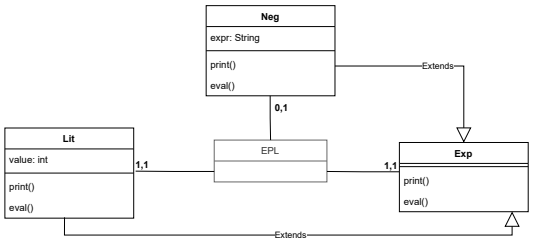


**Figure 20:** The EPL Delta Module: DEvalLitNeg

| Process steps | User task | User Effort Level (Low, Medium, High) | Coding skills required | User profil |
|---|---|---|---|---|
| Delta core creation | Pharo/Smalltalk coding | High | Yes | Expert |
| Prepare implementations of reusable methods | Create and understand JSON file | Middle | Yes | Developper |
| Delta project initialization | Fill in a GUI | Low | No | Any |
| *Delta module* creation | Fill in a GUI | Low | No | Any |
| Product derivation | Fill in a GUI | Low | No | Any |
| M2T transformation | Fill in a GUI & Create a source code folder | Low | No | Any |
| *Delta module* sequence visualization | Input text in GUI | Low | No | Any |

**Table 5**
Effort evaluation for User Tasks in implementing a Concrete Use Case with PhaDOP.

Users create a JSON file containing reusable artifacts for each Delta module, following the proposed metamodel depicted in Figure 8. An example JSON file, mirroring the structure shown in Listing 5, illustrates the organization of reusable artifacts.

## 7.3. Product derivation and Java code generation

The product derivation process utilizes the user interface on the left side, as illustrated in Figure 4. To apply the Delta Module, users select the corresponding option from the tool's home interface sub-menu and enter the required information in the user interface. After initiating the source code generation for the product variant, users must input the necessary details in the dedicated user interface for code generation. Users must manually create the folder where the source code will be generated and specify the folder location in the user interface.

## 7.4. User effort experiment results

The preceding three subsections describe the tool process used in our case study, from creating the Delta Core to generating code. To evaluate the practicality of PhaDOP, we have compiled Table 5, which outlines the steps the framework can perform automatically and those that require significant user effort.

Several characteristics have been identified to evaluate the effort required by users when using PhaDOP to implement a product line. These characteristics cover all process steps, from implementing the Core model to generating code. For each step, we have identified the manual task

that the user must perform. Additionally, we aim to assess user effort from low to high. High-level tasks require coding skills, while medium-level tasks involve updating technical files without coding skills. The table specifies if coding skills are necessary for a task and if the user profile can be completed. Based on Table 5, we can answer the first research questions, RQ1 and RQ2.

**Answer to RQ1 and RQ2:** Implementing the selected use case using PhaDOP demonstrates that it offers all the necessary functionalities for developing a software product line with reasonable effort. The only tasks that require technical skills from users are the implementation of the Delta Core and managing reusable method implementations. However, these tasks are infrequent and primarily occur at the beginning of the process. Users usually only need to input information through the graphical user interface (GUI). Specifically, 5 out of 7 steps, corresponding to 71% of the process, can be completed by users of any profile.

### 7.5. Generated code evaluation

Now, we present the resulting code generated from the model variant of the EPL system. For this purpose, we partially use Famix2Java [10], a visitor designed to export FamixJava [24] models to Java code. Specifically, it helps with the transition from constructed classes to Java files.

Listings 10, 11, and 12 demonstrate the generated classes.

Listing 10: Generated class Exp

```
@ComponentAnnotation
public class Exp extends ParentClass {
    @MethodAnnotation
    String print() {
     //Comment:nothing yet
    }
}
```

Listing 11: Generated class Lit

```
@ComponentAnnotation
public class Lit extends Exp {
    int value;
    Lit(int n){
        this.value = n;
    }
    void print(){
        System.out.println(this.value);
    }
    //Optional - added - DLitEval - Modifies Lit by introducing eval
    int eval(){
        return this.value;
    }
}
```

Listing 12: Generated class Add

```
@ComponentAnnotation
public class Add extends ParentClass {
    String expr1;
    String expr2;

    @MethodAnnotation
    String Add() {
     this.expr1 = a ; this.expr2 = b ;
    }

    @MethodAnnotation
    String print() {
    {  this.expr1.print();System.out.print( " + " ); this.expr2.print();
    }

}
```

[10] https://github.com/moosetechnology/FAMIX2Java

| | Generated artifact | Missing code |
|---|---|---|
| Class | Yes | - |
| Method signature | Yes | - |
| Method parameters | - | Yes |
| Method implementation | Yes | - |
| Class annotation | Yes | - |
| Method annotation | Yes | - |
| Package import | - | Yes |
| Implement interface | - | Yes |

**Table 6**
Table summarizing the artifacts that can be generated automatically and those that require manual intervention when working with PhaDOP

The code generated closely resembles the legacy source code presented in Section 4. It successfully creates classes, attributes, methods, constructors, and annotations similar to existing systems. However, improvements are needed in method parameter generation, and importing classes remains pending. Table 6 summarizes the artifacts that can be generated or completed manually using the current version of the tool. This observation allow us to answer the research questions RQ3 and RQ4.

Table 6 demonstrates the similarity between the generated code and the original legacy codebase used as the basis for implementing the EPL product line. To validate this, we can directly compare the corresponding source code pairs: Listing 10 with Listing 1, Listing 11 with Listing 2, and Listing 12 with Listing 3.

**Answer to RQ1 and RQ2:** After analyzing the generated code, it adheres to the expected Java structure and formatting standards. The code generated from the presented use case is directly compilable without encountering errors. However, it is important to note that when classes rely on methods imported from other classes, the current tool does not support importing, which could lead to compilation errors. While the generated code is of high quality and allows error-free execution, it may have limitations when dealing with dependencies between classes.
The amount of generated code is sufficient for the considered use case. It is important to note that the tool currently supports 63% (5 out of 8) types of artifact generation.

## 8. Related work

Few tools exist for implementing Software Product lines using Delta-Oriented Programming paradigms.

At the code level, DeltaJ [13] is a groundbreaking tool for the concrete implementation of DOP. It is based on Java syntax and supports adding, modifying, and deleting methods and class fields. ParametricDeltaJ [26] extends DeltaJ 1.5 to consider attributes as parameters. PYDOP [16] is a Python3 [11] library that implements DOP and provides transformation operations for Python modules and classes.

At the architectural level, Delta-MontiArc [10] is a delta-oriented variability modeling language designed to represent architectural variability. It provides statements for modifying architectural models defined in MontiArc. SiPL [19] is a tool

[11] https://www.python.org/

suite for implementing SPLs using the DOP paradigm at the model level. It operates at a higher level of abstraction than DeltaJ and generates code in multiple languages using model-to-text transformation.

PhaDOP addresses many challenges by relying on Model-Driven Engineering to be platform-independent and provide a set of functionalities and a user-friendly GUI. It handles the refinement from high-level abstract models to concrete code and offers a comprehensive solution for implementing SPLs using DOP paradigms.

## 9. Threats to Validity

The potential threats to the validity of our study include several aspects that could affect the interpretation and generalization of our findings. Firstly, while our tool simplifies delta management by operating at the model level, transitioning from models to code after derivation is impossible for several languages. Currently, our tool only supports code generation in Java, relying on Famix2Java for parsing. Extending this support to other languages would require the development of additional parsers. Furthermore, the emphasis on removing entities during the derivation process may limit flexibility, particularly in scenarios that require frequent addition of entities.

Secondly, the current inability of our tool to handle the sequential application of dependent Delta Modules poses a limitation. Although we have outlined how this functionality should work, it requires further refinement and experimentation. Furthermore, although we have provided configurations to activate Delta Modules, it may be necessary to exercise more stringent control during their application to ensure accurate derivation.

Thirdly, managing reusable artifacts via JSON files could also impact scalability and maintenance, potentially necessitating more efficient management solutions.

Lastly, our tool effectively handles *Complex Core* strategies, but challenges may arise in scenarios requiring support for adding entities. This limitation may be more noticeable in large-scale systems, where detecting all possible Delta Modules may be impractical.

Despite addressing the gap in related work, PhaDOP still has its limitations. These limitations highlight the importance of further refining and improving our tool to enhance its usability, flexibility, and scalability.

## 10. Conclusions

The PhaDOP framework presents a transformation-centric approach to Software Product Lines tailored for implementing SPLs using the Delta-Oriented Programming paradigm and Model-Driven Engineering. It primarily focuses on generating Object-Oriented code with an emphasis on entity removal. However, ongoing efforts are dedicated to expanding its capabilities to cover a wider spectrum of operations, including entity addition. We validate the framework's functionality by demonstrating an end-to-end process with a simple use case. We are committed to continuously improving and enhancing the PhaDOP framework.

## References


[1] Nicolas Anquetil, Anne Etien, Mahugnon H Houekpetodji, Benoit Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatiha Djareddir, Jerôme Sudich, and Moustapha Derras. Modular moose: a new generation of software reverse engineering platform. In *Reuse in Emerging Software Engineering Practices: 19th International Conference on Software and Systems Reuse, ICSR 2020, Hammamet, Tunisia, December 2–4, 2020, Proceedings 19*, pages 119–134. Springer, 2020.

[2] Wesley KG Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering*, 22(6):2972–3016, 2017.

[3] Alexandre Bergel. *Agile Visualization with Pharo: Crafting Interactive Visual Support Using Roassal*. Springer, 2022.

[4] Lorenzo Bettini and Ferruccio Damiani. Xtraitj: Traits for the java platform. *Journal of Systems and Software*, 131:419–441, 2017.

[5] Alan W Brown. Model driven architecture: Principles and practice. *Software and systems modeling*, 3:314–327, 2004.

[6] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An exploratory study of cloning in industrial software product lines. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 25–34. IEEE, 2013.

[7] Clement Dutriez, Benoît Verhaeghe, and Mustapha Derras. Switching of gui framework: the case from spec to spec 2. *International Workshop on Smalltalk Technologies*, 2019.

[8] Gabriel Coutinho Sousa Ferreira, Felipe Nunes Gaia, Eduardo Figueiredo, and Marcelo de Almeida Maia. On the use of feature-oriented programming for evolving software product lines—a comparative study. *Science of Computer programming*, 93:65–85, 2014.

[9] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. Sqlite: past, present, and future. *Proceedings of the VLDB Endowment*, 15(12):3535–3547, 2022.

[10] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented architectural variability using monticore. In *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*, pages 1–10, 2011.

[11] Siva Prasad Reddy Katamreddy and Sai Subramanyam Upadhyayula. Getting started with spring boot. In *Beginning Spring Boot 3: Build Dynamic Cloud-Native Java Applications and Microservices*, pages 29–45. Springer, 2022.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[13] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. Deltaj 1.5: delta-oriented programming for java 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 63–74, 2014.

[14] Philip Langer, Manuel Wimmer, and Gerti Kappel. Model-to-model transformations by demonstration. In *International Conference on Theory and Practice of Model Transformations*, pages 153–167. Springer, 2010.

[15] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 105–114, Cape Town, South Africa, 2010. ACM Press.

[16] Michael Lienhardt. Pydop: A generic python library for delta-oriented programming. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference-Volume B*, pages 30–33, 2023.

[17] Paul B Monday. Implementing the data transfer object pattern. In *Web Services Patterns: Java™ Platform Edition*, pages 279–295. Springer, 2003.

[18] Boubou T Niang, Giacomo Kahn, Nawel Amokrane, Yacine Ouzrout, Mustapha Derras, and Jannik Laval. Using moose platform for the implementation of a software product line according to model-based delta-oriented programming. In *IWST22—International Workshop on Smalltalk Technologies*, 2022.

[19] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. Sipl–a delta-based modeling framework for software product line engineering. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 852–857. IEEE, 2015.

[20] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer, 2005.

[21] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond: 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings 14*, pages 77–91. Springer, 2010.

[22] Douglas C Schmidt. Model-driven engineering. *Computer-IEEE Computer Society-*, 39(2):25, 2006.

[23] Maya RA Setyautami, Rafiano R Rubiantoro, and Ade Azurat. Model-driven engineering for delta-oriented software product lines. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 371–377. IEEE, 2019.

[24] Sander Tichelaar. Famix java language plug-in 1.0. *Technical Report*, 1999.

[25] Thomas von der Maßen and Horst Lichter. Modeling variability by uml use case diagrams. In *Proceedings of the International Workshop on Requirements Engineering for product lines*, pages 19–25. Citeseer, 2002.

[26] Tim Winkelmann, Jonathan Koscielny, Christoph Seidl, Sven Schuster, Ferruccio Damiani, Ina Schaefer, et al. Parametric deltaj 1.5: propagating feature attributes into implementation artifacts. In *CEUR WORKSHOP PROCEEDINGS*, volume 1559, pages 40–54. CEUR-WS, 2016.