Implementing Delta-Oriented SPLs using PEoPL: 
An Example Scenario and Case Study

Benjamin Behringer
University of Luxembourg, Luxembourg
htw saar, Germany
benjamin.behringer@htwsaar.de

Moritz Fey
htw saar, Germany
kim.moritz.fey@htwsaar.de

Abstract
Software product line implementation techniques are complementary. Thus, moving fluidly between them would be beneficial. Our tool PEoPL, a novel instantiation of the MPS language workbench, supports projecting a common variational AST into user-editable projections, each of which represents a different product line implementation technique. PEoPL supports FOP-like, annotative and product projections and allows developers to move fluidly between them. In this paper, we lay the foundation for future delta-oriented projectional editors. We use an example scenario to discuss a mapping of DeltaJ language concepts to the variational AST and operations employed in PEoPL. In a case study, we show that PEoPL is expressive enough to represent DeltaJ product lines, and at the same time supports all delta manipulations.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.13 [Software Engineering]: Reusable Software

Keywords Software product lines, delta-oriented programming, language workbenches, projectional editing

1. Introduction
A software product line (SPL) is a set of related programs sharing commonalities and variabilities [19, 20, 37], which can be distinguished in terms of end-user visible program features [9]. Selecting a set of features allows the end-user to tailor the SPL according to her requirements. Based on a feature selection, a generator produces an individual program variant automatically.

There are three major paradigms to make features explicit in an SPL’s implementation [30, 31, 43]. First, annotative approaches where special markers are scattered across the program to declare feature code [28, 30, 34, 39, 47]. Second, compositional approaches, which physically separate features into modules based on the idea of feature-oriented programming (FOP) [8, 10, 12, 38]. Third, delta-oriented, transformational approaches, which separate features into deltas that allow powerful code manipulations [33, 40, 41]. Which technique should we choose to implement a SPL [14, 29, 31, 46, 53]? Instead of requiring developers to ponder with this question, it might be beneficial to integrate implementation techniques in a common environment [14–16]. This idea is strongly supported by the structured document algebra (SDA), which suggests that implementation techniques share a common basis [13, 14]. Yet, there have been no tools supporting this claim so far.

To address this lack of tool support, we proposed an SDA-based meta-model in prior work [16]. The model defines the structure of a variational abstract syntax tree (AST), a 150% model that represents all variants of the SPL. We prototypically implemented a version of our model in a tool called PEoPL (projectional editing of product lines) [5]; a novel instantiation of the MPS language workbench [4, 36]. MPS enables mixing diverse notations such as text, symbols, tables and graphics based on projectional editors [22, 49, 50, 52]. Projectional editing does not involve parsing. Instead, developers see and interact with a projection of an AST, while any editing gesture directly changes this AST [50].

PEoPL introduces variability to MPS and supports FOP-like, annotative and product projectional editors. Developers can choose the technique best suited for their tasks, integrate techniques and move fluidly between them on demand. Our aim is to expand the set of techniques supported by PEoPL to underline its flexibility.

In this paper, we provide the basis to extend PEoPL with delta-oriented projectional editors in the future. Our goal is to underline that PEoPL can handle SPLs, which are implemented using deltas, while maintaining both expressiveness and the behavior of program variants.

We show how syntactical elements used in DeltaJ 1.5 [33], the most recent delta-oriented language for Java 1.5, map...
to the variational AST employed in PEOPL. Thereby, we elaborate how deltas compare to our notion of modules, and contrast the algebraic operations used in both approaches.

As an example and for discussion, we slightly adapted the expression product line (EPL) [35] presented in [33] for DeltaJ 1.1 (Sec. 3). To evaluate PEOPL’s capabilities, we adopted the DeltaJ SimpleTextEditor (STE) SPL (Sec. 4) [1], and compared expressiveness, redundancy and readability of PEOPL and DeltaJ STE versions using the metrics proposed by Koscielny et al. [33].

Due to the close relationship between feature- and delta-oriented approaches (i.e., both separate features into modular units), we used our FOP projection to adopt the code of each SPL. We also explored the SPLs in our annotative projection.

We contribute: (1) the basis for PEOPL’s future delta-oriented projectional editors; (2) an example scenario and a case study showing that PEOPL can handle SPLs written in DeltaJ; (3) a basic, preliminary algorithm for refactoring DeltaJ SPLs into PEOPL; (4) an online appendix with a replication package [5].

2. Background

In the following, we briefly discuss important details of PEOPL and DeltaJ.

2.1 Projectional Editing of Product Lines

Figure 1 gives an overview of our tool PEOPL [5], which we prototypically implemented using MPS [4, 36]. PEOPL renders different user-editable projections (e.g., a compositional FOP-like projection and an annotative one) on the basis of a common variational AST.

Variational AST. The variational AST is a 150% model from which product variants can be derived. Its structure is given by a language-independent meta-model [16], which we have built on the basis of the SDA—a mathematical model of feature modularization [13, 14].

Aside from language-related elements, the AST used in PEOPL contains three additional ingredients: modules, fragments and variation points (VPs) [14, 16]. A fragment is constituted of a set of syntax tree nodes. Variability is enabled through modules that assign fragments to VPs. In other words, a module contains a set of fragments that fill (multiple) VPs with content.

Module Configuration. To derive a product from the variational AST, we implemented three basic operations, which can be applied to modules [14, 16]: addition (+), subtraction (−), and overriding (→). Addition of modules allows introducing new AST elements to a variant, while subtraction allows removing them. Overriding is a combination of subtraction and addition, and thus allows replacement. The SDA defines overriding a module \( n \) by a module \( m \) as follows [14]:
\[
m \rightarrow n = n = m + (n - m).
\]

The result of a module configuration is a set of fragments, each of which will be included in the final product. All other fragments will be pruned from a transient AST. Detailed information on configuring modules can be found in [16].

SPL Declaration and Constraints. In PEOPL, we distinguish two ways for mapping a feature model’s features to modules: a basic mapping, where each module represents either the implementation of exactly one feature or the interaction of features (1); and similar to DeltaJ’s SPL declaration (explained shortly), a conditional mapping, where modules can be included conditionally using a propositional formula over features (2). In the latter, modules are not required to match the features in a feature model, and module configurations are build automatically based on a feature selection.

In both ways, developers can express domain-specific constraints such as mandatory and alternative features as well as dependencies using propositional logic—the underlying description of a feature model (cf. [11]). Implementation-specific dependencies between modules are detected automatically and need not be expressed manually.

Small Example of PEOPL’s Basics

To clarify PEOPL’s variational AST, its module configuration capabilities and basic projections, we discuss a small example we took from the STE case study (Sec. 4). Figures 2a-c show two projections and the underlying variational AST. Moreover, there are three modules: Base, Single and Multiple (Fig. 2d), each of which has a certain color. For instance, the color light gray is assigned to the module Base.

Variational AST. The AST contains a set of structural elements (e.g., a class and method declaration) as well as, four fragments \((f_0, f_{1.1}, f_{1.2} \text{ and } f_{1.3})\) that are assigned to different VPs \((v_{p_0} \text{ and } v_{p_1})\) to express variability. To visualize the assignment of fragments to VPs through modules, we use coloring. For instance, the light gray module Base assigns the fragment \( f_0 \) to the variation point \( v_{p_0} \).

Fragments annotate AST nodes. For instance, \( f_0 \) annotates the class SimpleTextEditor. We took up the subtree rule: if a node gets removed, all its children get removed as well [32]. This way, fragments act as containers.

Moreover, we use a special statement in PEOPL called FeatureBlock to group statements (i.e., siblings). Any statement is assigned to a FeatureBlock by design, either directly or through an ancestor.

Module Configuration. To avoid ambiguity in variants, a module can assign a VP only once. Yet, multiple modules can assign different fragments to the same VP, while only one fragment can fill the VP in the final product variant. In other
words, if multiple fragments are assigned to the same VP, they are mutually exclusive. For instance, all modules assign fragments to $v_{p_0}$. However, in a variant, there can be either the $\text{ThrowStatement}$, or one of the $\text{ReturnStatement}$.

The three module operations (+, −, −→) allow producing a product variant. Table 1 clarifies these operations in the context of our example. It shows different configurations and the corresponding set of calculated fragments. For instance, solely selecting Base results in: $v_{p_0}$ is filled by $f_0$ (ClassConcept) and $v_{p_1}$ by $f_{1.1}$ (ThrowStatement).

Configuration (2) shows the addition of Base and Single, which results in an erroneous set of fragments. While $v_{p_0}$ can still be filled by $f_0$, filling $v_{p_1}$ is ambiguous, as $f_{1.1}$ and $f_{1.2}$ try to fill the VP. Thus, this configuration results in an error $\updownarrow$. Configuration (3) even raises the level of ambiguity ($f_{1.1}$ vs. $f_{1.2}$ vs. $f_{1.3}$) and thus also results in an error $\updownarrow$.

Configuration (4) shows that subtraction allows removing feature details by removing conflicting fragments. We leverage the connection of $f_{1.2}$ to $v_{p_1}$ to remove $f_{1.1}$ from the set of fragments. As a result, $v_{p_1}$ remains empty.

Overriding allows selecting a fragment from a set of alternative fragments. Configurations (5), (6) and (7) show that ordering matters when using the override operator. For instance, in configuration (5), the module Base overrides Single and thus $f_{1.1}$ fills $v_{p_1}$. In configuration (6), it is the other way around, $f_{1.3}$ fills $v_{p_1}$. Note that the filling of $v_{p_0}$ by $f_0$ remains in all configurations, as overriding is just a combination of addition and subtraction.

**Projection.** PEOPL supports rendering a variational AST into different user-editable projections. Figure 2a shows a FOP-like projection of the module Single, Figure 2b an annotative one. In both projections, VPs are implicit and need not be handled manually.

The annotative projection is simple, as it is just a projection of the complete AST. To indicate variability, it shows the assignment of fragments to VPs through modules by means of colored bars (cf. [23–25, 30]). For instance, the variability of the class SimpleTextEditor is indicated by a light-gray colored bar. The aforementioned mutual exclusiveness of the statements is indicated by the $\oplus$-symbol.

The FOP projection enables product line development in a modular fashion (i.e., code of all other modules is hidden). For instance, the module Single refines the class SimpleTextEditor and therein the method getText by introducing a new return statement. The projection also indicates that the return statement is alternative to any other feature code. Note once again that editing is not performed on the concrete syntax, but directly on the underlying variational AST. Thus, it is possible to move fluidly between both projections and even edit side-by-side.

As we focus on mapping DeltaJ to PEOPL, we skip details and refer interested readers to the PEOPL-website [5].

### 2.2 Delta-oriented Programming

In **delta-oriented programming (DOP)**, features are separated into (multiple) delta modules (a.k.a. deltas) [18, 33, 40, 41]. In other words, a delta aggregates code snippets that are related to features. Instead of being a simple increment in program functionality (cf. step-wise refinement [12, 21, 42]), a delta allows three types of program manipulations: addition, modification and removal [33].

In DeltaJ, the keyword **adds** supports adding new compilation units, package information, imports, (abstract) classes, and interfaces. Moreover, fields and methods within classes and interfaces as well as the addition of new superclasses and enumerations are supported. Figure 5 shows a simple delta adding three classes.

Modification and removal are handled by the keywords **modifies** and **removes**, respectively. Both are applicable to packages, classes, interfaces, types, methods and others. Details on program manipulations can be found in [33].

To configure a product line, DeltaJ provides a so-called **product line declaration**, which allows developers to declare features and deltas, feature constraints, as well as the interconnection of features and deltas. Moreover, it allows specifying
products by a selection of features. In the following section, we discuss a product line declaration as well as all DeltaJ keywords and their application.

3. Example Scenario

As an example scenario, we chose the EPL [35], which is based on the expression problem [48]. In particular, we took the DeltaJ 1.1 EPL version presented by Koscielny et al. [33], slightly adapted it to be DeltaJ 1.5 compatible and implemented it using the DeltaJ Eclipse-plugin [1] as well as PEOPL. All results can be found at the PEOPL website [5].

Figure 3 shows the EPL’s feature model, which provides five features: Lit, Add, Neg, Print and Eval. An expression can be a literal (Lit), addition (Add) or a negation (Neg). The model requires that a product variant supports literals at least. In other words, Lit is a mandatory feature, while Add and Neg are optional ones. On top of this data, the model introduces two operations: Print and Eval. The feature Print allows outputting the expression’s textual representation to a terminal, while Eval returns the expression’s value.

In the remainder, we briefly describe an implementation of the EPL using DeltaJ (Sec. 3.1) and the way this implementation can be mapped to PEOPL’s variational AST (Sec. 3.2). We separate our discussion among problem space (stakeholder perspective) and solution space (developer perspective) [9].

3.1 EPL in DeltaJ

(i) Problem Space. Figure 4 shows the DeltaJ declaration of the EPL, which conforms to the feature model. Aside from declaring features (Line 2) and deltas (Lines 3-4), we use a constraint to specify that Lit and Print must be selected in each product variant (Line 5). This constraint corresponds to the mandatory features declared in the feature model.

The Partitions section allows introducing a mapping of features to deltas (Lines 6-13). For instance, if features Lit and Print are selected, which is the case in the product configuration called Basic (Line 14), then the delta DLitAddPrint must be applied (Line 6). If multiple deltas have to be applied, application proceeds in the ordering of the partitions (i.e., from top (DLitAddPrint) to bottom (DOptionalPrint)).

(ii) Solution Space. Next, we discuss the deltas. Figure 5 illustrates the implementation of the features Lit, Add and Print in a single delta module called DLitAddPrint that contains the base product on which all further delta manipulations are applied.

To produce a product that just contains the features Lit and Print (i.e., the mandatory ones; cf. Fig. 3), we apply the

![Figure 3. Expression product line—feature model](image-url)

![Figure 4. Product line declaration of the EPL](image-url)

![Figure 5. Delta module for features Lit, Add, Print](image-url)

![Figure 6. Delta module for removing class Add](image-url)

![Figure 7. Delta modules for feature Eval combined with Lit and Add](image-url)

![Figure 8. Delta module for feature Neg combined with Print, Eval and Add](image-url)
delta depicted in Figure 6. The delta removes the class Add from the base product. In our opinion, the application of the removes operation in this context is artificial. In particular, we could just introduce another delta that adds the class Add (Lines 12-21 in Fig. 5) to the base product. However, we wanted to show that PEOPL can indeed handle all DeltaJ operations, and thus left the product line as is (cf. [33]).

Figure 7 depicts two deltas that add evaluation functionality to the base product: DLitEval and DAddEval. Figure 8 shows the delta that introduces the feature Neg (Lines 1-8), as well as the deltas responsible for handling the interaction of Neg with Print, Eval, and Add, respectively. Thereby, for instance, DNegPrint introduces the glue code necessary for a proper cooperation of Neg and Print.

The delta DOptionalPrint (Lines 19-27) shows the modification of an already defined method on the statement level. DeltaJ supports overriding the method with the modifiers keyword, while access to the original code is given by the original keyword. Then, in a final product, there are two methods: the original one and the modified one, while the latter calls the former. This strategy is also known from FOP approaches such as FeatureHouse [8].

Each module holds a set of fragments, each of which is assigned to a VP. For instance, the module MLitAddPrint assigns $f_0$, which contains the variational AST of the class Exp, to $vp_0$ (Fig. 9b).

There are many ways to automate a refactoring from DeltaJ to PEOPL. Figure 10 shows a basic algorithm for refactoring the DeltaJ operations into PEOPL's AST. Entry point is the method consumeDelta, which calls handler methods for each delta operation (Lines 3-7). The algorithm is very basic and incomplete, and needs to be extended for a more profound automated refactoring. It is presented here to discuss the refactoring of the EPL in an algorithmic way.

**Addition.** There are four classes in the EPL: Exp, Lit, Add introduced by the delta DLitAddPrint (Fig. 5), and the class Neg added by DNeg (Fig. 8, Lines 1-8). By hiding all fragments in the variational AST, except the ones introduced by MLitAddPrint and MNeg, we obtain the exact same Java ASTs as constructed of the corresponding deltas DLitAddPrint and DNeg. Thus, adding root nodes (e.g., classes) is semantically equivalent in PEOPL and DeltaJ.

Note that just adding elements results in a simple presence condition in the variational AST (i.e., a vp is either filled by a fragment or left empty), similar to what we know from tools like CIDE [30, 32].

Figure 10 includes an algorithm, which enables an automated refactoring of DeltaJ’s adds operation into PEOPL's variational AST. The method adds takes the current delta handler (i.e., an object that holds a delta’s ASTs, which in-
cludes all manipulations) and the current target node (i.e., the node of which a copy will be added to PEoPL) as an input. In Line 11, we call addVariability, which annotates PEoPL’s copy of the target node. For each delta, there is a corresponding module (Line 39). In Lines 40, 42, and 45, we create a new fragment, assign a new VP to it and associate it with the selected module. Finally, we annotate the target node with the fragment (Line 46).

**Modification.** DeltaJ allows modifying existing classes, interfaces, methods and more. For instance, the delta DLitEval modifies the classes Exp and Lit by adding the method eval (Fig. 7, Lines 1-4). Again, this addition is a presence condition in the AST. For instance, MLitEval assigns $f_1$ to $vp$ and therewith adds the declaration of eval to the tree (Fig. 9b). The modifies algorithm reflects this refactoring by switching to adds—the next manipulation (Fig. 10, Line 15).

DeltaJ also supports modification on the statement level. For instance, DOptionalPrint modifies the method print by surrounding the original code with the printing of parentheses (Fig. 8, Lines 22, 24). The PEoPL counterpart MOptionalPrint adds the printing of parentheses to the AST (Fig. 9d, $vp_a$ and $vp_{10}$). Instead of introducing a new method and the necessary method call to a variant (DeltaJ), the information is directly embedded into the original method (Fig. 9d).

The algorithm (Fig. 10) searches for the target method in the variational AST (Line 18), and the original keyword in the DeltaJ AST (Line 19). Then, it groups all previous and next siblings of the original keyword in the respective FeatureBlocks, adds them to the variational AST and introduces variability (Lines 23-28).

However, ordering may matter. In DeltaJ, the method declaration and method call structure depends on the ordering specified in the partitions section of the product line declaration. ASTs of product variants are constructed accordingly. As ordering can be crucial for behavioral correctness, we provide an AST re-ordering mechanism with PEoPL. In particular, if ordering matters, the AST’s statements can be restructured according to the ordering specified in the module declaration (Fig. 9a). As a result, modifications in DeltaJ and PEoPL produce semantically equivalent code.

**Removal.** DeltaJ supports removing code elements. For example, the delta DRemAdd removes the class Add.

In PEoPL, we support removal of tree nodes by introducing alternative fragments and applying the subtract operation in the module configuration. For instance, to remove the class Add introduced by the module MLitAddPrint (Fig. 9d), we introduce a dummy class (i.e., MRemAdd assigns $f_{7,2}$ to $vp$). Then, the module configuration MLitAddPrint $-$ MRemAdd removes the class Add from a product variant.

The algorithm is straightforward (Fig. 10, Lines 31-35). We search the target node in the variational AST, create a dummy sibling (an empty copy of the target node), and add a fragment, which is associated with the same VP as the target node, to the dummy.

Note that we work in a projectional environment. Thus, it is possible to hide the dummy class and show a remove symbol instead. An alternative to such dummy elements would be to introduce a new annotation, which allows tagging arbitrary nodes for removal.

(ii) Problem Space. Table 2 provides an overview of the EPL module configurations, which produce variants semantically equivalent to the ones produced by the feature selections in DeltaJ (Fig. 4, Lines 14-16). For instance, in the configuration Basic, the fragments $f_0$, $f_3$ and $f_4$ are included in the final product variant. All other fragments are pruned.

As we mapped each delta module to a corresponding module in PEoPL, a basic one-to-one mapping from the feature model to our modules is not possible (cf. Sec. 2.1). Figure 11a shows the constraints we would need to specify in PEoPL’s EPL version to ensure a correct module selection in the module configuration (Table 2). For instance, MLitAddPrint must be included (Line 1). Then, if MRemAdd is selected, which removes the class Add, neither MAddEval nor MOptionalPrint can be selected anymore as they modify the class Add (Line 2). Figure 11b shows the implementation-related dependencies calculated by PEoPL automatically.

Inspired by DeltaJ, PEoPL also allows a delta-like product declaration (cf. Fig. 4). Figure 12 shows an excerpt that illustrates the conditional mapping of features to modules,
which basically corresponds to the mappings in DeltaJ (Fig. 4, Lines 6-13). The concepts of PEoPL are different to the ones of DeltaJ such that (multiple) modules, which are included based on the when-clause, have an ordering and developers need to specify the operation to apply (using a will-clause). For instance, the module $\text{REMAdd}$ will remove conflicting module details (i.e., the class Add) when the feature Add has not been selected (Fig. 12, Line 4).

Based on these mappings and feature selections, we can build the module configurations of Table 2 automatically.

### Table 2. Module configurations and results for the expression product line

<table>
<thead>
<tr>
<th>Product</th>
<th>Module configuration</th>
<th>Calculated fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>$\text{MLitAddPrint} - \text{MRemAdd}$</td>
<td>$f_0, f_3, f_4$</td>
</tr>
<tr>
<td>Partial</td>
<td>$\text{MOptionalPrint} + \text{MNeg} + \text{MLitAddPrint}$</td>
<td>$f_0, f_3, f_4, f_7, f_8, f_9, f_{10}, f_{13}, f_{14}, f_{15}$</td>
</tr>
<tr>
<td>Full</td>
<td>$\text{MOptionalPrint} + \text{MNeg} + \text{MLitAddPrint}$</td>
<td>$f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9, f_{10}, f_{11}, f_{12}, f_{13}, f_{14}, f_{15}, f_{16}, f_{17}$</td>
</tr>
</tbody>
</table>

#### a) Domain-specific constraints (manual)

1. $\text{MLitAddPrint}$
2. $\text{MNeg} \implies (\text{Add} \lor \text{Eval})$
3. $\text{MLitEval} \land \text{MRemAdd} \implies \text{MAddEval}$
4. $\text{MLitEval} \land \text{MNeg} \implies \text{MNegEval}$
5. $\text{MNeg} \implies \text{MNegPrint}$
6. $(\text{MNegPrint}) \implies (\text{MNeg} \lor \text{Eval})$

#### b) Implementation-specific constraints (automatic)

1. $\text{MLitAddPrint}$
2. $\text{MAddEval} \implies \text{MLitEval}$
3. $\text{MNeg} \implies \text{MNegEval}$
4. $\text{MNegPrint} \implies \text{MLitAddPrint}$
5. $\text{MNegEval} \implies \text{MLitEval}$

#### Figure 11. EPL module constraints

```plaintext
..Conditional mappings {
   [MLitAddPrint] when (Lit & Print);
   [MNeg]  will add when (Neg);
   [MLitEval]  will add when (Eval);
   [MRemAdd]  will remove when  (Add);
   [MAddEval]  will add when (Add & Eval);
   [MNegEval]  will add when (Neg & Eval);
   [MNegPrint]  will add when (Neg);
   [MOptionalPrint]  will add when (Add & Neg); ...
```

#### Figure 12. EPL’s mapping of features to modules

4. **Case Study and Evaluation**

Next, we investigate whether PEoPL can handle delta-oriented SPLs in a case study to lay the foundation for future delta-oriented projectional editors. All results are available in our online appendix [5]. We define two research questions:

- **RQ1.** Is the variational AST used in PEoPL expressive enough to implement a real-world, delta-oriented SPL?

- **RQ2.** How do product variants produced by PEoPL and DeltaJ compare in terms of redundancy and readability?

With RQ1, we aim to demonstrate the expressiveness of our variational AST and PEoPL’s ability to handle all common delta-operations. With RQ2, our objective is to show that product variants produced with PEoPL are slightly less redundant and easier to read (i.e., no boilerplate code), which is important as PEoPL allows the editing of concrete variants.

#### Comparability of Empirical Data.

Koscielny et al. compared DeltaJ 1.5 to the Eclipse RCP approach [33]. To ensure comparability of our PEoPL results with the DeltaJ findings of Koscielny et al. [33], we took up their evaluation methodology. In particular, our RQ1 corresponds to theirs [33].

With regard to RQ2, we took up their approach and compare product variants.

Comparing SPL implementations (i.e., not the produced products) of DeltaJ and PEoPL is not expedient, as there are no files in PEoPL and thus metrics such as lines of code are not available. Moreover, we would need to use different tools for our measurements, which would pose an additional confounding parameter.

**Subject System.** Koscielny et al. used the STE [26] product line for their study [33]. Unfortunately, the STE version they used is not publicly available. Thus, we had to choose a slightly different STE version [1] as a subject system. The STE has 11 features that are implemented using 10 deltas. A total of 128 valid product variants can be produced [33]. The DeltaJ implementation uses the Java API (java.util and java.io), Eclipse SWT API [7], and several Java 1.5 language concepts like anonymous classes and packages.

Each product variant is a text editor supporting diverse functionalities, for instance syntax highlighting. Each editor needs a text field, which can come in two different, mutually exclusive appearances: either it is a single area for typing (Single), or an area that allows editing in multiple tabs (Multiple).

To ensure comparability, we manually migrated the DeltaJ STE version to PEoPL. Thereby, we mapped each delta to a corresponding module and the product line declaration to a set of module configurations (cf. Sec. 3). In particular, we used DeltaJ to produce two product variants, which together contain all possible code artifacts. Then, we imported both products into PEoPL and used our FOP-like projection to copy and paste all delta-related code into corresponding modules. This adoption strategy was straightforward, as we were able to work with a methodology close to deltas.

**Metrics.** In the course of their DeltaJ evaluation, Koscielny et al. used DeltaJ and Eclipse RCP to produce four STE product variants each, and then compared the results. For comparability reasons, we produced the exact same variants of the STE using PEoPL. In particular, we produced a minimal basic as well as a full variant (containing all features), each in a single and tabbed text field version.

Table 3 shows the products generated with DeltaJ and PEoPL along with some metrics, which we mainly adopted from Koscielny et al. [33]. To evaluate RQ1, we used the number of files and classes, and method calls (MC) as a metric for comparison. In addition to these metrics, we used the following quantities for measuring redundancy and readability...
of product variants (RQ2): lines of code (LOC) and average methods per class (MPC) [33]. In addition, we measured the average statements per method (SPM). All measurements were conducted using the tool SourceMonitor [6].

**Results.** In addition to the raw data presented in Table 3, we show relative values between DeltaJ and PEoPL measurements in Figure 13. For instance, PEoPL’s variant of a full editor with multiple tabs (M_Full) only has 80.7% lines of the code of the corresponding DeltaJ version. Next, we discuss the results in the light of our research questions:

- **Answering RQ1.** PEoPL handles the STE example without requiring workarounds and thus we expect it to be expressive enough to handle other delta-oriented SPLs as well. In particular, compared to the DeltaJ STE implementation, PEoPL has the same number of files and classes (Table 3), as both approaches do not require glue code for orchestrating features. Moreover, PEoPL has insignificantly less method calls on average (96.7%, Fig. 13). These numbers are an indicator for the fact that we did not have to change the code’s structure during adoption.

Note that PEoPL is not bound to DeltaJ’s level of expressiveness. If required, PEoPL also allows addition, removal and modification of code on a more fine-grained level. For instance, it is possible to make changes in the middle of a method and even to a method’s parameter(s), its return type and more. Thus, it is just a matter of how such use cases are handled in a delta-oriented projectional editor. One possibility is to mix projections, for instance by integrating annotative and delta projections.

- **Answering RQ2.** The code of products produced with PEoPL tends to be slightly less redundant compared to the one produced by DeltaJ. As mentioned above, PEoPL and DeltaJ share the same amount of files and classes. Yet, there is a fundamental conceptual difference between the two. To modify a method on the statement level, DeltaJ uses the original keyword, which results in new methods and corresponding method calls in the final product. In contrast, in PEoPL, all statements get colored and thus remain in the original method. As a result, the difference between produced products tends to grow with the number of refinements on the statement level. For instance, a full STE with a single window (S_Full) in PEoPL has a smaller code base (81.1%), less methods per class (93%), and at the same time more statements per method (105.1%).

Due to this difference on the statement level, we argue that the source code of products produced with PEoPL is easier to read. In particular, products are not obfuscated with boilerplate code of generated methods. In DeltaJ, this might not be of importance as generated products are not intended for editing, anyway. However, in PEoPL it is of great importance, because projectional editing of products is possible as well. A variational AST with nested method calls as in DeltaJ would heavily increase the effort for providing such a product projection. Instead, reordering the AST seems to be easier in cases where ordering matters.

Aside from this quantitative evaluation, we manually compared each DeltaJ and PEoPL variant for semantical equivalence using DiffMerge [2], with the result that we could not find any differences.

**Threats to Validity.** We used only two, relatively small SPL implementations (EPL and STE) for comparing PEoPL and DeltaJ. Thus, we cannot draw sound conclusions for other large-scale product lines.

In our evaluation, we did not compare programming interfaces (or the concrete syntax), as there is no delta-oriented projectional editor available currently. Yet, the studies helped us to make sure that we can indeed build such editors.

Our DeltaJ measurements (Table 3) slightly differ from the ones presented by Koscielny et al. [33]. For instance, they
measured an average of 2.61 methods per class in the STE example, while we measured an average of 2.75 methods per class (Table 3). We assume that these differences are introduced by using a different version of the investigated STE. Unfortunately, the STE version used by Kosielný et al. is not available publicly. Yet, we do not consider these differences problematic, as we compared relative measurements using the same STE version in DeltaJ and PEOPL.

5. Discussion

Next, we discuss experiences gained from the EPL and STE case studies, PEOPL’s limitations, and future work.

Automated Refactoring. Refactoring DeltaJ into PEOPL SPLs is valuable future work, but a subject on its own. We presented a very basic refactoring algorithm (Fig. 10), which is limited to DeltaJ language concepts used in the EPL and STE example SPLs (except alternatives). Nevertheless, it serves as a starting point for a more advanced version.

Package Imports. In DeltaJ, developers need to deal with the Java package system. For instance, the removal of otherwise unused imports must be handled explicitly (cf. [33]). In contrast, PEOPL does not require developers to care about unused imports. In particular, PEOPL relies on the package handling mechanism of MPS: if there is no dependency to a package from within the source code, the import gets removed automatically during generation. Thus, unused imports are avoided inherently. As a result, directly modifying a package declaration itself is currently not possible in PEOPL. Modifying the package declaration corresponds to modifying a model’s name and thus would be no technical issue. Yet, we currently see no additional value in doing so.

Java 1.8 Support and Beyond. DeltaJ currently supports full Java 1.5. Providing support for Java 1.8 in PEOPL is straightforward, as MPS supports full Java 1.8. In particular, our underlying meta-model is language-independent and we simply restricted coloring of Java AST elements using an include/exclude list. On this basis, it is easy to allow coloring of, for instance, lambda expressions.

To support new languages aside from Java, we plan to build upon prior work [17], and add support for implementing variational fault trees to PEOPL, for instance.

Alternatives. Method replacement is done implicitly in DeltaJ by leaving out the original keyword in a method modification. FOP approaches like FeatureHouse do it the same way. We argue that in large methods it might be difficult to distinguish between a method replacement and refinement as one needs to search for the original keyword first. Moreover, developers are required to lift this information up into the SPL’s declaration (i.e., making the replaceability or mutual exclusiveness explicit using domain-specific constraints) and thus need to maintain future changes.

In PEOPL, we embed alternatives (i.e., multiple fragments assigned to the same VP) directly into the variational AST. The advantage of this approach is that alternatives are already explicit in the implementation and do not need to be lifted up into a constraint manually. Moreover, projectional editors are highly flexible and allow showing textual or visual markers to indicate a method’s refinement or replacement.

Current Projectional Editors. The annotative projection of PEOPL allowed us to explore the STE in a way that is closer to a standard (non-delta) Java implementation. Thereby, we revealed some duplicated statements. In particular, three language extensions for syntax highlighting were put to an extension map twice (i.e., once by the base feature and once by each syntax-highlighting feature). Due to hidden statements, the duplicated code was not easy to identify neither in its original delta-oriented version nor the FOP version in PEOPL. Thus, the annotative projection seems to be handy for identifying issues related to feature interactions. Still, the FOP projection helped us to explore the EPL and STE in a cohesive way close to deltas.

Adding Projectional Editors. Like the meta-model employed in PEOPL, the underlying technique of DeltaJ, delta modeling, is language-independent and not bound to text. Delta modeling has already been applied in various scenarios, for instance, the graphical programming environment of Matlab/Simulink [27]. The tool DeltaEcore aims automating the creation of new delta languages [45].

When adding new implementation techniques to PEOPL, we need to define projectional editors, nodes that can be annotated, and more. An automated creation would be a convenient addition and remains subject for future work.

Delta-oriented Projectional Editors. We plan to take our FOP-like projectional editors as a basis for PEOPL’s future delta-oriented editors. This way, addition and modification will be supported by the new editor directly. To enable removal, we currently use dummy elements. A more flexible way would be to introduce a new removal annotation, which could be used by developers to tag the nodes to be removed explicitly.

Due to the great flexibility of projectional editing, it is possible to design editors that conform to the DOP syntax and look exactly the same (Fig. 5 to 8). Yet, in contrast to editing plain text, a developer’s editing gestures change the AST directly. Designing editors that allow a flexible, more text-like editing can be challenging, yet is possible. We plan to provide automatic AST transformations based on string patterns to allow typing the modifies keyword for example.

6. Related Work

To our best knowledge, there has been no attempt to implement deltas using a projectional approach, yet. Thus, we distinguish only three lines of related work: integration of product line implementation techniques (1), projectional editing of product lines (2), and PEOPL-related concepts and tools (3).

Integration. There are only few approaches aiming at an integration of implementation techniques. On the theoretical
side, there are the *compositional choice calculus* (CCC), a sound formal programming language [53], and the SDA, a mathematical formalization of feature modularity [13, 14]. The CCC is language-dependent in so far as it is bound to a concrete “choice syntax”. We, in contrary, aim for a language-independent handling of choices. Aside from PEOPL, there is currently no tool that supports the concepts proposed by the SDA in the context of projectional editing [16].

In practice, there are only two other approaches allowing an integration [31, 54]. Kästner et al. refactor annotations into FOP and vice versa [31]. However, refactoring does not allow a fluid movement and necessitates workarounds [31, 44]. Walkingshaw and Ostermann build upon the CCC proposing editable views on variational programs [54]. Their approach has the idea of a variational AST in common with PEOPL. However, they pursue a language-dependent parser-based approach and thus, in contrast to PEOPL, a mixing of diverse notations and even languages is not possible. In their approach, the AST can be pretty-printed into different editable documents. Changes (add, replace, remove) made to these documents can be applied to the AST. The same capabilities for changes are available within PEOPL.

**Projetional Approaches.** To our best knowledge, there is only one language-oriented projetional approach for implementing product lines. Völter et al. discuss the implementation of product lines using *domain-specific languages* (DSLs) [51] and implement a nice way to handle presence conditions in their powerful tool for embedded software development called *mbeddr* [3]. Yet, mbeddr does not aim at integrating different implementation techniques and thus supports only an annotative projection.

**Other related Concepts and Tools.** CIDE [29] is a tool from the annotative world closely related to PEOPL. CIDE pursues a parser-based approach and stores variability in XML-files accompanying each Java compilation unit. In contrast, PEOPL embeds variability directly into the AST, while pursuing a more flexible projectional approach. The key difference is, however, that CIDE only allows adding feature details. Remove and replace operations are not available.

We already provided a detailed discussion of DOP as a closely related approach in this paper (cf. Sections 3 and 4). The key differences between PEOPL and DeltaJ are that we employ a variational AST (150% model) and leave the concrete syntax of modules open to the different projections.

7. Conclusion

To provide a basis for future delta-oriented projetional editors, we investigated whether PEOPL can handle product lines written in DeltaJ 1.5 without requiring workarounds. We discussed how to adopt the EPL written in DeltaJ using PEOPL. We mapped DeltaJ language concepts to PEOPL’s variational AST and operations, and presented a basic, preliminary algorithm to handle the refactoring. To evaluate our approach in a larger scenario, we implemented the STE product line using PEOPL and compared product variants produced by both tools.

As a result, we were able to show that PEOPL is expressive enough to handle a delta-oriented SPL. Variants produced with PEOPL tend to be slightly less redundant and better readable compared to the ones produced by DeltaJ. This is of importance as PEOPL allows editing of product variants.

Based on our observations and preliminary results, we plan to implement delta-oriented projetional editors and an automated refactoring from delta-oriented SPLs to PEOPL.

**References**


