Refactoring Delta-Oriented Software Product Lines

Sandro Schulze
TU Braunschweig
Braunschweig, Germany
sanschul@tu-bs.de

Oliver Richers
TU Braunschweig
Braunschweig, Germany
oliver.richers@gmail.com

Ina Schaefer
TU Braunschweig
Braunschweig, Germany
i.schaefer@tu-bs.de

ABSTRACT

Delta-oriented programming (DOP) is an implementation approach to develop software product lines (SPL). Delta-oriented SPLs evolve over time due to new or changed requirements and need to be maintained to retain their value. Refactorings have been proposed as behavior-preserving program transformations that improve the design and structure of (object-oriented) software systems. However, there is a lack of refactoring support for software product lines since refactoring of SPLs is more complex than of single systems. For refactoring SPLs, we have to preserve the behavior of probably thousands of programs instead of only one. In this paper, we address the refactoring of software product lines by presenting a catalogue of refactorings for delta-oriented SPLs. Additionally, we propose code smells to guide developers to potential refactoring opportunities. We show how code smells can aid the identification of SPL refactorings and how these refactorings improve the evolvability and maintainability of delta-oriented SPLs.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; D.2.13 [Software Engineering]: Reusable Software; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms

Design, Languages

Keywords

Software Product Lines, Evolution, Delta-oriented Programming, Refactoring

1. INTRODUCTION

Software Product Lines (SPL) [4, 19] gained momentum in recent years, in both, academia as well as industry. An SPL facilitates the development of a whole product portfolio instead of developing each product in isolation from scratch. To this end, a developer can manage a common set of features by means of a product line. In this context, a feature is an increment in functionality that is visible to any stakeholder. As a result, a product line increases the reusability and decreases maintenance effort and time-to-market. Recently, delta-oriented programming (DOP) [21, 23] has been proposed as a modular, yet flexible implementation technique for software product lines. DOP modularizes the features of an SPL into delta modules, which can be applied to create particular variants of the SPL.

However, similar to standalone software systems, SPLs undergo changes during their lifecycle. This process is called software evolution, and it is inherent to every software system (and every SPL) due to new or changing requirements [12, 13]. However, this evolution may lead to software aging that can potentially increase the complexity of anSPL. As a result, maintainability or reusability may decrease, which increases the effort for further SPL development. In particular, we focus on changes regarding variability modeling and implementation within DOP and do not consider other parts of the SPL engineering process [19].

An established technique to reduce complexity and to counter the process of software aging of single software systems on implementation level are refactorings [17, 7]. Refactorings are program transformations that change the internal structure of a program without altering its external (visible) behavior. Generally, such refactorings are applied in preparation for an evolutionary modification or during the process of maintenance (e.g., to clean-up the source code). When used as preparation, the aim of the refactoring is to change the internal structure of the program in a way, that the subsequent modification of its external behavior can be executed with minimum effort. When used as a clean-up, its aim is to reduce the complexity of the program and to improve the source code quality with respect to certain design guidelines. Furthermore, Fowler introduced the concept of code smells, which indicate refactoring opportunities [7].

Unfortunately, only little is known about both, refactoring and code smells, in software product lines. Moreover, tool support, which is crucial for refactoring, is non-existent. In this paper, we address this lack of refactoring support by providing refactorings for delta-oriented software product lines. In particular, we make the following contributions:

- We provide a catalogue of 23 refactorings for DOP that aim at cleaning up the SPL implementation, but also at supporting the efficient evolution of SPLs.
• To support the decision of applying refactorings, we propose code smells that are specifically tailored to delta-oriented SPL implementations.

• We show examples of evolutionary changes to delta-oriented SPLs as use cases and apply exemplary refactorings to retain a clear and maintainable structure of the SPL.

• We provide tool support for most of our refactorings as an Eclipse Plugin, which enables other researchers to reproduce or even extend our use cases.

The paper is structured as follows: In Section 2, we provide background on product lines and delta-oriented programming. In section 3, we motivate the necessity for refactorings in SPLs. Afterwards, we present our catalogue of refactorings and provide an example of how and when to use such refactorings in Section 4. We present the implementation of our refactorings in Section 5. In Section 6 we acknowledge related work before we conclude our results in Section 7.

2. SOFTWARE PRODUCT LINES

A software product line is a set of related software products that share a common set of features [4, 19]. A feature is any system property relevant (and visible) to some stakeholder, including not only the product customers, but also the software developers, analysts, system architects, administrators, etc. To create a concrete program of an SPL, the stakeholder selects the respective features (according to some requirements), and the desired product variant is generated. Through the systematic management of the commonalities and variabilities in the process of software product line engineering, a high degree of software reuse can be achieved, which increases the number of products while time to market of maintenance costs decreases [19].

To describe commonalities and variabilities in SPLs, feature models are commonly used [5, 2, 6]. Feature models can be described by feature diagrams [8] or propositional formulas [2]. In Figure 1, we show an example for a feature diagram of an ATM software product line. Within such feature diagrams, we distinguish between two kinds of features: compound features such as the feature Languages and primitive features such as the feature English. Compound features are used to group other features. Within our example, feature Cash corresponds to the functionality to withdraw money while the features English and German refer to the language used by the ATM. To express dependencies between features, we can define different constraints on both, compound and primitive features. For instance, in our example, the features Cash and Language are mandatory, which indicates that these features have to be selected in each variant. In contrast, a feature can also be optional for particular variants. Beside these constraints, we can also define constraints on feature groups, that are, sub-features of a compound feature. For example, the features English and German form an Alternative-Group, indicating that these features are mutually exclusive. Moreover, additional cross-tree constraints can be specified by means of propositional formulas capturing requires and excludes dependencies. For example, the formula $A \Rightarrow B \land C$ requires that whenever feature $A$ is selected, features $B$ and $C$ have to be selected as well.

The implementation of SPLs can be distinguished into three categories: compositional, transformational and annotative approaches. Examples for compositional approaches are feature-oriented programming (FOP) [14] or aspect-oriented programming (AOP) [9, 25], whereas the C preprocessor is an example for annotative approaches. In this paper, we focus on Delta-oriented programming (DOP) [23], a transformational approach, that subsumes feature-oriented programming. With feature-oriented programming, the general idea is to implement a specific feature of the product line in a distinct, cohesive unit, called feature module. A feature module can only extend or replace parts of other feature modules. In contrast, the delta-oriented programming approach uses the concept of delta modules which can be seen as generalized feature modules. In Table 1, we show the differences between feature and delta modules. A delta module may implement more than one feature, but may also implement only fragments of one or more features. Furthermore, a delta module can not only extend or replace existing functionality, but also remove parts of other implementations and thus the corresponding functionality. A particular product variant is generated by applying the modification of a selected set of deltas modules in a specified ordering.

---

**Table 1: Feature Modules versus Delta Modules**

<table>
<thead>
<tr>
<th>Approach</th>
<th>Features</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature Module</td>
<td>X - -</td>
<td>X X -</td>
</tr>
<tr>
<td>Delta Module</td>
<td>X X X</td>
<td>X X X</td>
</tr>
</tbody>
</table>

Delta-oriented programming supports proactive, extractive, and reactive development of software product lines [10]. In proactive development, the complete set of product variants has to be known in advance and is developed from scratch. The reactive approach develops only a basic set of products initially, and evolves the product line when new requirements or products are requested. The extractive approach starts with a set of existing, independently developed products. This set of legacy products is gradually transformed into a product line. New products can then be derived by modifying the existing products.

In Figure 2, we show a small example to illustrate the syntax of DELTAJ, an extension of JAVA for implementing delta-oriented SPLs. A delta module (here: DCash) is defined by the keyword delta. To add a class, method, or field to a delta module, we use the keyword adds. In our example, we add four classes in Line 3–8 and their initial implement-
tation, indicated by {...} which together implement the functionality to withdraw cash from the ATM. We can also alter an existing class within a delta module, for which we use the modifies keyword as in the delta modules DEnglish and DGerman which modify the Screen class to produce output in the selected language. We can also remove classes, fields, and methods. While removal of classes is indicated by the removes keyword, removal of fields and methods is indicated by their corresponding keywords removesMethod and removesField, respectively. Adding or removing fields and methods is only possible within classes that are referenced by adds or modifies.

The SPL declaration starts with the keyword spl, specifying which delta modules are applied for which product variant. First, using the keyword features, we can list all features of the feature model. Then, in a propositional formula following the keyword configurations, we can list all valid feature selections are provided along the lines of [2]. After the keyword deltas, the set of delta modules comprising the code base of the SPL is provided. An application condition is attached to every delta module in the when clause, in the form of a propositional formula over the features. If the complete formula evaluates to true for a given feature selection, the delta module is applied during product generation, otherwise it is ignored. In the example in Figure 2, each delta module is applied when its corresponding feature is selected. Additionally, the ordering in which the selected delta modules are applied, is specified by an ordered partition over the set of delta modules where a partition is enclosed by {...}. In order to guarantee that for each feature selection, a unique product variant is generated, the delta modules within one partition must be consistent, meaning that they may not change the same program entity, i.e., class, method or field. This restriction simplifies checking that the SPL is unambiguous. For generation of a particular program variant, the delta modules within one partition may be swapped, while the partitions must be applied in the specified ordering.

3. MOTIVATION

Software product lines evolve in the same way as ordinary single software systems to address new or (constantly) changing requirements [12]. For instance, a SPL for database management systems (DBMS SPL) may initially only contain some basic features, concerning the storage and retrieval of information. Later on, during its utilization and probably after some distressing experience, additional requirements such as an exhaustive logging of user activity or transaction management may be requested. The implementation of such supplementary and especially unanticipated requirements can require a complete redesign of some program parts, causing a tremendous amount of development time and cost. Even for its initial implementation, a proactive development may not be feasible, because of unclear requirements, especially regarding the future evolution of the SPL. Hence, a reactive development approach, based on an existing system, may become necessary. This, in turn, requires a flexible and modifiable code base, that can be seamlessly adapted to integrate new features or to modify existing ones.

Delta-oriented programming [23] is a viable approach to implement software product lines with mechanisms that provide both, flexibility (regarding evolutionary changes) and reusability of the code base. To implement new or to change existing functionality, a new delta module can be added that contains the required modifications to the existing code base. However, a series of such modifications, as usual during software evolution, almost inevitably increases the complexity of the internal SPL structure, especially if the modifications do not refer to an isolated new feature.

We illustrate the problems of such evolutionary changes to delta-oriented SPLs with the help of the ATM product line considered in Section 2. We suppose that the bank demands for a more advanced solution with the additional requirement that a customer can ask for his current balance.

In Figure 3, we show the implementation of the corresponding delta module DBalance, which extends the original ATM SPL by the feature Balance. This delta module removes some classes, methods and fields introduced by the delta module DCash and modifies the startExecution() method to call printBalance(). The printBalance() method implements the balance checking process: The bank is asked for the current balance of the banking account and the returned value is shown to the customer.

In the feature model part of the product line declaration in Figure 3 and in the feature digram in Figure 4, it is specified that the features Balance and Case form an alternative-group. Unfortunately, this contradicts with application conditions in the SPL declaration of the delta module DBalance, which states that the delta module DCash is always applied while DBalance is only applied if the corresponding feature Balance is selected. The reason for that incompatibility is that delta module DCash not only implements the feature Cash, but also the common base of both features Balance and Cash.

```java
delta DCash {
  adds class Bank {...}
  adds class Screen {...}
  adds class CashDispenser {...}
  adds class Controller {
    Bank bank;
    Screen screen;
    CashDispenser cashDispenser;
    void startExecution() {
      this.executeWithdrawal();
    }
  }
  void executeWithdrawal() {
    int amount = this.screen.askForAmount();
    this.bank.withdrawCash(amount);
    this.cashDispenser.dispense(amount);
  }
}

delta DEnglish {
  modifies Screen {...}
}

delta DGerman {
  modifies Screen {...}
}

spl Atm {
  features Cash, English, German
  configurations Cash && (English && !German || German && !English)
  deltas
  [DCash when Cash]
  [DEnglish when English]
  [DGerman when German]
}
```

Figure 2: Implementation of the ATM SPL
4. DELTA-ORIENTED REFACTORINGS

Refactorings [17, 7] for single programs improve the internal program structure without changing its external behavior, which is an important technique to support the evolution of software (and software product lines). Refactorings of software product line improve the structure of the product line without changing the behavior of the single products that can be derived. In this section, we present a catalogue of 23 refactorings of delta-oriented SPLs. First, we give an overview of these refactorings, which are summarized in Table 2 and 3. Second, we present use cases and examples for the application of particular refactorings by means of the introduced ATM SPL. Finally, we discuss important aspects of our proposed refactorings such as generalizability.

4.1 Overview

The proposed refactorings address all entities of delta-oriented SPLs, that is the delta modules and the SPL declaration, i.e., the feature model constraint, the application conditions and the delta module ordering. Hence, we first introduce the terminology for the proposed refactorings which refers to the respective target of each refactoring. Whenever the name of a refactoring refers to Feature, the refactoring addresses the feature model. In the context of DELTAJ, such a refactoring affects the part which starts with the features keyword (e.g., Figure 3, Line 17). In the same way, a refactoring that refers to Product Line targets at the product line declaration itself, indicated by the spl keyword (e.g., Figure 3, Line 16). A refactoring that refers to Delta Module targets a delta module (indicated by the keyword delta). A couple of refactorings refer to an Action. These refactorings refer to a concrete action within a delta module which is indicated by the keywords add, modifies, or removes. In a more general way, refactorings that refer to Delta Actions also address logical actions in delta modules. A logical delta action consists of statements that do not contain any of the aforementioned keywords, but are implicitly part of a modification specified in a delta module. For instance, in Figure 3, the statement in Line 11 is a usual variable assignment, but because it is part of an action (indicated by add void printBalance()), it is considered a delta action as well.
<table>
<thead>
<tr>
<th>Name of Refactoring</th>
<th>Typical Situation</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename Feature</td>
<td>The meaning of a feature has changed or its name is badly chosen.</td>
<td>Assign a new and appropriate name to the feature.</td>
</tr>
<tr>
<td>Rename Delta Module</td>
<td>The content of a delta module has changed or its name is badly chosen.</td>
<td>Assign a new and appropriate name to the delta module.</td>
</tr>
<tr>
<td>Rename Product Line</td>
<td>The name of a software product line does not reflect its purpose.</td>
<td>Assign a new and appropriate name to the software product line.</td>
</tr>
<tr>
<td>Extract Delta Action</td>
<td>The application condition of a single delta action is subject to be changed.</td>
<td>Move the delta action into a new delta module, retaining its application condition in the first step. Afterwards, the application condition can be changed as necessary.</td>
</tr>
<tr>
<td>Extract Conflicting Actions</td>
<td>Two delta modules with conflicting delta actions shall be made consistent, so that they can be used in the same product. Two delta actions are in conflicting if they modify the same program entity in a different way.</td>
<td>Extract the conflicting delta actions into separate modules.</td>
</tr>
<tr>
<td>Resolve Modification Action</td>
<td>A modification action shall be eliminated without changing the behavior of the software product line.</td>
<td>Change the application conditions of preceding delta actions so that the modification action can be converted into an addition action.</td>
</tr>
<tr>
<td>Resolve Removal Action</td>
<td>A removal action shall be eliminated without changing the behavior of the software product line.</td>
<td>Change the application conditions of preceding delta actions so that the removal action can be removed.</td>
</tr>
<tr>
<td>Merge Delta Modules with</td>
<td>Two delta modules of the same or consecutive partition parts have equivalent application conditions.</td>
<td>Move the delta actions from one delta module into the other delta module and remove the remaining empty delta module.</td>
</tr>
<tr>
<td>Equivalent Conditions</td>
<td>Two delta modules have the same content, and thus one delta module is redundant.</td>
<td>Combine the application conditions of the delta modules using the boolean or-operator and remove one of the modules.</td>
</tr>
<tr>
<td>Merge Delta Modules with Equivalent Content</td>
<td>The delta actions of one delta module shall be incorporated into another delta module.</td>
<td>The second delta module is merged into the first delta module by applying its delta actions to the first delta module. An inverse delta module is created to revert this merge.</td>
</tr>
<tr>
<td>Merge Delta Modules with Inverse</td>
<td>The feature model of the product line shall be changed to support more feature configurations.</td>
<td>Merge the propositional formula describing all valid feature configurations into all application conditions, using the boolean and-operator.</td>
</tr>
<tr>
<td>Extract Configurations from</td>
<td>The application conditions of the delta modules are redundant with respect to the feature model constraint, e.g. some unnecessary checks for invalid feature conditions are made.</td>
<td>Simplify the application conditions with respect to the valid feature configurations defined by the feature model.</td>
</tr>
<tr>
<td>Conditions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Refactorings of DeltaJ Software Product Lines (1/2)

We present our refactorings in the spirit of the well-known and commonly accepted object-oriented refactorings of Fowler et al. [7]. Hence, in Table 2 and 3, we group our refactorings by their purpose and provide a short explanation regarding typical situations and recommended actions for each of them.

In the following, we roughly describe each group of refactorings. The first group in Table 2 focuses on renaming different entities of delta-oriented SPLs such as features or delta modules. Renaming is a common task during evolution, for example, to better reflect the purpose of a particular entity or to avoid duplicated names. The second group extracts delta actions into a new delta module. This may be necessary due to conflicts between particular actions or for changing application conditions. In the case, that a certain action is not needed anymore, for example, due to changed requirements that affect a certain delta module, it is desirable to remove this action to clean up the code. To this end, we propose the refactorings of group three. During evolution, we may reach the point where particular delta modules are redundant or interchangeable. Hence, it is desirable to merge these modules in a specific way, which is addressed by the fourth group of Table 2. Another issue, specific to software product line, are the valid features configurations and, thus, the possible products that can be derived from an SPL. With the refactorings of the fifth group, we support the addition of possible products by refactoring the valid features configurations or application conditions of the delta modules.

1The different groups of refactorings are separated by horizontal lines in the respective tables.
<table>
<thead>
<tr>
<th>Name of Refactoring</th>
<th>Typical Situation</th>
<th>Recommended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resolve Duplicated Actions</td>
<td>Two delta actions execute the same modification, e.g., add equivalent methods or remove the same class.</td>
<td>Extract the delta actions into a single new delta module, combining the application conditions using the boolean or-operator. If the delta action is a removal or modification action, resolve it by the refactorings Resolve Removal Action or Resolve Modification Action. If the delta action is an addition action, resolve all succeeding removal or modification actions. Finally, remove all dead delta modules by executing the refactoring Remove Dead Delta Module.</td>
</tr>
<tr>
<td>Remove Dead Delta Action</td>
<td>A delta action has no effect on any product, e.g., a field added by a delta action is always removed by another delta action.</td>
<td></td>
</tr>
<tr>
<td>Remove Dead Delta Module</td>
<td>A delta module is never used because its application condition is a contradiction, either by itself or with respect to the valid feature configurations.</td>
<td>Remove the dead delta module and all references to it.</td>
</tr>
<tr>
<td>Remove Empty Delta Module</td>
<td>All delta actions of a delta module have been extracted or moved into other delta modules.</td>
<td>Remove the empty delta module and all references to it.</td>
</tr>
<tr>
<td>Merge Compatible Partition Parts</td>
<td>Two partition parts contain only pairwise commuting delta modules.</td>
<td>Merge the two partition parts into a single partition part.</td>
</tr>
<tr>
<td>Remove Empty Feature</td>
<td>A feature is defined in the feature model, but has no effect on the application conditions of the delta modules.</td>
<td>Remove the feature from the feature list and replace all references to the feature in the application conditions and feature model constraint with the boolean constant false.</td>
</tr>
<tr>
<td>Remove Unused Feature</td>
<td>A feature is defined in the feature model, but not used by any product of the product line.</td>
<td>Remove the feature from the feature list and replace all references to the feature in the application conditions and feature model constraints with the boolean constant false.</td>
</tr>
<tr>
<td>Merge Duplicated Features</td>
<td>Two features are defined in the feature mode, but each feature has the same effect in the application conditions of the delta modules.</td>
<td>Merge both features into a single feature by replacing any reference to one of the original features with a reference to the new feature. Remove references to the features from products, referencing only one of the features. Thereafter, merge both features into a single feature by replacing any reference to one of the original features with a reference to the new feature.</td>
</tr>
<tr>
<td>Merge Joined Features</td>
<td>Two features are defined in the feature model, but each feature has only any effect on the application conditions of the delta modules if both of them are selected.</td>
<td></td>
</tr>
<tr>
<td>Simplify Application Conditions</td>
<td>After executing some refactorings, e.g., Resolve Removal Action or Remove Unused Feature, the application condition of a delta module got unnecessarily complex.</td>
<td>Replace the application condition of the delta module with an equivalent, but simplified version.</td>
</tr>
<tr>
<td>Simplify Feature Configurations</td>
<td>After refactoring the feature model, the propositional formula of the valid feature configurations became unnecessarily complex.</td>
<td>Replace the propositional formula with an equivalent, but simplified version.</td>
</tr>
</tbody>
</table>

Duplicated or dead code may occur in product lines as well. With the first two groups of refactorings in Table 3, we address this problem by proposing refactorings that remove dead or duplicated delta actions as well as complete delta modules. Similarly to code artifacts, a feature model of an SPL also evolves. This can yield that the dependencies amongst features are hard to understand. With the refactorings of the third group of Table 3, we provide means to restructure feature models such as removing unused or duplicated features. Such refactorings do not affect the number of possible products, because all existing products can still be generated. Finally, we propose two refactorings that deal with simplifying application conditions and feature configuration constraints in case that they become too complex, for instance, as a consequence of other refactorings.

### 4.2 Selected Refactorings

In the following, we describe the mechanisms for two selected refactorings used again in the next subsection. For a comprehensive overview and description of all refactorings, we refer to [20].

#### Simplify Application Conditions

Many automated refactorings modify the application conditions of delta modules mechanically, which can produce very complex propositional formulas. Because of that, the
Simplify Application Conditions refactoring can be executed as a final clean-up after the execution of other refactorings.

Strategy. This refactoring can be triggered automatically by other refactorings, or the developer can trigger it manually for a single application condition or the complete software product line.

Input. Either an application condition must be selected, or if all application conditions of a product line shall be simplified the software product line must be selected.

Preconditions. No specific preconditions exist for this refactoring. The DeltaJ program must be well-formed, though.

Mechanics
- For each application condition, the propositional formula following the when-statement is analyzed, and a simplified version is computed, using a suitable minimization algorithm.
- The original propositional formula of the when-statement is replaced with the new simplified version.

Resolve Removal Action
Every removal action can be eliminated by modifying the application conditions of preceding delta actions, affecting the same target entity. This is true for all class, method, or field removal actions.

Execution Strategy. The developer can manually trigger this refactoring for each class, method or field removal action. Alternatively, the transformation can be automatically triggered for each class, method or field removal action, which are contained in delta modules, or even the complete software product line.

Input. The developer has to select a removal action, a delta module or the software product line.

Preconditions. No specific preconditions exist for this refactoring. The DeltaJ program must be well-formed, though.

Mechanics. Suppose that a delta action contained in the delta module DRemove shall be resolved.
- First, all preceding delta actions (i.e., actions executed before the removal action) that affect the same target (class, method or field), have to be extracted. This can be done by using the refactoring Extract Delta Action, internally. Here, preceding delta actions refers to those delta actions, which are contained in delta modules preceding DRemove with respect to the delta module ordering.
- The extracted delta modules with the preceding delta actions can now be disabled for those feature configurations, in which the application conditions of the delta module DRemove evaluate to true. That is, given the declaration DRemove when R, the module references of all preceding extracted delta actions need to be converted from DBefore when C to DBefore when C && R.
- Thereafter, the removal action can safely be deleted from the delta module DRemove.
- As a final clean-up step, the refactoring Merge Delta Modules with Equivalent Conditions, Remove Empty Delta Module and Simplify Application Conditions can be executed, if required.

4.3 Delta-Oriented Refactorings in Action
We now illustrate refactorings of delta-oriented SPLs in more detail to prove their applicability and usefulness using the ATM product line, introduced in Section 2. Our starting point is the incompatibility of the features Cash and Balance mentioned in Section 3. Our goal is to make them compatible so that both features can be selected for certain product variants, according to the feature diagram in Figure 5. To achieve this goal, several refactoring steps are necessary, which we explain in the following. We explicitly point out that the actual application of the refactorings is performed (semi-)automatically, while determining the code, subject to refactoring, is primarily a manual task.

Preparing the Evolution
Within the initial implementation of the delta module DBalance (and thus of the feature Balance), different remove and modifies actions disable some functionality of delta module DCash in case the feature Balance is selected (cf. Figure 3). Consequently, the first step to make both features compatible is to undo these actions. To this end, we apply the refactorings Resolve Removal Action and Resolve Modification Action to the delta module DBalance.

In Figure 6, we show the implementation of the ATM SPL after applying these refactorings to the implementation shown in Figure 3. The applied refactorings automatically extract the delta actions for the class CashDispenser, for the field Controller.cashDispenser and for the two methods startExecution() and executeWithdrawal() from the delta module DCash into separate delta modules. The application conditions of the (newly created) delta modules are defined such that the delta modules are not applied if the feature Balance is selected which is achieved by adding the negation of this feature to the existing application conditions. This makes the removal and modification actions in DBalance obsolete such that they are deleted.

Removing Redundancy
As a result of the previous refactoring, we obtain several micro delta modules with complex application conditions. Furthermore, different delta modules have identical application conditions, which can lead to inconsistent changes in future evolution steps. To reduce the complexity and to remove the redundancy, we can merge delta modules with the same application conditions by applying the refactoring Merge Delta Modules with Equivalent Conditions. As a result, all delta modules of Figure 6 that start with the prefix DCash_ are merged into the delta module DCashExtracted, which we show in Figure 7.

Improving Readability
To improve readability and to support developers with understanding the source code, the names of different program entities, such as the delta modules, should correctly reflect their content [15]. To this end, we apply the refactoring Rename Delta Module to rename the delta module DCash into DCashCommon and DCashExtracted into DCash. For the same purpose, we apply the refactoring Simplify...
delta DCash { 
  adds class Bank {...} 
  adds class Screen {...} 
  adds class Controller {
    Bank bank; 
    Screen screen; 
  } 
}

delta DBalance { 
  modifies Controller {
    adds void startExecution() {
      this.printBalance(); 
    } 
    adds void printBalance() {
      String balance = this.bank.getBalance();
      this.screen.printBalance(balance);
    } 
  } 
}

delta DCash_Controller_cashDispenser { 
  adds class CashDispenser {...} 
}

delta DCash_Controller_executeWithdrawal { 
  modifies Controller {
    void executeWithdrawal() {
      String amount = this.screen.askForAmount();
      this.bank.withdrawCash(amount);
      this.cashDispenser.dispense(amount);
    } 
  } 
}

delta DCash_Controller_startExecution { 
  modifies Controller {
    void startExecution() {
      this.executeWithdrawal();
    } 
  } 
}

delta DCashExtracted { 
  adds class CashDispenser {...} 
  modifies Controller {
    adds CashDispenser cashDispenser;
  } 
}

delta DBalance { 
  modifies Controller {
    adds void startExecution() {
      this.executeWithdrawal();
    } 
  } 
}

delta DBalance { 
  modifies Controller {
    adds void startExecution() {
      this.executeWithdrawal();
    } 
  } 
}

delta DBalance { 
  modifies Controller {
    adds void startExecution() {
      this.executeWithdrawal();
    } 
  } 
}

delta DBalance { 
  modifies Controller {
    adds void startExecution() {
      this.executeWithdrawal();
    } 
  } 
}

delta DBalance { 
  modifies Controller {
    adds void startExecution() {
      this.executeWithdrawal();
    } 
  } 
}

delta DBalance { 
  modifies Controller {
    adds void startExecution() {
      this.executeWithdrawal();
    } 
  } 
}

delta DBalance { 
  modifies Controller {
    adds void startExecution() {
      this.executeWithdrawal();
    } 
  } 
}

Figure 7: ATM SPL after resolving removal and modification actions

Method startExecution() in Line 13 and Line 25 of Figure 7 such that the resulting program is not well-formed. To eliminate this conflict, we apply the refactoring Extract Conflicting Actions, which extracts the two conflicting methods into separate delta modules DCash_startExecution and DBalance_startExecution, respectively (cf. Figure 9).

Adapting the Feature Model

In order to add the configuration where the features Balance and Cash are simultaneously selected, we have to adapt the feature model of the ATM SPL. We apply the refactoring Merge Configurations into Conditions to add the feature model constraint to the application conditions of the delta modules in order to maintain the applicability of the delta modules if the feature model is changed. This refactoring merges the propositional formula describing the feature model into all application conditions, using the logical AND operation which is illustrated in Figure 10.

Evolving the ATM SPL

The applied refactorings have prepared the existing implementation of the ATM SPL in a way that allows easily extending the product line to comprise also a product variant containing the Cash and the Balance feature. First,
we create a delta module \textit{DCashAndBalance} for the part that is common for both, \textit{DCash} and \textit{DBalance}, in method \texttt{startExecution}. Second, the feature model is changed to contain the combination of the \textit{Cash} and \textit{Balance} features. As a clean-up step, the refactoring \textit{Extract Configurations from Conditions} can be used to reduce again the verbosity of the application conditions. Listing 11 shows the evolved software product line, realizing the feature model in Figure 5.

With this example, we have shown refactorings that support tasks required regularly in real-world SPLs. We applied refactorings for both purposes, i.e., for supporting SPL evolution (e.g., in \textit{Preparing the Evolution}) as well as for increasing maintainability (and readability) of code (e.g., in \textit{Remove Redundancy}). However, even within our small example, it is difficult to decide \textit{when} to apply a certain refactoring. To support developers in making this decision, we briefly introduce code smells for delta-oriented SPLs in Appendix A.

\subsection*{4.4 Discussion}

In this section, we discuss issues that play a pivotal role for the applicability or the proposed refactorings.

\textbf{Scope of refactorings.}

Most of the presented refactorings are based on the well-known object-oriented refactorings proposed by Fowler et al.\cite{fowler2002refactoring}. We argue that this is reasonable because of the similarities in evolution and maintenance of product lines in relation to single object-oriented systems. For instance, renaming is a common task for different reasons, such as better reflection of purposes or avoidance of name conflicts. Moreover, we argue that extracting code fragments is a common task in SPLs, independent of the actual approach. For example, in AOP it may be necessary to extract code from an advice to split this advice to provide a more fine-grained composition of an SPL. Similarly, extracting methods or classes from features is particularly useful, if it is necessary to split this feature either for supporting more variants or for reducing the size of a (too dominant) feature. However, we also provide refactorings that are specific to SPLs such as those concerning features or application conditions of delta modules. Nevertheless, there may be refactorings which are currently not considered, but turn out to be important during evolution of product lines. To find such refactorings, analyzing the evolution of SPLs (and recurring patterns along this process) is a promising approach.

\section*{Scalability.}

To illustrate the applicability of our refactorings, we use the example of an ATM product line. While this is a small and intuitive SPL, real-world product lines exhibit more features and code and, thus, are more complex. This raises the question to what extent our refactorings are scalable to such SPLs. The answer is twofold. Generally, applying the refactorings is possible even for large-scale SPLs, because the refactorings are executed automatically. Nevertheless, there may be limitations in understanding what has been changed by a refactoring in such SPLs, because the changes...
Although the proposed refactorings are tailored to DOP, we argue that they are applicable for other compositional SPL implementation approaches, such as FOP or AOP, as well. To adopt our refactorings for these alternative approaches, we have to distinguish two kinds of refactoring targets: features and source code. For the former, our proposed refactorings can be used "as is", because features are independent of the underlying language. We only have to take the mapping between the feature model and the corresponding source code (feature modules or aspects) into account. For instance, in case of the Rename Feature, we have to ensure that the renaming is performed at all places (which includes possible mappings). Moreover, for FOP we can even apply the refactorings that target on delta modules by only replacing delta by feature modules, since DOP encompasses FOP [20]. While the effort for adapting the source code refactorings to the other approaches may be higher, the underlying concepts can be reused anyway. For instance, extracting delta actions corresponds to extracting parts of an aspect and changing the corresponding advice (or any expression that maps the aspect to a certain feature) in AOP. Another example is the merging of delta modules in case of equivalent application conditions. For AOP, we would have to search for aspects that contribute to the same feature or address identical join points. Hence, even on the source code level, our refactorings can be generalized.

5. IMPLEMENTATION

We provide tool support for applying the refactorings presented in this paper. The primary goal of this implementation is to prove the feasibility of the proposed refactorings and to evaluate their benefits in practice. The implementation of the delta-oriented refactorings is based on the prototype implementation of the DELTAJ language, presented in [22]. The prototype is implemented as a set of Eclipse plug-ins, using the Xtext framework with Eclipse 3.7.1 and Xtext 2.2.1. It can be obtained from our project page ².

The Xtext framework [26] is an open-source framework for the development of domain-specific languages within the Eclipse framework. The goal of Xtext is to make the development of programming language as easy and efficient as possible. This is achieved by generating big parts of the language infrastructure from the language grammar file, which is a domain-specific language itself.

**Figure 11: Evolved ATM SPL**

```java
delta DCommon {...}
delta DCash {...}
delta DBalance {...}
delta DCash_startExecution {...}
delta DBalance_startExecution {...}

delta DCashAndBalance {
  modifies Controller {
    adds void startExecution() {
      if(this.screen.askForTypeOfRequest())
        this.executeWithdrawal();
      else
        this.printBalance();
    }
  }
}
spl Atm {
  features Cash, Balance, English, German
  configurations (Cash || Balance) && (English && !German || German && !English)
  deltas
    [DCommon when Cash || Balance,
    DCash when Cash,
    DCash_startExecution when Cash && !Balance]
    [DBalance when Balance,
    DBalance_startExecution when Balance && !Cash]
    [DCashAndBalance when Cash && Balance]
    [DCommon when Cash, DCommon when German]
}
```

**Figure 12: Architecture of the implementation**

In Figure 12, we illustrate the architecture and the dependencies between the developed Eclipse plug-ins. The prototype implementation for the DELTAJ language is realized by the plug-ins DeltaJ Language and DeltaJ UI. The DeltaJ Language plug-in defines the grammar of the DELTAJ language and provides corresponding source code parsers and validators, while the DeltaJ UI plug-in extends the Eclipse UI with a source file editor and a project creation wizard. Based on the DELTAJ language plug-ins, the delta-oriented SPL refactorings are realized by the DeltaJ Transformations plug-in. The refactorings are implemented by altering the abstract syntax tree of the parsed DELTAJ program. New tree nodes can be created and added to the abstract syntax tree, while existing nodes can be moved within or removed from the abstract syntax tree. The Xtext framework automatically applies the transformations of the abstract syntax tree to the source code shown in the editor. For example, if a tree node of type DeltaModule is inserted into the abstract syntax tree, the corresponding code for the delta module is automatically inserted into the source code of the DELTAJ program.

To enable the user to trigger the implemented refactorings, the DeltaJ Transformations UI plug-in extends the Eclipse outline view with a dynamic context menu. In the outline

²https://svn.isf.cs.tu-bs.de/redmine/projects/deltaj_ext-refactoring/files
³http://xtext.itemis.com/
view, for each language entity, a context menu with a set of possible refactoring transformations is offered. For example, in Figure 13, a selected modification action can be resolved, removed or extracted into a separate delta module.

![Figure 13: Outline view of the DeltaJ user interface](image)

6. RELATED WORK

While Opdyke and Fowler initially proposed refactorings for object-oriented programming [17, 7], approaches for refactoring of software product lines have been proposed as well. In the following, we mention important work and compare it to ours. Liu et al. proposed feature-oriented refactoring for FOP [14]. However, they rather aim at decomposing existing programs into features and, thus, support the extractive product line development approach [10]. Beside this, they rather propose formal foundations for this process than practical refactorings or even tool support. In contrast, we aim at improving the structure of existing delta-oriented SPLs and propose concrete refactorings.

Similarly, Alves et al. propose refactorings for product lines with a special focus on extractive SPL development [1]. Additionally, they consider refactorings for feature models that aim at improving the model structure. In the same way, Thüm et al. provide a sound reasoning on refactoring feature models [27]. While we support the latter as well (e.g., by simplifying feature configurations), we additionally provide real source code transformations. Furthermore, we aim at restructuring instead of creating software product lines, which is also different from Alves et al. [1].

Kuhlemann et al. propose the concept of refactoring feature modules (RFM) for refactoring software product lines [11]. RFMs are specific feature modules that contain all code necessary for applying an object-oriented refactoring. However, while this approach can only apply refactoring for certain variants (those, that have an RFM in their configuration), our refactorings affect the whole SPL.

Recently, we proposed variant-preserving refactorings in a catalogue-like manner for FOP [24]. While this is similar to this paper, we only provided four refactorings, tailored to FOP, which is less comprehensive than the refactorings in this paper. Moreover, we provide no tool support nor an evaluation for the refactorings in FOP.

Finally, two other papers represent important work, because they are complementary to ours. First, Borba proposed a theory of product line refinement that enables stepwise SPL evolution [3]. To show the usability, he explicitly describes how this could work for exemplary artifacts of a product line. However, the focus of this work is rather theoretical and, thus, no implementation or case study exist. Nevertheless, it has the same focus as our work, that is, supporting evolution and refactorings of product lines. Second, Monteiro presented a catalogue of refactorings for aspect-oriented programming [16]. Also code smells specific to AOP are considered, and it is shown how existing code smells indicate crosscutting concerns. Although this work is very similar to ours regarding refactoring and code smells (except for the paradigm), two differences remain: First, we explicitly focus on software product lines and, thus, consider all entities of SPLs. In contrast, Monteiro solely supports aspect-oriented programs without taking variability into account. Second, we provide tool support, integrated in an IDE, for applying most of the proposed refactorings (semi-)automatically.

7. CONCLUSION

Software, and in particular SPLs, evolves over time in order to meet new or changing requirements, which increases its complexity and impedes maintainability. As a result, it is crucial for the success of SPLs to counter the implications of evolutionary changes to ensure the reusability and maintainability of the artifacts encompassed by the SPL. In this paper, we tackle this problem by proposing a catalog of refactorings for delta-oriented software product lines to address the preparation of evolution or the improvement of maintainability. Additionally, we proposed code smells for delta-oriented SPLs for identifying possible refactoring opportunities. Since refactoring SPLs is manually tedious or even infeasible, we provide an implementation to apply the refactorings semi-automatically.

Beyond that, we discuss several aspects of the proposed refactorings that are crucial for a broader acceptance. Amongst others, we provided arguments why and how our refactorings are generalizable regarding other compositional approaches and how they make up with large-scale SPLs.

8. REFERENCES

APPENDIX

A. CODE SMELLS

In the following, we shortly explain code smells in delta-oriented SPLs and their countermeasures in terms of refactorings. A more comprehensive overview of these code smells can be found in [20].

**Duplicated Delta Action.** Two delta actions with identical content exist in the SPL. Recommended refactoring: *Resolve Duplicated Actions.*

**Dead Delta Action.** The effect of a delta action is always overwritten by another delta action. Recommended refactoring: *Remove Dead Delta Action*

**Dead Delta Module.** The application condition of a delta module is a contradiction, either by itself or with respect to the valid feature configurations. Recommended refactoring: *Remove Delta Module*

**Empty Delta Module.** A delta module contains no delta actions. Recommended refactoring: *Remove Empty Delta Module*

**Unused Feature.** A feature is defined, but not used by any product. Recommended refactoring: *Resolve Unused Feature*

**Duplicated Features.** Two distinct features have the same effect. Recommended refactoring: *Merge Duplicated Features*

**Joined Features.** Two distinct features have only any effect if both are selected. Recommended refactoring: *Merge Joined Features*

**Complex Feature Configurations.** The propositional formula, specifying the valid feature configurations, is more complex than necessary. Recommended refactoring: *Simplify Feature Configurations*

**Complex Application Conditions.** An application condition is more complex than necessary. Recommended refactoring: *Simplify Application Conditions*