Investigating the Safe Evolution of Software Product Lines

Laís Neves, Leopoldo Teixeira, Paulo Borba
Informatics Center
Federal University of Pernambuco
50740-540, Recife – PE – Brazil
{lmn3, lmt, phmb}@cin.ufpe.br

Demóstenes Sena
Federal Institute of Education, Science and Technology of Rio Grande do Norte
59015-000, Natal – RN – Brazil
demostenes.sena@ifrn.edu.br

Vander Alves
Computer Science Department
University of Brasília
70910-900, Brasília – DF – Brazil
valves@unb.br

Uirá Kulesza
Computing Department
Federal University of Rio Grande do Norte
59072-970, Natal – RN – Brazil
uira@dimap.ufrn.br

Abstract

The adoption of a product line strategy can bring significant productivity and time to market improvements. On the other hand, evolving a product line is risky because it might impact many products and their users. So when evolving a product line to introduce new features or to improve its design, it is important to make sure that the behavior of existing products is not affected. In fact, to preserve the behavior of existing products one usually has to analyze different artifacts, like feature models, configuration knowledge and the product line core assets. To better understand this process, in this paper we discover and analyze concrete product line evolution scenarios and, based on the results of this study, we describe a number of safe evolution templates that developers can use when working with product lines. For each template, we show examples of their use in existing product lines. We evaluate the templates by also analyzing the evolution history of two different product lines and demonstrating that they can express the corresponding modifications and then help to avoid the mistakes that we identified during our analysis.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Software Product Lines

General Terms Investigation, analysis

Keywords Software product line, refactoring, product line evolution

1. Introduction

A software product line (PL) is a set of related software products that are generated from reusable assets. Products are related in the sense that they share common functionality. This kind of reuse targeted at a specific set of products can bring significant productivity and time to market improvements [21, 27]. To obtain these benefits with reduced upfront investment and risks, previous work [3, 8, 18] proposes to minimize the initial product line (domain) analysis and development process by deriving a product line from an existing product. A similar process applies to evolving a product line, when adding new products or improving the PL design requires extracting variations from previous parts shared by a set of products.

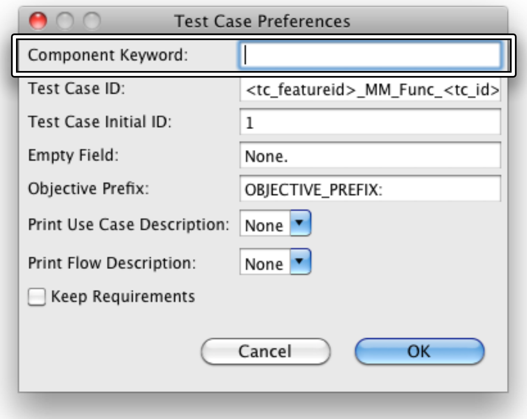
Manually extracting and changing different code parts when evolving a PL requires substantial effort, especially for checking necessary conditions to make sure the extraction is correctly performed. Moreover, this process is tedious and can also easily introduce defects, modifying the behavior of the products before the extraction process, and compromising the promised benefits on other dimensions of costs and risks.

To better understand this process, in this paper we discover and analyze concrete product line evolution scenarios and, based on the results of this study, we describe a number of safe evolution templates that developers can use when working with product lines. For this, we rely on a notion of PL refinement that preserves the PL original products behavior while allowing the generation of new products in the resulting PL [6]. We use the definitions and suggestions from previous works [5], [6] to identify evolution scenarios and then generalize these scenarios to other situations through the templates.

These templates specify transformations that go beyond program refactoring notions [14, 22] by considering both sets of reusable assets that not necessarily correspond to valid programs, and extra artifacts, such as feature models (FM) [11, 15] and configuration knowledge (CK) [11], which are necessary for automatically generating products from assets. For each template, we show examples of their use in existing product lines. We evaluate the templates by also analyzing the evolution of two different product lines and demonstrating that they can express the corresponding modifications and then help to avoid the mistakes that we identified during our analysis.

This paper makes the following contributions:

- we discover and analyze PL evolution scenarios by mining part of a PL SVN history (Section 2);


Test Case Preferences

Component Keyword:	
Test Case ID:	<tc_featureid>_MM_Func_<tc_id>
Test Case Initial ID:	1
Empty Field:	None.
Objective Prefix:	OBJECTIVE_PREFIX:
Print Use Case Description:	None
Print Flow Description:	None

☐ Keep Requirements

Figure 1. Scenario A - *Test Case Preferences* widow with the new *Component Keyword* field

- – we identify and describe precisely a number of product line safe evolution templates that abstract, generalize and factorize the analyzed scenarios (Section 4);
- – we show evidence that the identified templates can justify all evolution scenarios in the SVN history of two PLs and could avoid the mistakes that we found during our analysis. We also show the frequency of use for each template in the analyzed scenarios (Section 5);

Besides these sections, Section 6 lists the works related to our research and Section 7 presents the concluding remarks.

2. Motivating Example

In order to better illustrate the problems that might occur when manually evolving PLs, we present a maintenance scenario based on TaRGeT PL [12]. TaRGeT is a product line of automatic test generation tools and is implemented using Eclipse RCP plug-ins technology. TaRGeT has been developed since 2007 and its current version has 42 implemented features and counts approximately 32,000 lines of code. The history track of 3 major releases and several minor releases is available in a SVN repository.

While analyzing TaRGeT's SVN history we have found several evolution steps that were supposed to be safe, that is just a design improvement or the addition of new products, but actually introduced errors to the product line. **Scenario A** describes one of these cases, the implementation of the new *Component Keyword* text field in the *Test Case Preferences* window. Figure 1 shows the new field. This field should only appear when the *XLS STD* feature is selected. This feature is related to TaRGeT's output format and, when selected, generates test suites in a format compatible with Microsoft Excel.

With that in mind, Figure 2 describes changes applied to the feature model, configuration knowledge and source code artifacts in order to address this evolution scenario. The CK notion that we here is represented as a table and maps feature expressions (in the left-hand side column) to asset names (in the right-hand side column). In this case, we only show the parts of configuration knowledge and feature model that are related to the example context. As TaRGeT has many features implemented, due to limited space it is

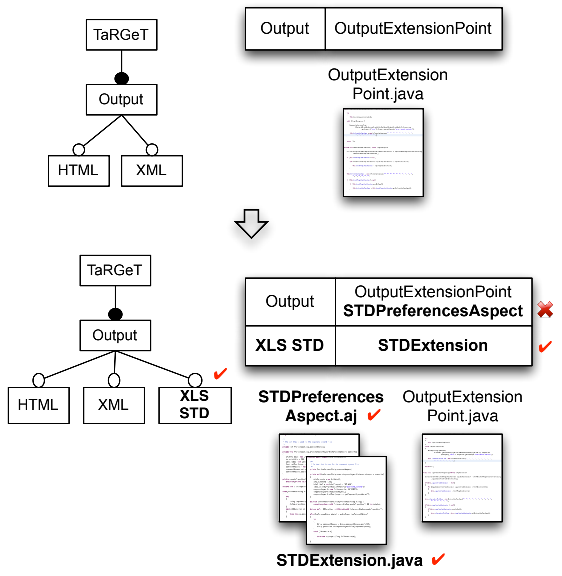

TaRGeT
Output
HTML
XML
Output | OutputExtensionPoint
OutputExtension Point.java
TaRGeT
Output
HTML
XML
XLS STD ✔
Output | OutputExtensionPoint STDPreferencesAspect ✖
XLS STD | STDExtension ✔
STDPreferences Aspect.aj ✔
OutputExtension Point.java
STDExtension.java ✔

Figure 2. Scenario A - Adding the new optional feature *STD Output*

not possible to show everything here. The tick signs indicate what was changed.

In summary, To modify the product line, the developers first created the aspect *STDPreferencesAspect*, which is responsible for introducing the *Component Keyword* field. However, when updating the configuration knowledge, they made a mistake and the aspect was associated to the *Output* feature instead of *XLS STD*. The developers then tested the product with the *XLS STD* feature selected and saw that the new field was present in the window as expected. They also tested the products with other output formats but they only verified if the main features were correct. They haven't noticed that the new *Component Keyword* field became visible in all product configurations, introducing a bug in the product line. As a consequence, they thought that the products were working as expected and committed the modified code to make the changes effective.

This example demonstrates that manually evolving a PL is error-prone, because in order to make sure that the behavior of existing products is not affected, one usually has to analyze different artifacts, like feature models, configuration knowledge and the product line assets (such as classes, configuration files or aspects). In addition, the bugs that might be introduced during manual evolution of PL could be difficult to track because they are present only in certain product configurations.

Analyzing TaRGeT's evolution history through releases 4.0 to 6.0 (from January to July 2009), we identified a total of 20 evolution scenarios. We verified that the minimum number of modified classes in these scenarios was 1 and the maximum was 54, and the average number of modified classes was 12.9. We also found that about 20% of these modifications introduced a defect in the product line. This shows that issues like the one presented might happen often and demand special attention.

To better understand the problems that might occur with manual PL evolution, in following sections we discover and analyze concrete product line evolution scenarios and, based on the results of this study, we describe a number of safe evolution templates that developers can use when working with product lines.

3. Product Line Refinement and Safe Evolution

To guide our PL evolution analysis and help us to identify the evolution scenarios, we rely on a notion of PL refinement [2, 5] that is based on a notion of program refinement [6], which is useful for comparing assets with respect to behavior preservation. In this section we briefly introduce these necessary concepts to understand the PL safe evolution templates described in the next section.

Similar to program refinement, PL refinements are behavior-preserving transformations that go beyond source code, and might transform FMs and CKs as well. The notion of behavior is lifted from programs to product lines. In a product line refinement, the resulting PL should be able to generate products (programs) that behaviorally match the original PL products. So users of an original product do not observe behavior differences when using the corresponding product of the new PL. This is exactly what guarantees safety when improving the design of a PL or extending the PL to generate new products

In most of PL refinement scenarios, however, many changes need to be applied to the code assets, feature models and configuration knowledge, which sometimes leads the refactored PL to generate more products than before. As long as it generates enough products to match the original PL we still satisfy existing users. This is illustrated by Figure 3, where we refine the simplified MobileMedia product line (detailed in Section 5) by adding the optional Copy feature. The new product line generates twice as many products as the original one, but what matters is that half of them – the ones that do not have feature Copy – behave exactly as the original products. This ensures that the transformation is safe; we extended the product line without impacting existing users.

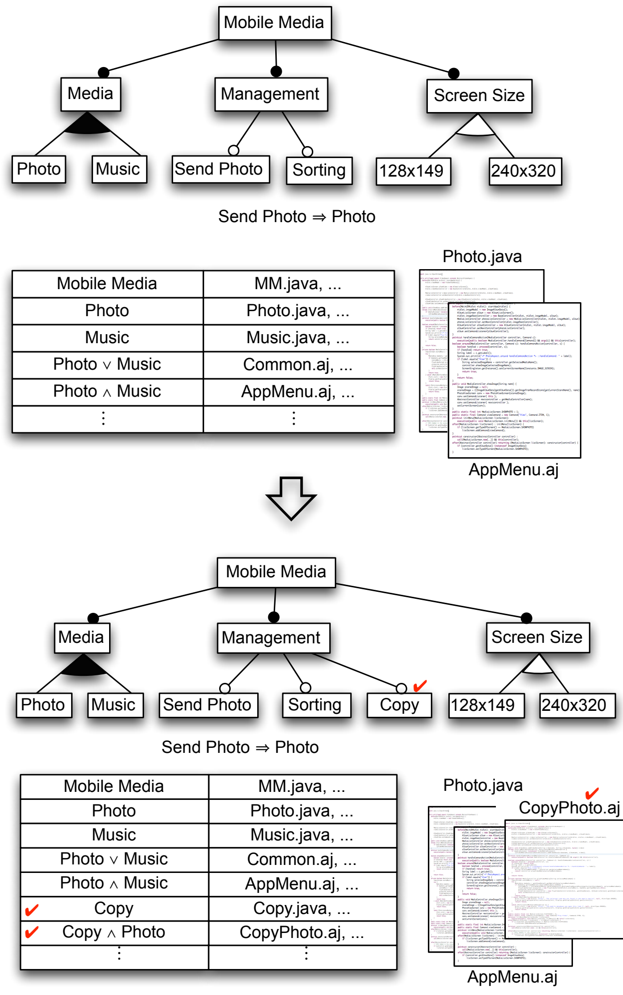
The PL refinement notion that we rely on formalize these ideas and is defined in terms of program refinement [23] [9]. Basically, each program generated by the original PL must be refined by some program of the new PL. Figure 4 illustrates this by showing two products lines, *PL* and *PL'*. In this example, *PL* is refined by *PL'* because for each product in *PL* (represented by a star shape), there is a corresponding product in *PL'* that refines it (represented by a square shape) and meaning that every behavior presented by the product on the right is a possible behavior of the corresponding product on the left. As we explained before, one can also have new products in *PL'* and still preserve the refinement relation.

4. Safe Evolution Templates for PLs

Based on this notion of refinement, in this section, we describe how we discovered and analyzed concrete product line evolution scenarios and, from the results of this study, we present a number of safe evolution templates. The templates provide guidance on how to structure extracted variant parts and help to avoid problems that might occur when evolving PLs manually.

The templates are valid modifications that can be applied to a product line thereby improving its quality and preserving existing products' behavior. The PL transformations listed here involve artifacts like feature models, configuration knowledge specifications and core assets as well. It is important to mention that the refinement notion that we rely on (see Section 3) is independent from the used language for feature model and configuration knowledge [6]. However, the safe evolution templates presented here are specific to the language of these artifacts.

To discover the safe evolution templates, we identified and analyzed different evolution scenarios from the TaRGeT PL between releases 4.0 to 5.0. During this time, we identified a total of 11 safe evolution scenarios, that means evolution steps according to the refinement notion that we rely on. After this step, we analyzed the changes performed in code assets, feature model and configuration knowledge. We also considered SVN commit comments and revi-



Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
⋮	⋮

Mobile Media	MM.java, ...
Photo	Photo.java, ...
Music	Music.java, ...
Photo ∨ Music	Common.aj, ...
Photo ∧ Music	AppMenu.aj, ...
✔ Copy	Copy.java, ...
✔ Copy ∧ Photo	CopyPhoto.aj, ...
⋮	⋮

Figure 3. Adding an optional feature refinement

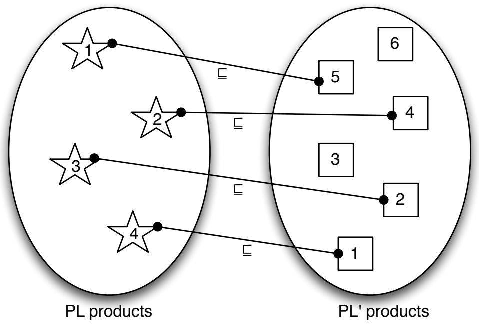
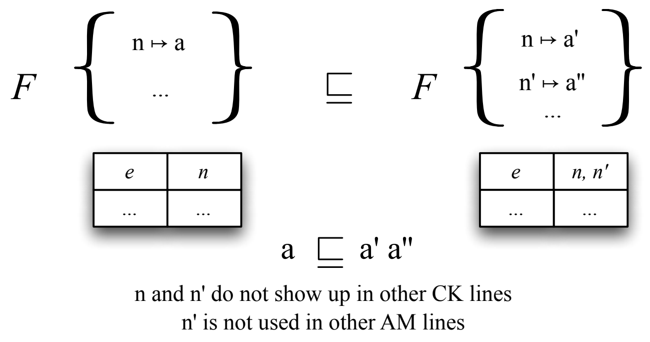


Figure 4. Product Line Refinement

sion history annotations in the source code files. Based on these results, we derived a set of safe evolution templates that abstract, generalize and factorize the analyzed scenarios and can be used in different contexts.

Regarding the used notation, each template described here shows the feature model, asset mapping and configuration knowledge status before and after the transformation. Feature models



F
n ↦ a
...
⊑
F
n ↦ a'
n' ↦ a''
...
e
n
...
...
e
n, n'
...
...
a ⊑ a' a''
n and n' do not show up in other CK lines
n' is not used in other AM lines

Figure 5. Template 1 - Split asset

contain only the features that are necessary to understand the templates. These are the features that are involved or are affected by the template being described. If we want to say that a feature may contain other features related to it, this can be expressed by a trace above or below the feature.

We represent the configuration knowledge by a two-column table, in which the left-hand column contains feature expressions that are mapped to names of assets, represented in the right-hand column. The ellipsis indicate other lines different from the one that is explicitly expressed. An asset mapping (AM) maps asset names into assets. It is useful to avoid conflicting assets names in the CK. Two curly braces represent the AM, grouping a mapping of asset names, on the left-hand side, to assets, on the right-hand side.

In the case that feature model, asset mapping or configuration knowledge are not changed in the transformation, they are expressed in the template by the single letters *F*, *A* and *K*, respectively. Each template also declares meta-variables that abstractly represent the PL elements, for example an arbitrary feature expression or an arbitrary asset name. The letters *F*, *A* and *K* are also meta-variables. If these variables appear in both sides of a transformation, this means that they remain unchanged. On the bottom we express the pre-conditions to the transformation.

4.1 Template 1 – Split Asset

When analyzing evolution scenarios that changed a mandatory feature to optional, we observe that this type of operation usually involved tracking the code related to the feature and extracting it to other artifacts, like aspects, property files or Eclipse RCP plug-in extensions (in case of TaRGeT).

To generalize these cases, we derived Template 1, illustrated by Figure 5. This template indicates that it is possible to split an asset *a* into two other assets *a'* and *a"* as long as the composition of assets *a'* and *a"* refines asset *a*. The first restriction is necessary to guarantee that the behavior of the old asset *a* is preserved. We added the second restriction to simplify the CK representation.

When *n* appears in other CK lines, it is only necessary to change all occurrences for *n*, *n'*. Another variation of this template is that *n'* can also represent an existing asset in the PL. We could discuss other variations too, but the focus here is on the basic template. The variations can often be derived from the basic template by composing it with other templates. We can say that Template 1 is a PL refinement because for each product that contained asset *a* before, there is now a corresponding refined one that contains the composition of assets *a'* and *a"* after the transformation. This specific transformation improves product line quality because it is possible to modularize feature behavior in different assets.

In our study, this template was used, for example, in the scenario that appears in Figure 6, where developers want to extract the

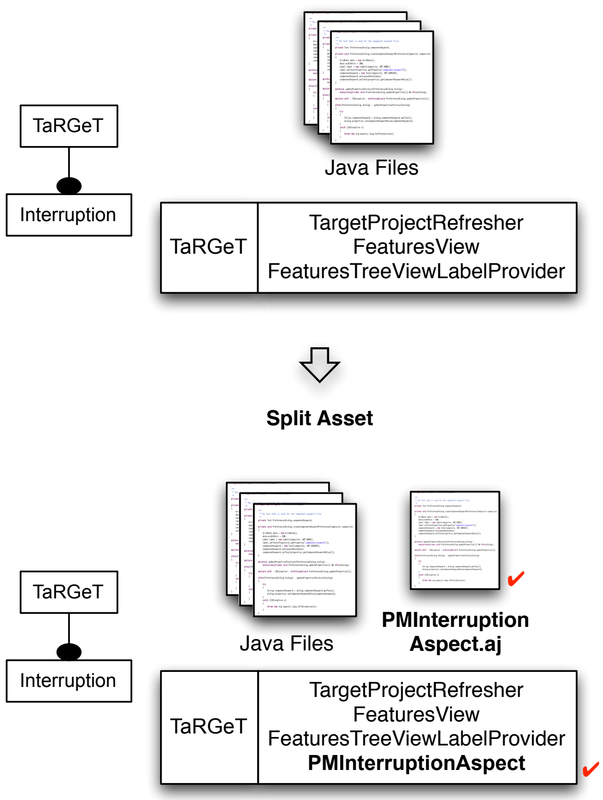


Figure 6. Split asset example

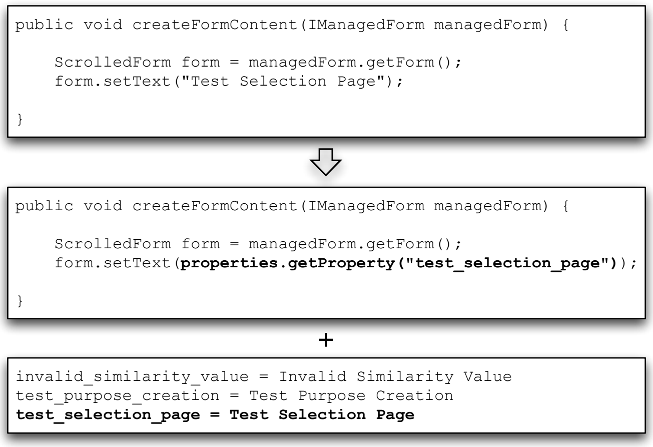
Interruption feature code that is scattered along three different Java classes (the ones listed in the CK) to a new Aspect. For this, they applied the *Split Asset* template and its variations several times and extracted the feature code to the new aspect *PMInterruptionAspect*. As can be noticed, the FM is not affected by the transformation. The source files in the figure represents the AM.

4.1.1 Split Asset – Code Transformations

In order to better explain the code transformations, like in the example presented on Figure 6, we need transformations that deal specifically with code assets. This is necessary because the general abstract PL templates only establishes the refinement constraint. So that is why we specify more precisely code transformations templates that complement the general templates for PL safe evolution. The existing refactorings in Eclipse IDE are an example of these code transformations to refactor Java code assets.

For Template 1, there are many variation extraction mechanisms described in other works [1, 3] that could be classified as code transformation templates. However, we only mention here the ones that we observed in our analysis of TaRGeT PL. All these code templates are helpful because they propose valid transformations that do not deteriorate the product line. So if a developer needs to maintain the product line, with the templates he reduces the possibility of introducing errors, increasing confidence.

In a practical scenario, like the one that appears in Figure 6, the developer first selects a PL template that makes the necessary transformation in the configuration knowledge and feature model, in this case the Split Asset template, and then selects the code transformation templates to actually implement the necessary changes in code assets. In the above mentioned example, we applied templates



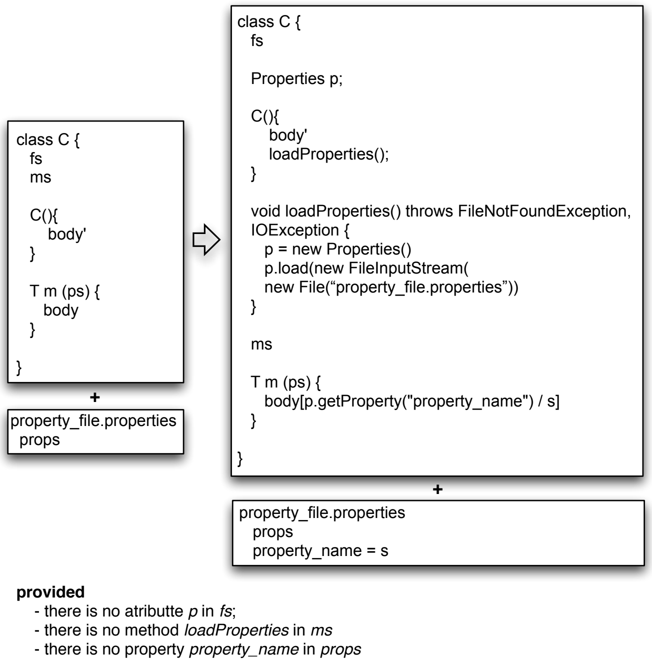
```
public void createFormContent(IManagedForm managedForm) {

    ScrolledForm form = managedForm.getForm();
    form.setText("Test Selection Page");

}
```

```
public void createFormContent(IManagedForm managedForm) {

    ScrolledForm form = managedForm.getForm();
    form.setText(properties.getProperty("test_selection_page"));

}
```

+

```
invalid_similarity_value = Invalid Similarity Value
test_purpose_creation = Test Purpose Creation
test_selection_page = Test Selection Page
```

Figure 7. Example of text extraction to properties file



```
class C {
  fs
  ms

  C(){
    body'
  }

  T m (ps) {
    body
  }

}
```

+

```
property_file.properties
  props
```

```
class C {
  fs

  Properties p;

  C(){
    body'
    loadProperties();
  }

  void loadProperties() throws FileNotFoundException,
  IOException {
    p = new Properties()
    p.load(new FileInputStream(
    new File("property_file.properties"))
  }

  ms

  T m (ps) {
    body[p.getProperty("property_name") / s]
  }

}
```

+

```
property_file.properties
  props
  property_name = s
```

provided
- there is no atributte *p* in *fs*;
- there is no method *loadProperties* in *ms*
- there is no property *property_name* in *props*

Figure 8. Abstract template for text extraction to properties file

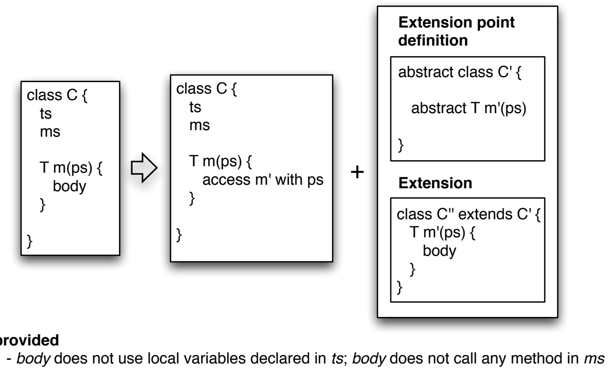
Extract Resource to Aspect - after, *Extract Method to Aspect*, *Extract Context* and *Extract After Block* to extract the feature code to the *PMInterruptionAspect*. These code templates first appeared in a catalog of refactoring templates to extract code from classes to aspects, using AspectJ [3]. The templates from this catalog rely on aspect oriented programming to modularize crosscutting concerns, which often occur in PLs.

Another code template that we identified during our analysis is useful to extract constants in the source code, usually user interface texts, to a properties file. We could define many other code templates to other types of values. This operation is commonly performed when there is the need to localize the user interface to support different languages, like in the example shown in Figure 7, where the form title text *"Test Selection Page"* is moved to a properties file. In this code template the original class is refined by the composition of the new refactored class with a call to the properties file, and the properties file itself.

Figure 8 shows the abstract transformation template. The notation used follows the representation of programming laws [23]. On the right-hand side, all occurrences of text *s* in *body* are replaced by a call to the property that contains its value. We denote the set of field declarations, method declarations and properties declarations by *fs*, *ms* and *ps*, respectively. We use *T* to represent the return type of method *m*. In class *C* constructor we place a call to a new method responsible to load the properties file. On the bottom we list the transformation restrictions.

In our study, we also identified code transformation scenarios that involved variation extraction to extension points. To capture that, we have a code template that represents this operation. This template uses Eclipse plug-ins extension point pattern and defines that it is possible to extract code within a class, create an extension point and replace the code in the class by a call to existing extensions that implement that new extension point. A new extension is then created with the extracted code. As TaRGeT is an Eclipse RCP application, we observed that many variations are implemented with the extension point mechanism and can be justified by this template.

Figure 9 presents the abstract transformation. On the left-hand side, is the *C* class that has at least a method *m* with a *body*. On the right-hand side, method *m* is refactored and *body* is replaced by a call to the *m'* method defined in the abstract class *C'*, which actually represents the extension point itself. We need other configuration files to implement extension points and corresponding extensions, but we have omitted them for clarity. The extension basically contains a class *C"* that extends the abstract class *C'* and implements method *m'*, where *body* is placed with the necessary modifications.



```
class C {
  ts
  ms

  T m(ps) {
    body
  }

}
```

```
class C {
  ts
  ms

  T m(ps) {
    access m' with ps
  }

}
```

+

Extension point definition

```
abstract class C' {

  abstract T m'(ps)

}
```

Extension

```
class C" extends C' {
  T m'(ps) {
    body
  }
}
```

provided
- *body* does not use local variables declared in *ts*; *body* does not call any method in *ms*

Figure 9. Abstract template for extension points

4.2 Template 2 – Refactor Asset

Another template that we propose based on the observations of our study is showed in Figure 10. This template defines that it is possible to modify an asset *a*, transforming it into asset *a'*, as long as the new asset *a'* refines the original asset *a*. We assure refinement because each product that contained asset *a* now has a corresponding refined one that contains asset *a'*.

Template 2 also relies on code transformation templates. For example, we could use Template 2 combined with existing refactorings for object-oriented, aspect-oriented, and conditional compilation programs. In practice, we know that some of these code transformations might change other code assets. For instance, if a class that is used by other classes is renamed, these classes need to be modified as well. We can have variations of Template 2 to deal

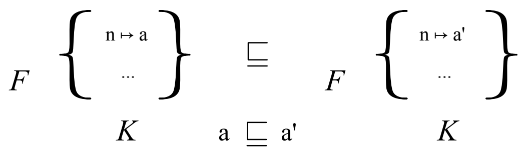


Figure 10. Template 2 - Refactor asset

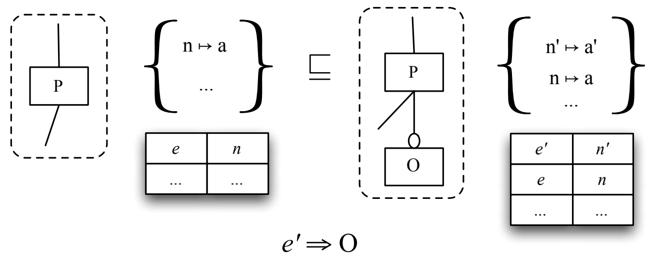


Figure 11. Template 3 - Add new optional feature

with these cases. They need to assure that the resulting product line is well formed and that the products that use these classes are not affected.

4.3 Template 3 – Add New Optional Feature

Template 3 emerged when analyzing evolution scenarios like the one described in our motivating example, when one introduces an optional feature to the PL. This template, presented in Figure 11, states that it is possible to introduce a new optional feature *O* and add a new asset *a* associated to a feature expression *e'* in the configuration knowledge only if the restriction that says that selecting *e'* implies selecting *O* is respected. The restriction assures that the new assets are only present in products that have feature *O* selected and that products built without the new feature correspond exactly to the original PL products. We assure refinement because the resulting product line has the same products that it had before in addition to products that contain feature *O*, and we improve the resulting PL quality by increasing its configurability.

In practice, the template implementation should be flexible enough to allow the association of more than one asset to the new optional feature in the CK. Figure 12 shows the application of Template 3 in the scenario that we described in our motivating example. The new assets *STDExtension* and *STDPreferencesAspect* are correctly associated to the new *XLS STD* feature in the CK.

4.4 Template 4 – Add New Mandatory Feature

Template 4, represented in Figure 13, indicates that we can insert a new mandatory feature, represented by *M*, on a feature model as long as we preserve the configuration knowledge, represented by *K*, and the asset mapping, represented by *A*. This transformation is a refinement because the asset mapping and the configuration knowledge do not change, so the products before and after the transformation are still precisely the same. It increases quality because it improves feature model readability.

If we had added new assets and associated them with feature *M* in the configuration knowledge, it would not be possible only with this template to ensure that these artifacts would not alter the behavior of existing products. Consequently, we could not assure that the transformation was a safe evolution step. It is possible to generalize even more the template considering that we can add any

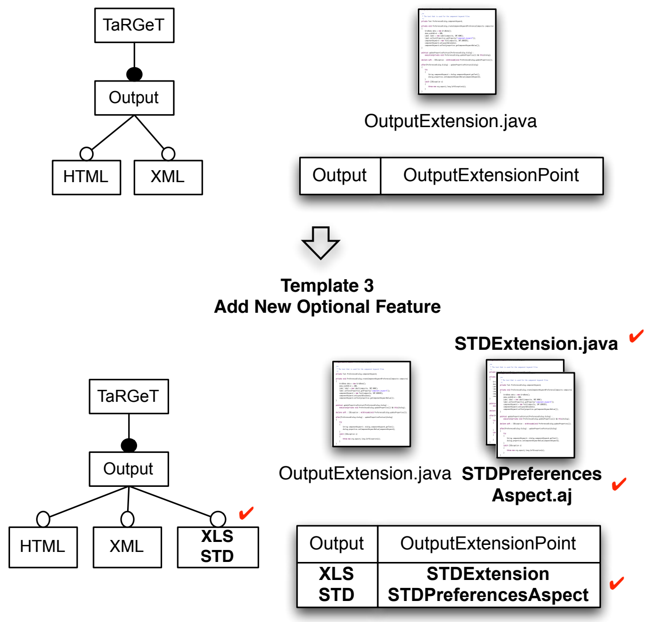


Figure 12. Add new optional feature example

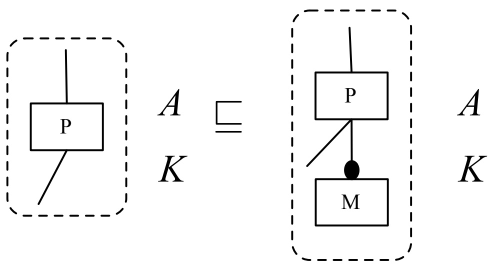


Figure 13. Template 4 - Add new mandatory feature

kind of feature to the FM (mandatory, optional, alternative, or) if we preserve code assets and CK.

Figure 14 illustrates the template utilization. In this example, developers inserted a new mandatory feature *Word* under the *Input* feature. This feature identifies the possible formats of use case documents that TaRGeT accepts as input. The *Word* feature represents the Microsoft Word format. This transformation is useful to improve FM readability. Similarly to this operation, it is possible to add any kind of feature (optional, alternative, or) in the FM, as long as the CK is preserved.

We observed in our study that Template 4 is usually used together with other templates following it. We decided to divide this transformation into two steps to facilitate the automation and reuse of the templates, since it is possible to combine templates to derive more complex transformations.

4.5 Template 5 – Replace Feature Expression

Template 5 in Figure 15 expresses that it is possible to change the feature expression associated to an asset *n* in the configuration knowledge from *e* to *e'* only if one respects the restriction that these expressions are equivalent considering the feature model. The restrictions specify that all product configurations from a feature

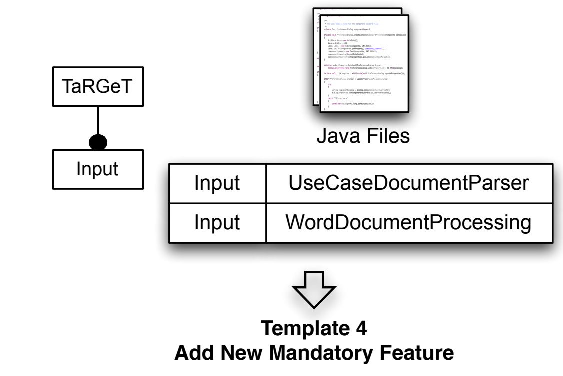

TaRGeT
Input
Java Files
Input | UseCaseDocumentParser
Input | WordDocumentProcessing
Template 4
Add New Mandatory Feature
TaRGeT
Input
Word
Java Files
Input | UseCaseDocumentParser
Input | WordDocumentProcessing

Figure 14. Add new mandatory feature example

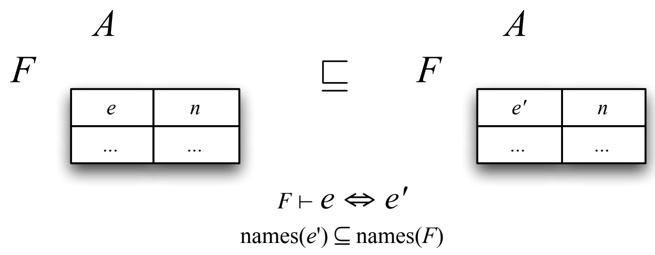


Figure 15. Template 5 - Replace feature expression

model *F* lead to equivalent evaluation for the feature expressions in both *e* and *e'*. It also specify that the feature expression in *e'* only references names from *F*. This template improves product line quality by enhancing configuration knowledge readability.

We found many occurrences of this template combined with Template 4. Figure 16 illustrates how we can use Template 5 in the example described in Figure14. In this example, the template changed the feature expression related to the *WordDocumentProcessing* asset from *Input* to *Word*. This is possible because as *Word* is under *Input*, selecting the first means that the second is also selected. This operation is useful because it improves both CK and FM readability and, in the example, structures the PL for the introduction of new *Input* formats.

4.6 Other Templates

When deriving the templates we assured first that they complied with the refinement notion that we rely on. After deriving the preliminary versions, we realized that in some cases the restrictions associated to the templates were too strong and that they could not be used in other situations different from the ones we analyzed. So

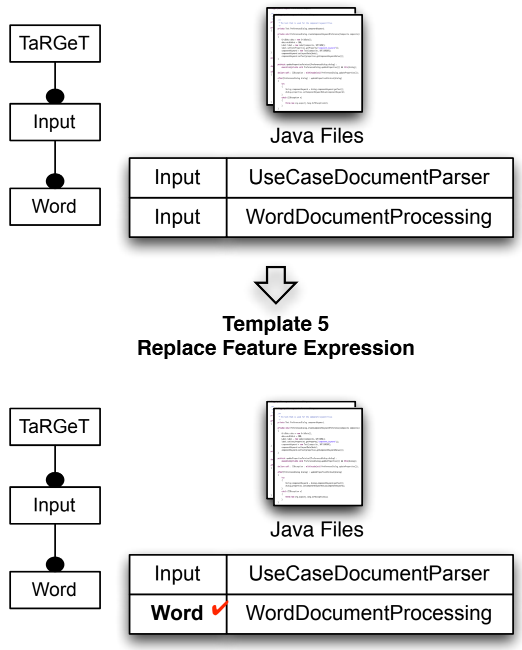

TaRGeT
Input
Word
Java Files
Input | UseCaseDocumentParser
Input | WordDocumentProcessing
Template 5
Replace Feature Expression
TaRGeT
Input
Word
Java Files
Input | UseCaseDocumentParser
Word | WordDocumentProcessing

Figure 16. Template 5 - Replace feature expression example

we decided to discard unnecessary conditions in order to make the templates more general and consequently more useful.

Besides, we found that we could divide some templates into more steps, which improved understanding and would help an automation process in the future in order to provide better support to developers. So we refactored and evolved these templates and new ones were derived.

We also observed that in some evolution scenarios that we analyzed, it was usually necessary to combine more than one template, for example, templates 4 and 5. This information can be useful when defining a strategy to compose the templates in an automated solution.

Table 1 summarizes all templates proposed in this work. Templates 1 to 5 are detailed in this section. Template 3 was first mentioned in a previous work [5]. Template 6 defines transformations that occur in configuration knowledge and asset mapping when adding a new alternative feature in the feature model. Similarly, Template 7 defines the same transformations to include an OR feature in the feature model. Finally, Template 8 defines asset removal and contains the program transformation law for class elimination [7] defined as a code transformation template. We derived this template when analyzing another PL, which we describe in more details in Section 5 The templates not detailed here can be found in our website [1].

5. Analyzing Product Line Safe Evolution

We chose two different product lines to perform our study. We first studied TaRGeT [12], which we previsously mention in Section 2. We also analyzed MobileMedia [13], a PL for media (photo, video and audio) management on mobile devices. This section describes how we investigated the evolution scenarios and presents an analysis on the expressivity of our safe evolution templates. We also

[1] `http://twiki.cin.ufpe.br/twiki/bin/view/SPG/SPLRefactoringTemplates`

Table 1. Safe Evolution Templates for Product Lines

Template	Name
1	Split Asset
2	Refactor Asset
3	Add New Optional Feature
4	Add New Mandatory Feature
5	Replace Feature Expression
6	Add New Alternative Feature
7	Add New OR Feature
8	Delete Asset

show the frequency of use of each template in the evolution scenarios.

5.1 Results

To evaluate the relevance of our safe evolution templates, we analyzed the evolution of two different product lines. We also examined whether our templates were sufficient to express the modifications implemented in all these scenarios. We summarize the results in Table 2 .

We divided the analysis of TaRGeT in two different steps. First, we analyzed the evolution scenarios implemented from release 4.0 to release 5.0 (from January to July 2009) to derive the transformation templates that we presented in Section 4. Then, in the second step, we analyzed the evolution scenarios implemented between releases 5.0 and 6.0 (from July 2009 to January 2010) in order to verify if the templates that we had previously identified could address the changes performed in each scenario. As mentioned in Section 2, we identified 11 evolution scenarios between releases 4.0 and 5.0 and 9 evolution scenarios between releases 5.0 and 6.0.

We also analyzed MobileMedia [13], a PL for media (photo, video and audio) management on mobile devices. MobileMedia has been refactored and evolved to incorporate new features in order to address new scenarios and applications along many releases. It has been implemented in two different versions: (i) an object-oriented Java implementation that uses conditional compilation to implement variabilities; and (ii) an aspect-oriented Java implementation that uses AspectJ aspects to modularize the PL variabilities, which we used in this study. We evaluated releases 4 to 8 and we identified 8 safe evolution scenarios performed in the product line during this period.

Table 2 lists the templates frequency of use in each safe evolution template in TaRGeT and MobileMedia PLs. We count only one occurrence of a template, even if it is used more than once in the same scenario. This is because we want to verify if our set of templates is sufficient to express the evolution steps, no matter how many times they are used.

According to Table 2, we observed that in the TaRGeT PL, Template 6 - Add New Alternative Feature was the most widely used in the analyzed scenarios. We believe that this was due to the implementation of different formats for input use case documents (Word, XML, XLS) and output test suites (XML, HTML and XLS). On the other hand, we did not find any occurrence of Template 8 - Delete Asset.

Among MobileMedia evolution scenarios, we did not found any occurrence of Template 7 - Add New OR Feature because it does not have any OR feature. Template 1 - Split Asset and Template 2 - Refactor Asset were the most used templates. The former was applied with code templates to extract subclass and to extract class member to aspect. The latter was applied in release 5 to introduce a new alternative feature in the PL. Finally, Template 8 - Delete Asset was used once when an exception handling class became no longer useful.

Table 2. Templates Frequence

Template	TaRGeT	MobileMedia
Split Asset	4 (13.79%)	6 (25%)
Refactor Asset	5 (17.24%)	6 (25%)
Add New Optional Feature	5 (17.24%)	4 (16.66%)
Add New Mandatory Feature	3 (10.34%)	1 (4.16%)
Replace Feature Expression	3 (10.34%)	4 (16.66%)
Add New Alternative Feature	8 (27.58%)	2 (8,33%)
Add New OR Feature	1 (3.44%)	0 (0%)
Delete Asset	0 (0%)	1 (4,16%)

Another fact that we have noticed when analyzing both PL history was that some operations usually involved a large number of modified classes. In TaRGeT PL, the average number of modified assets was 12.9 and in MobileMedia PL the average was 11 assets. This might indicate that if we implement the templates together with a development tool, we could also improve productivity. However, evaluate this is not the focus of our work.

In both cases we found that our transformation templates, in addition to the feature model and configuration knowledge refactorings listed in previous works [24], [2] were sufficient to justify all the analyzed evolution scenarios, which reinforces the expressivity of our safe evolution templates. Besides, during our analysis, we identified 3 cases that actually introduced new bugs in the product line, in TaRGeT's second set of evolution scenarios, and 4 cases in MobileMedia. Most of these errors are related to the implementation of a new feature that crashed the behavior of existing products, like the example that we described in our motivating example. We found that if our safe evolution templates had been used, these errors could be avoided.

We have also proved the templates' soundness with respect to the refinement definition. This assures that they do not introduce bugs in the product line. We proved soundness for all templates listed in Table 1 using the Prototype and Verification System (PVS) [20]. This formal proof is done in accordance with the general PL refinement theory (See Section 3). The templates proof details are not in the scope of this work, but they can be retrieved in our website[2].

These two aspects are important because together they address the problem of the likely chance of error in manual PL evolution; we assure that the templates do not introduce new bugs and also that the developer does not need to perform modifications that are not described by the templates.

5.2 Threats to Validity

The study about PL evolution detailed in this work provides encouraging results, but it is not yet complete. Concerning our study, this subsection lists the main threats to validity identified.

Since we performed our analysis manually, it is possible that some evolution scenarios have not been taken into consideration. In this case we may have missed evolution steps that could not be justified by our templates or scenarios that would enhance the expressivity of our set of templates. We identified the scenarios by analyzing different FM versions from each release and also by comparing different versions of assets in the SVN repository. We also relied on commit comments and revision history annotations present in source files that described the changes that developers executed. However, we believe that the identified evolution scenarios can represent real life operations and that the templates that we discovered can be used in different contexts from the ones that we analyzed. Because of the limited quantity of PLs analyzed, the quantitative results cannot be generalized with confidence, but the

[2] http://twiki.cin.ufpe.br/twiki/bin/view/SPG/TheorySPLRefinement

qualitative results are an evidence that our set of safe evolution templates is quite expressive.

Our approach is based on the fact that manually evolving PLs is error prone. In this study we found evidences of this issue by identifying evolution scenarios that were supposed to preserve the behavior of existing products but in fact introduced errors in the PL. Despite this, it is also possible that we have overlooked errors introduced by the manual changes during PL evolution and that the number of errors is even bigger, which encourages us to continue with our research.

Another threat to our work is the fact that MobileMedia is a small system that was developed for educational purposes. However, we observed that the same categories of evolution scenarios that we identified in TaRGeT PL were present in MobileMedia scenarios, which indicates that these scenarios are relevant to our study.

6. Related Work

The notion of product line refinement discussed here first appeared in a product line refactoring tutorial [5]. Besides covering product line and population refactoring, that tutorial illustrates different kinds of refactoring transformation templates that can be useful for deriving and evolving product lines. Another work [6] extended this initial formalization making clear the interfaces between the product line refinement theory and languages used to describe product line artifacts. In our work we use the existing definitions for PL refinement and the idea of safe evolution transformation templates, but we go further by proposing other new templates based on the analysis of a real product line and evaluating these templates in two other product lines.

Early work [10] on product line refactoring focus on Product Line Architectures (PLAs) described in terms of high-level components and connectors. This work presents metrics for diagnosing structural problems in a PLA, and introduces a set of architectural refactorings that can be used to resolve these problems. Besides being specific to architectural assets, this work does not deal with other product line artifacts such as feature models and configuration knowledge, as do the safe evolution templates presented in our work. There is also no notion of behavior preservation for product lines.

Several approaches [16, 17, 19, 26] focus on refactoring a product into a product line, not exploring product line evolution in general, as we do here with our templates. First, Kolb et al. [17] discuss a case study in refactoring legacy code components into a product line implementation. They define a systematic process for refactoring products with the aim of obtaining product lines assets. There is no discussion about feature models and configuration knowledge. Moreover, behavior preservation and configurability of the resulting product lines are only checked by testing. Similarly, Kastner et al. [16] focus only on transforming code assets, implicitly relying on refinement notions for aspect-oriented programs [9]. As discussed here and elsewhere [5] these are not adequate for justifying product line refinement. Trujillo et al. [26] go beyond code assets, but do not explicitly consider transformations to feature model and configuration knowledge as do our templates. They also do not consider behavior preservation; they indeed use the term “refinement”, but in the quite different sense of overriding or adding extra behavior to assets.

Liu et al. [19] also focus on the process of decomposing a legacy application into features, but go further than the previously cited approaches by proposing a refactoring theory that explains how a feature can be automatically associated to a base asset (a code module, for instance) and related derivative assets, which contain feature declarations appropriate for different product configurations. Contrasting with the refinement notion that we rely on, this theory does not consider feature model transformations and assumes an implicit notion of configuration knowledge based on the idea of derivatives. So it does not consider explicit configuration knowledge transformations as we do here. Their work is, however, complementary to ours since we abstract from specific asset transformation techniques such as the one supported by their theory. By proving that their technique can be mapped to the notion of asset refinement, both theories could be used together.

Thüm et al. [25] present and evaluate an algorithm to classify edits on feature models. They classify the edits in four categories: refactorings, when no new products are added and no existing products are removed; specialization, meaning that some existing products are removed and no new products are added; generalization, when new products are added and no existing products removed and arbitrary edits otherwise. In our work, we also analyzed edits in other artifacts like CK and code assets, in addition to FM. However, we are only interested in refactorings and generalization edits, not considering specialization and arbritary edits.

7. Conclusions

In this paper we investigate the safe evolution of PLs and based on the results of this study we present and describe a set of safe evolution templates that can be used by developers in charge of maintaining product lines. The described templates abstract, generalize and factorize the analyzed scenarios and are in accordance with the refinement notion that we rely on. The templates express transformations in feature models and configuration knowledge. We abstract code assets modifications through code transformation templates, which are more precise transformations that deal with changes in code level. Some of our general PL templates might have a set of code transformation templates associated to them.

We also present the preliminary results of a study that we performed to evaluate the evolution of two product lines. We show evidence that the discovered templates can justify all evolution scenarios that we identified in the SVN history of these two PLs. We present examples of how using our templates could avoid the mistakes that we found during our analysis and we show the frequency of occurrence of each template among the analyzed scenarios. These results, in addition to the templates formal proofs, intend to address the problem of the likely chance of erros in manual evolution in product lines.

We know that our results are limited by the context of the two product lines that we analyzed and that new templates (both general and specific for code assets) can be necessary to justify other transformations that we did not analyze in this paper. In order to complement these results, we intend to extend our study by analyzing other product lines from different domains.

Our results also showed evidence that PL manual evolution can be time consuming because it usually involves the analysis and modification of a great number of source code artifacts. We believe that the templates automation integrated with a development tool could address this issue. As future work, we intend to implement our templates integrated with FLiP [4], an existing product line refactoring tool developed by our group. We also plan to execute a controlled experiment to evaluate the time, and consequently, productivity gains when using our templates to evolve product lines.

Acknowledgments

We would like to thank CNPq, a Brazilian research funding agency, and National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, for partially supporting this work. In addition, we thank SPG[3] members for feedback and fruitful discussions about this paper.

References

[1] V. Alves, P. M. Jr., L. Cole, P. Borba, and G. Ramalho. Extracting and evolving mobile games product lines. In *SPLC 2005, Rennes, France*, Lecture Notes in Computer Science, pages 70–81. Springer, 2005.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. J. P. de Lucena. Refactoring product lines. In *GPCE 2006, Portland, Oregon, USA*, pages 201–210. ACM, 2006.

[3] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. *Transactions on Aspect-Oriented Software Development*, 4:117–142, 2007.

[4] V. Alves, F. Calheiros, V. Nepomuceno, A. Menezes, S. Soares, and P. Borba. Flip: Managing software product line extraction and reaction with aspects. In *SPLC*, page 354, 2008.

[5] P. Borba. An introduction to software product line refactoring. In *GTTSE'09 Summer School*, Braga, Portugal, 2009.

[6] P. Borba, L. Teixeira, and R. Gheyi. A theory of software product line refinement. In *ICTAC'10*, pages 15–43, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] A. Cavalcanti, P. Borba, A. Sampaio, and M. Cornelio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, Jan. 2004.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[9] L. Cole and P. Borba. Deriving refactorings for AspectJ. In *In Proc. Int'l Conf. Aspect-Oriented Software Development*, pages 123–134. ACM Press, 2005.

[10] M. Critchlow, K. Dodd, J. Chou, and A. van der Hoek. Refactoring product line architectures. In *1st International Workshop on Refactoring: Achievements, Challenges, and Effects*, pages 23–26, 2003.

[11] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. Addison-Wesley, 2000.

[12] F. Ferreira, L. Neves, M. Silva, and P. Borba. Target: a model based product line testing tool. In *Tools Session of CBSoft 2010*, Salvador, Brazil, 2010.

[13] E. Figueiredo, N. Cacho, M. Monteiro, U. Kulesza, R. Garcia, S. Soares, F. Ferrari, S. Khan, O. C. Filho, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability, 2008.

[14] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.

[15] K. Kang, S. Cohen, J. Hess, W. Novak, and A. S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[16] C. Kastner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *11th International Software Product Line Conference*, pages 223–232. IEEE Computer Society, 2007.

[17] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *21st International Conference on Software Maintenance*, pages 369–378. IEEE Computer Society, 2005.

[18] C. Krueger. Easing the transition to software mass customization. In *4th International Workshop on Software Product-Family Engineering*, volume 2290 of *LNCS*, pages 282–293. Springer-Verlag, 2002.

[19] J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE'06*, pages 112–121. ACM, 2006.

[20] S. Owre, J. Rushby, and N. Shankar. Pvs: A prototype verification system. In *11th International Conference on Automated Deduction*, pages 748–752. Springer-Verlag, 1992. ISBN 3-540-55602-8.

[21] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[22] D. B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

[23] A. Sampaio and P. Borba. Transformation laws for sequential object-oriented programming. In *PSSE*, volume 3167 of *Lecture Notes in Computer Science*, pages 18–63. Springer, 2004.

[24] L. M. Teixeira. Verification and refactoring of configuration knowledge for software product lines. *MSc Dissertation, Federal University of Pernambuco (UFPE)*, 2010.

[25] T. Thüm, D. S. Batory, and C. Kästner. Reasoning about edits to feature models. In *ICSE*, pages 254–264. IEEE, 2009. ISBN 978-1-4244-3452-7.

[26] S. Trujillo, D. Batory, and O. Diaz. Feature refactoring a multi-representation program into a product line. In *GPCE'06*, pages 191–200. ACM, 2006.

[27] F. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[3] http://www.cin.ufpe.br/spg