Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming

Vander Alves, Pedro Matos Jr., Leonardo Cole,
Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho

Informatics Center, Federal University of Pernambuco
P.O. Box 7851 - 50.732-970 Recife PE, Brazil
{vra,poamj,lcn,atv,phmb,glr}@cin.ufpe.br

Abstract. For some organizations, the proactive approach to product lines may be inadequate due to prohibitively high investment and risks. As an alternative, the extractive and the reactive approaches are incremental, offering moderate costs and risks, and therefore sometimes may be more appropriate. However, combining these two approaches demands a more detailed process at the implementation level. This paper presents a method and a tool for extracting a product line and evolving it, relying on a strategy that uses refactorings expressed in terms of simpler programming laws. The approach is evaluated with a case study in the domain of games for mobile devices, where variations are handled with aspect-oriented constructs.

1 Introduction

There are several approaches for developing software Product Lines (PL) [10]: proactive, reactive, and extractive [18]. In the proactive approach, the organization analyzes, designs, and implements a *fresh* PL to support the full scope of products needed on the foreseeable horizon. In the reactive approach, the organization incrementally grows *an existing* PL when the demand arises for new products or new requirements on existing products. In the extractive approach, the organization extracts *existing products* into a single PL.

Since the proactive approach demands a high upfront investment and offers more risks, it may be unsuitable for some organizations, particularly for small to medium-sized software development companies with projects under tight schedules. In contrast, the other two approaches have reduced scope, require a lower investment, and thus can be more suitable for such organizations. Although the extractive and the reactive approaches are inherently incremental, it should be pointed out that the proactive approach can be incremental as well. In this case, products are simply derived based on whatever assets are in the core asset base at the time (a *core asset* is an artifact used in the production of more than one product in a PL). However, there still needs to be a potentially high investment for this first increment and, although we do not need to have all core assets in hand before starting to build products, *all* such assets need to be designed and planned. An interesting possibility is to combine the extractive and the reactive approaches. But, to our knowledge, this alternative has not been addressed systematically at the architectural and at the implementation levels.

In all approaches, variability management must be addressed in the domain: while focusing on exploiting the commonality within the products, adequate support must be available for composing PL core assets with product-specific artifacts in order to derive a particular PL instance. The more diverse the domain, the harder it is to accomplish this composition task, which in some cases may outweigh the cost of developing the PL core asset themselves.

This paper addresses the issues of structuring and evolving the code of product lines in variant domains. In particular, we present a method that relies on the combination of the extractive and the reactive approaches, by initially extracting variation from an existing application and then reactively adapting the newly created PL to encompass other variant products. The method systematically supports both the extractive and the reactive tasks by defining refactorings. Additionally, we show that such transformations are derived from simple Aspect-Oriented Programming (AOP) laws [11]. Further, we evaluate our approach in the context of an mobile game product line. Finally, we provide tool support for the method.

Indeed, there are a number of techniques for managing variability from requirements to code level. Most techniques rely on object-oriented concepts. These techniques, however, are well-known for failing to capture crosscutting concerns, which often appear in variant domains. Mobile games, in particular, must comply with strict portability requirements that are considerably crosscutting, thereby suggesting AOP to handle variation [1], which is explored in our method.

The next section describes our approach, including its strategy and both extractive and reactive refactorings. Section 3 then analyzes how some refactorings from our method can be derived from existing elementary programming laws. In Section 4, the case study evaluating the approach is presented. Tool support for the method is described in Section 5. We discuss related work in Section 6 and offer concluding remarks in Section 7.

2 Method

Contrary to the proactive approach, we rely here on a combination of the extractive and the reactive approaches. Our method [4] first bootstraps the PL and then evolves it with a reactive approach. Initially, there may be one or more independent products, which are refactored in order to expose variations to bootstrap the PL. Next, the PL scope is extended to encompass another product: the PL reacts to accommodate the new variant. During this step, refactorings are performed to maintain the existing product, and a PL extension is used to add a new variant. The PL may react to further extension or refactoring.

The method is systematic because it relies on a collection of provided refactorings. Such refactorings are described in terms of templates, which are a concise and declarative way to specify program transformations. In addition, refactoring preconditions (a frequently subtle issue) are more clearly organized and not tangled with the transformation itself. Furthermore, the refactorings can be systematically derived from more elementary and simpler programming laws [11, 17]. These laws are appropriate because they are considerably simpler than most refactorings, involving only localized program changes, with each one focusing on a specific language construct.

In the remaining of this section, we detail the steps of our strategy, explaining the extractive and the reactive processes, and their associated refactorings. In Section 3, we explain these refactorings in terms of more elementary program transformations.

2.1 Extraction

The first step of our method is to extract the PL: from one or more existing product variants, strategies based on refactorings (detailed in Section 2.3) extract core assets and corresponding product-specific adaptation constructs. These constructs correspond to aspects and possibly supporting classes (classes only appearing in one product). Figure 1 depicts this approach. In this case, only one core asset is shown, but in general there could be more. Additionally, during evolution of the PL, a product-specific asset could become a core asset, in which case it be used to derive at least two PL members.

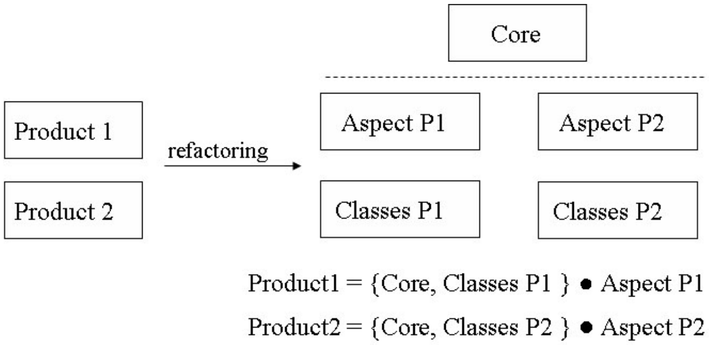

Core
Product 1
Product 2
refactoring
Aspect P1
Aspect P2
Classes P1
Classes P2
Product1 = {Core, Classes P1 } • Aspect P1
Product2 = {Core, Classes P2 } • Aspect P2

Fig. 1. Bootstrapping the Product Line. Core assets appear above the dashed line.

Product 1 and *Product 2* are existing applications in the same domain (for example, versions of a J2ME game for two platforms). *Core* represents commonality within these applications; it is usually a partially specialized OO framework, but can also contain aspects, in which case such aspects modularize the implementation of crosscutting concerns shared by at least two PL instances. The core is composed either with *Aspect P1* and its supporting classes (*Classes P1*), if any, or *Aspect P2* and its supporting classes (*Classes P2*), if any, in order to instantiate the original *specific* products. The • operator represents aspect composition (weaving). These aspects and their supporting classes thus encapsulate product-specific code.

In order to extract the variation within *Product 1* and *Product 2*, thus defining *Aspect P1* and *Aspect P2*, we must first identify it in the existing code base. When more than one variant exists, diff-like tools provide an alternative. In either case, however, such a view is too detailed at this point. Indeed, the developer first needs to determine the general concerns involved. This could be described more concisely and abstractly with concern graphs, whose construction is supported by a specific tool [24]. Concern graphs localize an abstracted representation of the program elements contributing to the implementation of a concern, making the dependencies between the contributing elements explicit. Therefore, the actual first step in identifying these variations is to build a concern graph corresponding to known variability issues.

Once the variability is identified, the developer should analyze the variability pattern within that concern. Depending on the pattern, a refactoring may be applied in order to extract it from the core (Section 2.3). Indeed, refactorings can be used to create product lines in an extractive approach, by extracting product-specific variations into aspects, which can then customize the common core [1, 4].

Although our method focuses on code assets, it is important to describe its interaction with configurability-level artifacts, such as feature models [13]. Indeed, the method requires feature modelling and a configuration knowledge, which are essential for effectively describing the PL variability and product derivation. It is outside the scope of this paper to describe in detail the transformation at the feature model level; we specifically address this issue elsewhere [3].

The mapping between features and aspects needs to be specified by a configuration knowledge mechanism [13], which imposes constraints on features and aspect combinations like dependencies, illegal combinations, and default combinations. Constraints involving only feature combinations are also specified in the feature model. Throughout the paper, we use a configuration knowledge mapping groups of features to aspects and classes: the set of features common to both products map to PL core assets; the set of product-specific features map to product-specific aspects and supporting classes.

However, our method does not bind a particular configuration knowledge. Other mappings are possible, depending on the granularity of the features and aspects. For example, with finer-grained variability, single features–rather than feature subsets–could be mapped directly to aspects. Other examples of configuration knowledge can be found elsewhere [20].

2.2 Evolution

Once the product line has been bootstrapped, it can evolve to encompass additional products. In this process, a new aspect is created to adapt the core to the new variant. Moreover, a new feature is added to the feature diagram in order to represent the new product, and the configuration knowledge is updated to map the new feature to the new aspect (Figure 2).

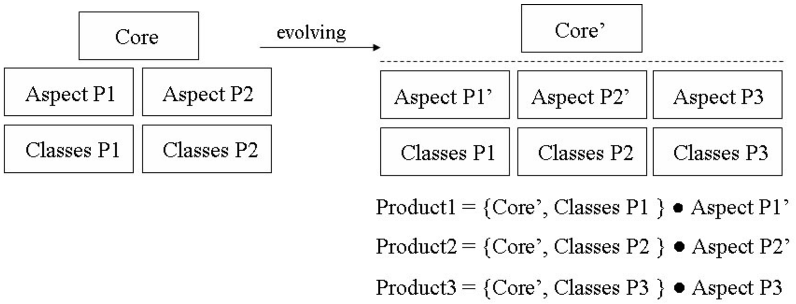

Core
evolving
Core'
Aspect P1
Aspect P2
Classes P1
Classes P2
Aspect P1'
Aspect P2'
Aspect P3
Classes P1
Classes P2
Classes P3
Product1 = {Core', Classes P1 } ● Aspect P1'
Product2 = {Core', Classes P2 } ● Aspect P2'
Product3 = {Core', Classes P3 } ● Aspect P3

Fig. 2. Evolving the Product Line. Core assets appear above the dashed line.

The refactorings in Table 1 (Section 2.3) can also be used for evolution. As Figure 2 also indicates, the core itself may evolve because *some* of the commonality between *Product 1* and *Product 2* might not be shared *completely* by *Product 3*. That is, *Product 3* has *different* commonality with *Product 1* and *Product 2* than these latter have with each other; therefore, a *slightly* different core is necessary. This may trigger further adaptation of the previously existing aspects, too. However, AspectJ tools can identify parts of the core on which these previous aspects depend, and some refactorings could also be aspect-aware according to the definition of Hanenberg et al [15]: evolving the core may change some join points within it, so the aspect-aware refactorings accordingly adjust aspects' pointcuts to refer to the new join points, thereby minimizing the need to revisit such previous aspects. The end of Section 2.3 discusses how our refactorings could be extended to be aspect-aware according to this definition.

Another evolution scenario (Figure 3) involves restructuring the product line to explore commonality within aspects. Such commonality (`AspectFlip` aspect) then becomes a core asset, since it is now explicitly shared by at least two PL instances.

Figure 3 can become more complex with the addition of new platforms and identification of reusable aspects. However, constraints in the feature model as well as the configuration knowledge (the mapping of features to aspects) limit aspect combinations, thereby providing support for scalability.

2.3 Refactoring Catalog

This subsection defines a refactoring catalog, which is a set of refactorings supporting the extractive and the evolutive activities described in the previous subsections. Section 4 shows some strategies (sequence of applications of refactorings from this catalog) that manage to handle the implementation of variable

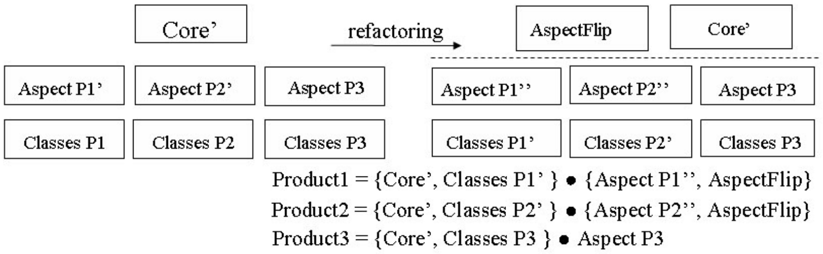

Core’
refactoring
AspectFlip
Core’
Aspect P1’
Aspect P2’
Aspect P3
Aspect P1’’
Aspect P2’’
Aspect P3
Classes P1
Classes P2
Classes P3
Classes P1’
Classes P2’
Classes P3
Product1 = {Core’, Classes P1’ } ● {Aspect P1’’, AspectFlip}
Product2 = {Core’, Classes P2’ } ● {Aspect P2’’, AspectFlip}
Product3 = {Core’, Classes P3 } ● Aspect P3

Fig. 3. Refactoring the Product Line. Core assets appear above the dashed line.

features [13]. We have developed this catalog empirically by analyzing variability in a number of mobile games [4, 1]. It has allowed us to address most variabilities in this domain, but we have not proved this catalog to be complete. We first specify the AspectJ subset necessary for applying these refactorings. Next, we motivate some refactorings by considering an example of feature extraction. Finally, we list the remaining refactorings.

AspectJ subset We consider a subset of AspectJ [11]. This simplifies the definition of transformations and does not compromise our results. However, this may limit the number of refactorings we are able to derive with our laws. For example, the use of `this` to access class members is mandatory. Also, the `return` statement can appear at most once inside a method body and has to be the last command. Additionally, we consider only the following pointcut designators: `call`, `execution`, `args`, `this`, `target`, `within` and `withincode`.

Restricting the use of `this` simplifies the preconditions defined for the laws. This can be seen as a global precondition instead of a restriction to the language. Most of the laws dealing with advice require this restriction. This restriction allows an easy mapping from the executing object referenced from `this` to the executing object exposed inside advice with the pointcut designator `this`.

We only support the mentioned pointcut designators because we think they may represent the core designators of this aspect-oriented language: they have sufficed for us capture join point in 4 different application domains in previous work [11] and in this work. Extending the set of laws to include other AspectJ constructs would be time demanding but not difficult. Besides, it would not affect the already defined laws. This is regarded as future work.

An Example In the context of the mobile device game domain, we consider the optional figures concern of a game. We examine the code declaring and using the `dragonRight` image. First, we consider class `Resources`.

```
class Resources {...
   Image dragonRight;...
   void loadImages() { ...
```

```
      dragonRight = Image.createImage("dragonRight.png");...
  } ...
}
```

where such field is not used anywhere else in the class. The developer may decide that `dragonRight` is an optional feature specific to Platform 1 (P_1) and thus could extract it into an aspect with inter-type declaration and advice constructs. We would thus have

```
class Resources { ...
   void loadImages() {...}
}

Aspect AP1 {
 Image Resources.dragonRight;
 after() returning(): execution(Resources.loadImages()) {
   dragonRight = Image.createImage("dragonRight.png");
 } ...
}
```

where `Resources` now represents a construct in the game core being built and `AP1` denotes an aspect adapting it for a specific platform, namely P_1. The fact that the field is not used anywhere else in the class allowed us to move the attribution towards the method border (end of method in this case), which allows the variation to be described by a single after advice.

Refactorings like these occur frequently and we thus generalize them using a notation that follows the representation of programming laws [11, 17]. Refactoring *Extract Resource to Aspect - after*, whose transformation template is shown shortly ahead, generalizes this transformation and has the purpose of extracting a single variant field, along with part of its usage, into an aspect.

On the left-hand side of Refactoring 1's transformation template, the `f` field and the `this.f=exp;` command (*exp* is an arbitrary expression) denote the variability pattern to be extracted. On the right-hand side, such variability is extracted into aspect `A`. Aspect `A` uses an inter-type declaration construct to introduce field `f` of type T (in the transformation template, T also encompasses the access modifier) into class `C` and an advice construct to add the extracted command to method `m`.

We denote the set of type declarations (classes and aspects) by *ts*. Also, *fs*, *ms* and *pcs* denote field declarations, method declarations and pointcut declarations, respectively. $\sigma(C.m)$ is used to denote the signature of method m of class C, including its return type and the list of formal parameters. $\Gamma(ps)$ denotes the type list from parameters ps, and αps denotes the parameter names from ps. For brevity, we write `exec` and `ret` instead of `execution` and `returning`, respectively.

Each refactoring provides preconditions to ensure that the program is *syntactically valid* (not necessarily syntactically equivalent) and *semantically equivalent* (behavior preserving) after the transformation. The first and second preconditions are necessary to ensure that the code still compiles after applying the transformation, whereas the last three preserve behavior. In particular, although the right-hand side of the refactoring template is not *syntactically* equivalent to the left-hand side, both sides are *semantically* equivalent, since the third refactoring precondition (shown shortly ahead) guarantees that the `this.f=exp` command can be the last one or in the middle of method `m`.

Refactoring 1 ⟨Extract Resource to Aspect - after⟩

```
ts
class C {
   T f
   fs
   ms
   T′ m(ps) {
      body
      this.f = exp;
      body′
   }
}
```

→

```
ts
class C {
   fs
   ms
   T′ m(ps) {
      body
      body′
   }
}
privileged aspect A {
  T C.f;
  after(C cthis, ps) ret(T′ t) :
     exec(T′ C.m(Γ(ps)))
     && this(cthis)
     && args(αps) {
       cthis.f = exp[cthis/this];
  }
}
```

provided

- `A` does not appear in *ts*;
- if the field `f` of class `C` is private, such field does not appear in *ts* nor in *ms*;
- `f` does not appear in *body'*; `exp` does not interfere with *body'*;
- `A` has the highest precedence on the join points involving the signature $\sigma(C.m)$;
- there is no designator `within` or `withincode` capturing join points inside `this.f=exp;`

In the preconditions above, we require that, if the field `f` of class `C` is private, such field does not appear in *ts* nor in *ms* because, when moved to the aspect, the field would be private with respect to the aspect and not with the class, hence a reference to `f` in *ts* or *ms* would not compile (according to AspectJ semantics, visibility modifiers of inter-type declarations are related to the aspect and not to the affected class).

The preconditions on the third bullet are necessary to allow moving the command `this.f=exp;` to the end of method `m`, which is done as an intermediate step during refactoring. Section 3.2 and Figure 4 explain the refactoring in terms of consecutive applications of elementary fine-grained transformations. The precondition requiring `exp` not to interfere with *body'* is specified at a semantic level, but it can also be specified syntactically if we have further information about the structure of *exp*, which happens frequently, including in our example above and in our case study. In such cases, *exp* is a static method call on third-party API to load image attributes, thus not interfering with *body'*.

Despite its syntactic form, the semantic intent of the higher precedence precondition is the following: the newly created after *advice* has the highest precedence on the join points involving the signature $\sigma(C.m)$. However, the only way AspectJ allows specifying precedence among advice of different aspects is by specifying precedence on *aspects* containing these advice, thus implying that *all* advice of a certain aspect `A` have precedence over *all* advice of another aspect `B`, which is a too coarse-grained way to do so. In fact, we may want some advice of `A` to have precedence over some advice of `B` and some advice of `B` to have precedence over advice of `A`, which would lead to an unsolvable constraint among the precedence of such aspects.

Therefore, applying the same refactoring twice works if the code is extracted into the same aspect (advice precedence within the aspects is addressed as shown shortly ahead); otherwise, it will depend on whether there is already a precedence constraint on the existing aspects. If so, the refactoring could not be applied; otherwise, the refactoring can be applied and the new aspect `A` will have the highest precedence. This is a limitation of AspectJ's expressiveness. An AspectJ extension could be accomplished to define advice precedence on a fine-grained approach, by using the ABC compiler [6], for example. In this case, the semantic intent of the refactoring could be expressed syntactically.

The fifth precondition means that there are no `within` or `withincode` pointcut designators in any aspect in the PL that could match join points in the `this.f=exp;` statement. This precondition is necessary because moving such statement may break those pointcuts. Despite declarative, this precondition is verifiable by examining the PL aspects in the IDE using AJDT's API.

The refactoring described creates aspect `A`. A slight variation of this refactoring assumes `A` already exists. In this case, such aspect would have a particular form after applying the transformation:

```
 privileged aspect A {
 T C.f;
 pcs
 bars
 afs
 after(C cthis, ps) ret(T′ t) :
   exec(T′ C.m(Γ(ps)))
   && this(cthis)
   && args(αps) {
     cthis.f = exp[cthis/this];
 }
}
```

Note that, in this case, the advice can not be considered as a set, since order of declaration dictates precedence of advice. According to the AspectJ semantics, if two advice declared in the same aspect are `after`, the one declared later has precedence; in every other case, the advice declared first has precedence. Thus, we divide the list of advice in two. The first part (*bars*) contains the list of all `before` and `around` advice, while the second part contains only `after` advice (*afs*). This separation ensures that `after` advice always appear at the end of the aspect. It also allows us to define exactly the point where the new advice should be placed to execute in the same order in both sides of the refactoring. Additionally, for advice declared in different aspects, precedence depends on their hierarchy or their order in a `declare precedence` construct (this is addressed by the fourth precondition of the refactoring). Similar considerations apply to the remaining refactorings. For brevity, we will assume the aspect is created in each case.

Remaining refactorings Table 1 summarizes all refactorings from our catalog.

Table 1. Summary of Refactorings.

Refactoring	Name
1	Extract Resource to Aspect - after
2	Extract Method to Aspect
3	Extract Context
4	Extract Before Block
5	Extract After Block
6	Extract Argument Function
7	Change Class Hierarchy
8	Extract Aspect Commonality

Some of the refactorings in Table 1, such as *Change Class Hierarchy*, are coarse-grained; others, such as *Extract Argument Function*, are fine-grained;

some, such as *Extract Method to Aspect*, have medium granularity. Part of their names refers to an AspectJ construct that encapsulates the variation. For example, the *Extract Resource to Aspect - after* we described previously extracts the variant part of a concern, appearing as a field and its uses in the class, into AspectJ's `after` construct. Finally, the refactorings we present are not aspect-aware according to the definition of Hannenberg et al [15], but these could be adapted to be so by relaxing some preconditions such as the fifth of the *Extract Resource to Aspect - after* refactoring and accordingly changing the `within` and `withincode` pointcuts involved following the guidelines presented elsewhere [15]. In a broad sense, however, our refactorings are aspect-aware, since they can be used in the presence of aspects and manipulate aspects constructs in transformation templates and preconditions.

3 Formal Reasoning for AspectJ Refactorings

This section analyzes how some aspect-oriented extractive refactorings can be decomposed into or derived from existing elementary programming laws [11], which are simpler and easier to reason about than the refactorings, thereby increasing correctness confidence in such extractive transformations. This is specially relevant because it reduces the burden on testing, which is extremely expensive in the PL scenario. Section 3.1 reviews some existing fine-grained aspect-oriented programming laws [11]. Then, in Section 3.2, we relate such refactorings and laws by showing how the former can be described in terms in of the latter. This is illustrated by decomposing refactoring *Extract Resource to Aspect - after* of Section 2.3 into a set of programming laws.

3.1 Programming Laws

Programming laws [17, 11], like refactorings, are transformation structures which preserve program consistence and behavior. In contrast, they are much simpler than most refactorings: they involve only localized program changes, and each one focuses on a specific language construct.

Differently from refactorings, laws can be applied not only from the left to right side, but also in the opposite direction. Therefore, there are different preconditions depending on the direction the law is used. This is represented by arrows, where the symbol (←) indicates that this precondition must hold when applying the law from right to left. Similarly, the symbol (→) indicates that this precondition must hold when applying the law from left to right. Finally, the symbol (↔) indicates that the precondition must hold in both directions.

For example, the following law has the purpose of adding an after advice. On the left-hand side of the law, $body'$ is the last block of code to execute in method `m`. Thus, we can extract it to an after advice. On the right-hand side, $body'$ is not present in method `m`, although it is executed after the execution of method `m` by an after advice declared in aspect `A`. In this aspect, the symbols used in the advice construct have the same meaning as in Refactoring 1.

Law 1 ⟨Add After-Execution Returning Successfully⟩

```
ts
class C {
   fs
   ms
   T m(ps) {
      body
      body′
   }
}
privileged aspect A {
   pcs
   bars
   afs
}
```

=

```
ts
class C {
   fs
   ms
   T m(ps) {
      body
   }
}
privileged aspect A {
   pcs
   bars
   afs
 after(C cthis, ps) ret(T′ t) :
   exec(T′ C.m(Γ(ps)))
   && this(cthis)
   && args(αps) {
     body′[cthis/this]
   }
}
```

provided

- (→) *body′* does not use local variables declared in *body*; *body′* does not call `super`;
- (↔) `A` has the highest precedence on the join points involving the signature $\sigma(C.m)$;
- (↔) there is no designator `within` or `withincode` capturing join points inside *body′*;

The highest precedence precondition of this law is analogous to the highest precedence precondition of Refactoring 1, which was discussed in Section 2.3. Likewise, the last precondition of this law corresponds to the fifth precondition of Refactoring 1. Therefore, the constraint refers to any aspect.

Examining the left-hand side of this refactoring, we see that *body′* executes before all after advice possibly declared for this join point. This means that the new advice on the right-hand side of the law should be the first one to execute, preserving the order in which the code is executed in both sides of the law. Thus, the after advice should be placed at the end of the after list (*afs*). Moreover, in order to ensure that the new advice created with this law is the first one to execute, we have a precondition stating that aspect `A` has the highest precedence over other aspects defined in *ts*. This precondition must hold in both directions.

The next law represents the language construct which introduces a field into a class. Analyzing this transformation from the left to the right, we can see that `field` declaration is removed from class `C`. However, we introduce `field` in this class by using an inter-type declaration construct declared in aspect `A`.

Law 2 ⟨Move Field to Aspect⟩

```
ts                              ts
class C {                       class C {
   fs                              fs
   T field                         ms
   ms                           }
}                          =    privileged aspect A {
privileged aspect A {              T C.field
   pcs                             pcs
   bars                            bars
   afs                             afs
}                               }
```

provided

(→) The field *field* of class C does not appear in *ts* and *ms*.

This precondition is necessary for the same reason that the second precondition of Refactoring *Extract Resource to Aspect - after* is necessary, which was explained in Section 2.3.

3.2 Deriving Refactorings

In this section we use aspect-oriented programming laws [11] to show that the refactorings previously discussed in Section 2.3 are behavior preserving transformations. Although we do not conduct a strictly formal proof, the derivation is still useful for understanding refactorings in terms of simpler transformations. Additionally, representing the refactorings as a composition of programming laws helps to better define the preconditions under which those refactorings are valid. For their simplicity, programming laws [17] are suitable for this. A complete formal proof requires establishing the validity of the laws with respect to a formal semantics, which is still on going work [12].

The laws we use (defined elsewhere [11]) consider the entire context, and therefore apply to *closed* programs. Nevertheless, their associated side conditions are purely syntactic. Furthermore, although the context is captured for each particular law application, this is by no means a requirement that the context be fixed for successive transformations. If, eventually, a modified context no longer satisfies the conditions of a law previously applied, this does not invalidate the effected transformation; it just means that in the current context the application of the law would not be valid. Accordingly, the laws compose in the sense that their consecutive application is equivalent to a coarse-grained transformation (refactoring). Indeed, such composition is not as flexible as in Hoare's laws [17]–which can be applied to *open* programs–, but has sufficed to derive the refactorings.

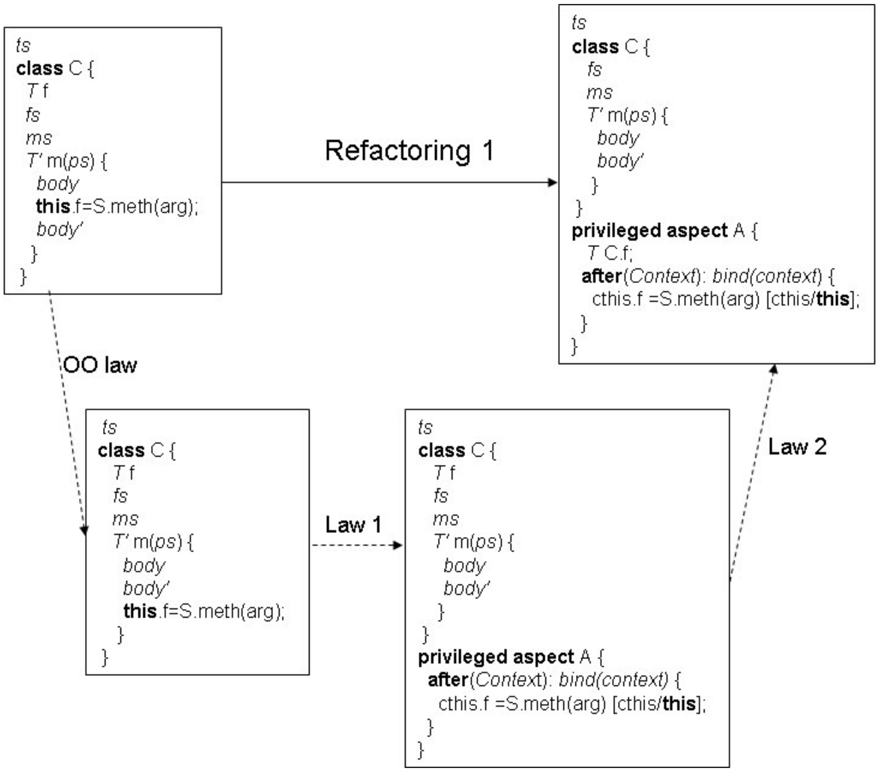
For example, Refactoring 1(*Extract Resource to Aspect - after*), presented in Section 2.3, can be represented as a sequence of object-oriented transformations and aspect-oriented programming laws (Figure 4). In this case, starting from the left-hand side template of this refactoring, we first need to rearrange the source code manipulating field `f` because AspectJ does not provide any mechanism to introduce crosscutting behavior in the middle of a method. In order to move the crosscutting code to an aspect, we first need to move the such code to the beginning or end of the method; this allows the creation of a `before` or `after` advice, respectively. In this refactoring, the crosscutting code was moved to end of the method (we name such transformation by OO law in Figure 4). The OO law holds if the code to be moved is independent of the remaining method code, which is guaranteed by the third precondition of the Refactoring 1 (Section 2.3). Once the crosscutting code is at the end of the method, we can use Law 1 (*Add after-execution returning successfully*), mentioned in Section 3.1, to create a new advice that is triggered after the method's successful execution. At this point, Law 2 (*Move Field to Aspect*) can be applied to extract field `f` into the aspect. The summary of transformations necessary to accomplish this refactoring is shown in Figure 4.

The remaining refactorings can be similarly derived from programming laws. In Table 2, each row summarizes the *derivation* of a refactoring whose name is on the first column (this matches the refactorings from Table 1) in terms of the consecutive applications of aspect-oriented laws (defined elsewhere [11]) in the second column. In the table, consecutive application of laws is represented by →, and repeated application of the same law is denoted with a superscript *.

Table 2. Summary of Refactorings Derivations. Consecutive application of laws is represented by →. Repeated application of a law is denoted with a superscript *.

Refactoring	**Derivation of refactoring in terms of laws**
Extract Method to Aspect	Extract Method → Move Method to Aspect
Extract Resource to Aspect - after	OO Refactoring → Add After-Execution Returning Successfully → Move Field to Aspect
Extract Context	Add Around-Execution
Extract Before Block	Add Before-Execution
Extract After Block	Add After-Execution Returning Successfully
Extract Argument Function	Add Around-Call
Class Hierarcy	Extend From Super Type
Extract Aspect Commonality	Change Advice Order → Move Advice* → Merge Advice

We notice that refactorings can have different levels of complexity when compared to laws. Some refactorings, like *Extract Aspect Commonality*, can be considerably coarse-grained, representing a combination of some laws. On the other



```
ts
class C {
  T f
  fs
  ms
  T' m(ps) {
    body
    this.f=S.meth(arg);
    body'
  }
}
```

```
ts
class C {
  fs
  ms
  T' m(ps) {
    body
    body'
  }
}
privileged aspect A {
  T C.f;
  after(Context): bind(context) {
    cthis.f =S.meth(arg) [cthis/this];
  }
}
```

```
ts
class C {
  T f
  fs
  ms
  T' m(ps) {
    body
    body'
    this.f=S.meth(arg);
  }
}
```

```
ts
class C {
  T f
  fs
  ms
  T' m(ps) {
    body
    body'
  }
}
privileged aspect A {
  after(Context): bind(context) {
    cthis.f =S.meth(arg) [cthis/this];
  }
}
```

Fig. 4. Derivation of Refactoring *Extract Resource to Aspect - after*. The dashed lines denote application of programming laws (fine-grained transformations); the continuous line denote the application of the refactoring (coarse-grained transformation)

hand, some refactorings, like *Extract Before Block*, can be mapped directly into a single law.

4 Case Study: Rain of Fire

Rain of Fire (RoF) is a classic arcade-style game where the player protects a village from different kinds of dragons with catapults. The game is a *commercial product* currently offered by service carriers in South America and Asia. Although it is less than 5K LOC, LOC is neither a necessary nor sufficient condition for complexity. In fact, complexity in the mobile game domain arises mostly due to variability. In general, the mobile game domain is highly variant due to a strong portability constraint: applications have to run in numerous platforms, giving rise to many variant products [2], which are under a tight development cycle, where proactive planning is often unfeasible to achieve.

Although the PL that actually exists in our industrial partner encompasses 12 members, in this case study we investigated how RoF was adapted to run in 3 platforms (P_1, P_2, and P_3), which encompass most variability issues in this PL. P_1 relies solely on MIDP 1.0, whereas P_2 and P_3 rely on MIDP 1.0 and a proprietary API. Some of the variability issues within these products are as follows: optional images, proprietary API for flipping images, screen sizes, and image loading policy. After applying our approach (details shortly ahead), the resulting PL has the feature model of Figure 5, and the following instances, as described by the selection of features.

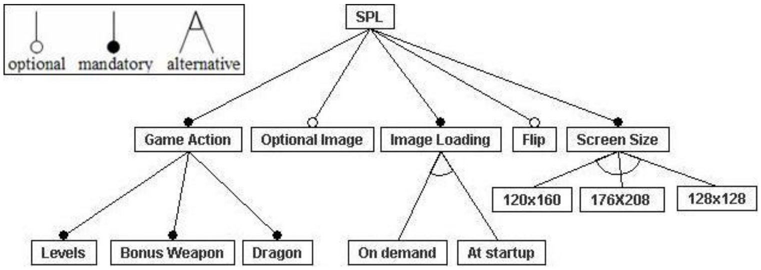


Fig. 5. Variability within Game Product Line

```
P1 = {Dragon, Bonus Weapon, Levels, Optional Image, At startup, Flip, 176x208};
P2 = {Dragon, Bonus Weapon, Levels, On demand, Flip, 120x160};
P3 = {Dragon, Bonus Weapon, Levels, Optional Image, At startup, 128x128};
```

Although this case study has focused only on 3 instances, the feature model shows that other configurations are also possible: the feature model has a total of 24 configurations. Future work is underway to address more configurations.

In order to evaluate our approach, we created a PL implementation of the three products and then compared the PL version with the original implementation of these products. To create and evolve the PL, we first identified the variabilities (such as optional images) with concern graphs and then moved their definition to aspects using the *Extract Resource to Aspect* refactoring. In another step, we addressed method body variability within the platforms. Accordingly, we made extensive use of the *Extract Method to Aspect* refactoring. The *Extract After Block* and *Extract Before Block* refactorings were used when the variant code appeared at the end or beginning of the method body. On the other hand, the *Extract Context* refactoring was used when the variation surrounded common code, representing a context to it. The *Extract Argument Function* refactoring was used when variation appeared as an argument for a method call. Finally, we used the *Change Class Hierarchy* refactoring to deal with class hierarchy variability.

During the evolution of the PL to include P_3, we had to deal with the *load images on demand* concern. This concern was specific to this platform, as it had constrained memory and processing power. To implement this concern, we had to define a method for each screen that could be loaded. Before a screen was loaded, the corresponding method was called. In contrast, in P_1 and P_2 implementations, the images were loaded only once, during game start-up. In this case, there was only one method that loaded all the images into memory. This situation illustrates the scenario in Figure 2.

We addressed this by applying a sequence of *Extract Method* refactorings in the core to break the single method loading all images into finer-grained methods loading images for each screen; the call of this single method was then moved from the core to P_1's and P_2's aspects, and the calls to such smaller methods were moved to P_3's aspect by the *Extract Before Block* refactoring.

Another evolution scenario took place when we realized that some commonality existed between P_1 and P_2 with respect to the *Flip* feature (proprietary graphic API allowing an image object to be drawn in the reverse direction, without the need for an additional image): these two platforms are from the same vendor and share this feature, which is not shared by P_3, from another vendor. Therefore, the *Flip* feature is isolated in the corresponding aspects of P_1 and P_2, but it would be useful to extract this commonality into a single module. In fact, we were able to factor this out into a single generic aspect (`AspectFlip`) with the *Extract Aspect Commonality* refactoring, thus illustrating the scenario in Figure 3.

Table 3 reports the occurrence of each refactoring for achieving the resulting PL. *Extract Method to Aspect* was the most frequently employed, since variability within method body was common for extracting most features. As the PL evolves, we expect to employ *Extract Aspect Commonality* more frequently.

For the resulting the PL, we also employed the *Move Field to Aspect* programming law from Section 3.1. This law was used 28 times. This is consistent with the results of Table 3, since we do not claim these to be complete (the argument in Section 3 is on soundness). Additionally, if we had only used the programming laws themselves instead of the refactorings (composition of the programming laws), we would have to apply approximately twice as many programming laws. In general, the method can combine the refactorings and the programming laws themselves. As the set of refactorings evolve closer to completeness, the direct use of the fine-grained programming laws is expected to decrease and the proportional use of the coarse-grained refactorings is expected to increase.

The resulting configuration knowledge maps sets of features to implementation artifacts:

```
{Levels, Dragon, Bonus Weapon} -> CoreClasses
{Flip} -> AspectFlip
{176x208, Optional Image, At startup} -> AspectP1
{120x160, On demand} -> AspectP2
{128x128, Optional Image, At startup} -> AspectP3
```

where *CoreClasses* is a set of core assets comprised of classes common to all products; `AspectFlip` is a core asset aspect dealing with the *Flip* feature; `AspectP1`,

Table 3. Occurrence of each refactoring

Refactoring	Name	Occurrence
1	Extract Resource to Aspect - after	5
2	Extract Method to Aspect	41
3	Extract Context	1
4	Extract Before Block	2
5	Extract After Block	10
6	Extract Argument Function	1
7	Change Class Hierarchy	1
8	Extract Aspect Commonality	1

`AspectP2`, and `AspectP3` deal partially with specific products features of products P1, P2, and P3, respectively. The arrow notation means that the set of features to its left, which are from the feature model represented in Figure 5, map to the aspects or classes to its right. According to this configuration knowledge and to the configuration of each product presented previously, the PL instances are synthesized by

```
P1 = CoreClasses • {AspectP1, AspectFlip};
P2 = CoreClasses • {AspectP2, AspectFlip};
P3 = CoreClasses • {AspectP3};
```

where • denotes composition. According to this derivation of the PL members, the PL core assets consist of the following: 1) eighteen classes in *CoreClasses*; 2) one core aspect (`AspectFlip`). P1 and P2 are each comprised of *CoreClasses* and two product-specific aspects; P3 is comprised of *CoreClasses* and one product-specific aspect.

Indeed, the configuration knowledge is coarse-grained: there are few reusable aspects across different PL instances. In fact, `AspectFlip` is the only reusable aspect. Current work is underway to explore finer-grained mappings (i.e. more reusable aspects across the PL) and more configurations. Additionally, some scalability issues include having to reduce the granularity of the aspects, so that these become reusable across more instances.

After creation and evolution of the PL, we analyzed code metrics. Table 4 shows the number of Lines of Code (LOC) for each product in the original implementation, in contrast with the PL implementation. We calculate the LOC of a PL instance as the sum of the core's LOC and the LOC of all aspects necessary to instantiate this specific product.

According to Table 4, LOC is slightly higher when comparing each PL instance with the corresponding product in the original implementation. This is caused by the extraction of methods and aspects, which increase code size due to new declarations. On the other hand, there is a 48% reduction in the total LOC of the PL implementation, when compared to the sum of LOCs of the single

Table 4. LOC in original and PL implementations

Original Implementation				**PL Implementation**					
P_1	P_2	P_3	Total	**Core assets**		P_1	P_2	P_3	Total
				Core classes	Core aspects				
2965	2968	3143	9076	2477	72	3042	3047	3210	4405

original versions. This was possible because of the core assets, which represent 57% of the PL LOC. Although, we have a considerable commonality between the three original products source code, it is worth to consider it as different code, because it is repeated for each product and tightly coupled with it. This code repetition increases the effort of program reasoning and maintenance. A reduction due to the avoidance of code repetition could also be obtained using a different product line approach or some modularization techniques, like componentization. Another factor that contributes to the reduction in PL LOC is the existence of reusable aspects.

Table 5 shows the sizes of the PL aspects. The only reusable aspect is considerably smaller than the product-specific ones. The small size of this aspect is convenient for it to be reusable across different PL instances. With a more fine-grained configuration knowledge, we expect that there would be a higher number of reusable aspects and the relative size of the product-specific ones would decrease. Eventually, it could happen that, for some PL instances, no product-specific aspect would be necessary, in which case such instance would be derived solely by reusing core aspects.

Analyzing Table 5 in conjunction with the configuration knowledge presented previously, we can infer that the relative size of aspect code in the PL members ranges from 16% for P1 and P2 to 20% for P3.

Table 5. LOC of aspects in the PL.

Aspect	**LOC**
AspectP1	421
AspectP2	426
AspectP3	661
AspectFlip	72

Another analyzed metric was the packaged application (jar files) sizes of the original and of PL implementations (Table 6). Jar files, that are released to final users, include not only the bytecode files, but also every resource necessary to execute the application, such as images and sound files In the case study products, additional resources represents, on average, 45% of the total jar file size. To measure the impact of our approach on bytecode size, we are considering, in Table 4, the jar files containing only the class files, excluding other resources.

The jar file size is a very important factor in games for mobile devices, due to memory constraints.

Table 6. Jar size (kbytes) in original and PL implementations

	Original Implementation		**PL Implementation**	
	size	reduced size	size	reduced size
P_1	32,4	29,0	67,5	38,4
P_2	33,2	28,8	69,1	33,3
P_3	56,1	52,4	93,5	56,7
Total	98,1	86,6	206,6	104,8

We can notice a jar size increase from original versions to PL instances. The reason for this is the overhead generated by the AspectJ weaver on the bytecode files. We also noticed that very general pointcuts intercepting many join points can lead to greater increases in bytecode file sizes. This considerably influenced us in the definition and use of the refactorings. Moreover, we can gain a significant reduction in the jar size when using a bytecode optimization tool [27]. The reduced size of each original version and PL instance are shown in Table 6.

Although in this case study the PL implementation offers to the user of our approach the same functionality but with a higher application size, our approach is useful mostly because of the benefits that the PL approach brings to the development process: reuse and maintenance are improved, code replication is minimized, and derivation of new products is faster and less costly. Further, the increase on bytecode size can be minimized by further advances in optimization tools. Our initial results show that, in cases where pointcuts matches few join points, by inlining the body of the advice in the base code, we can already reduce bycode size.

5 Tool Support

We have designed and implemented a prototype of a tool for supporting variability management in the PL context. Currently, the tool aims at extracting variations from existing products, by isolating such variations into aspects, which in turn customize the incrementally emerging PL core. The tool currently implements a subset of the refactorings discussed in Section 2. The subset comprises all but the *Extract Aspect Commonality* and the *Extract Argument Function* refactorings.

For example, in order to perform the *Extract Before Block Refactoring*, the user first selects a piece of code to extract (in this refactoring such piece must be at the beginning of a method) and then clicks on the button to perform the refactoring. Next, a dialog box pops up, where the user specifies to which aspect the variability is to be moved. It can be moved to either an existing aspect or to a new one available from the combo box. After confirming the dialog, the tool first checks the refactoring preconditions and, only if these are met, applies the refactoring transformation, resulting in a new version of the emerging core and in a new version of the aspect isolating the selected variability. For performing the other refactorings, the user interacts likewise with the tool.

The prototype has been implemented as an Eclipse plug-in [26]. It defines the *Product Line Perspective*, comprising some buttons in the tool bar corresponding to the refactorings.

Figure 6 illustrates the basic workings of the tool. At first, the user selection from the editor is captured as text via the Workbench API which is used by the *ExtractRefactoringAnalyzer* to look up for the equivalent statements in Abstract Syntax Tree (AST). Then, the refactoring preconditions are checked in the AST. In the sequence, the AST pieces necessary to perform the refactoring transformation are determined. Finally, such the transformation is performed in the original AST and also in the aspect. In the latter, the code is generated as text buffer without manipulating AspectJ's AST API, since such API itself is not currently available. In contrast, the AST API Java is available at JDT's `jdt.core.dom package` [26].

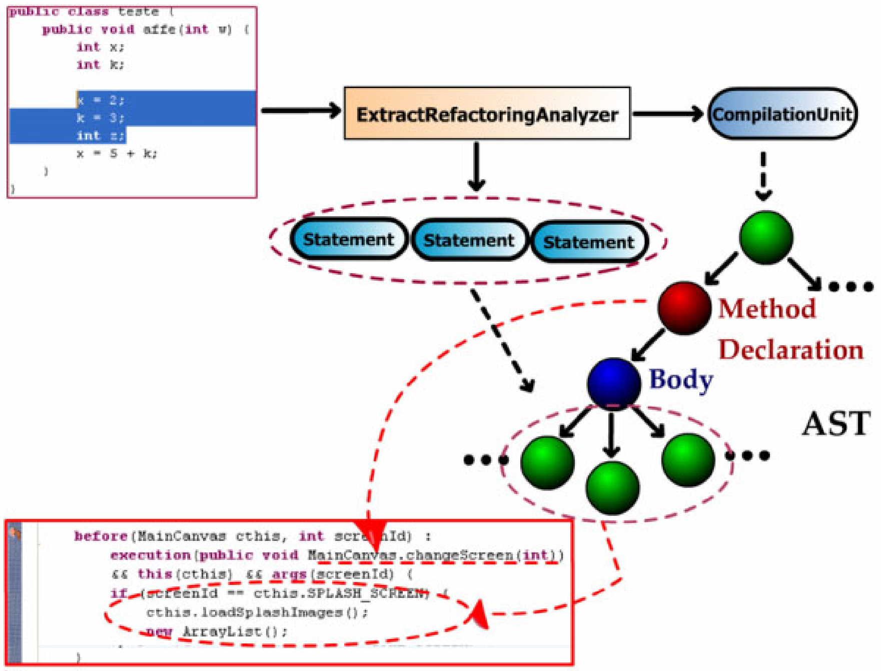
6 Related Work

Our research is in the convergence of a number of areas involving PLs, AOP, refactoring, programming laws, and model refactoring. In the next sections, we compare our work to research in recurring combinations of these areas.

6.1 AOP, PLs, and Refactoring

As in our previous research [4], the current work addresses PL refactoring in the extractive and reactive contexts. However, the current work has three major improvements over our previous research.

Formalization Section 3 of the current work shows how refactorings (coarse-grained program transformations) in the product line context can be understood in terms of programming laws (finer-grained program transformations). The programming laws themselves (not the refactorings) have been defined by the third author in previously [11]. But the novelty in the current work is to decompose the product line refactorings in terms of such laws. Figure 4 shows how Refactoring 1 (Section 2.3) is derived from consecutive applications of these laws, and the second column of Table 2 summarizes how each product line refactoring in the first column of the same table is decomposed into a sequence of application of such laws. Indeed, only the first column of the Table 2 is in previous work, but not the second column (the derivation), a conceptually substantial difference. In this way, the new work presented in Section 3 lays a more formal foundation for



```
public class teste {
    public void affe(int w) {
        int x;
        int k;

        x = 2;
        k = 3;
        int z;
        x = 5 + k;
    }
}
```

```
    before(MainCanvas cthis, int screenId) :
        execution(public void MainCanvas.changeScreen(int))
        && this(cthis) && args(screenId) {
        if (screenId == cthis.SPLASH_SCREEN) {
            cthis.loadSplashImages();
            new ArrayList();
    }
```

Fig. 6. Basic workings of the tool. User selection is parsed and matched in the program AST. Then the refactorings transformations are applied to classes and aspects, and the output generated.

the work presented previously [4]. By expressing refactorings in terms of programming laws, we can increase the confidence that the refactorings are indeed correct. This is feasible because the laws are simpler and easier to reason about than the refactorings themselves [12]. Further, this is relevant because it reduces the burden on testing, which is extremely expensive in the product line scenario.

Extension/refinement of method . Refactoring preconditions are a subtle issue in object-orientation [7, 8] and in aspect-orientation [12]. Accordingly, the preconditions of some refactorings we present have improved slightly since our previous work, and the current work has been updated accordingly. This is also a consequence of having a more formal foundation in the current work.

Tool support The scientific contribution of the prototype itself is rather limited, but it helps to show that our approach can benefit from effective tool support, which is essential in the product line scenarios we focus on (extractive and reactive). Additionally, this is a step towards showing the viability of AOSD in industry.

Prior research by other researches also evaluated the use of AOP for building J2ME product lines [5]. We complement this work by considering the implementation of more features in an real application, explicitly specifying the refactorings to build and evolve the PL, and raising issues in AspectJ that need to be addressed in order to foster widespread application in this domain. Additionally, we rely on concern graphs [24] to identify variant features. Concern graphs provide a more concise and abstract description of concerns than source code. Once the concern is identified, we extract it into an aspect and may further revisit it during PL evolution.

AOP refactorings have also been described elsewhere [23, 16]. The former proposes a catalog for object-to-aspect and aspect-to-aspect refactorings, whereas the latter provides an abstract representation of object-to-aspect refactorings as roles. However, their use in the PL setting is not explored, and the refactorings format follows the imperative style [14]; in contrast, our approach is template-oriented, abstract, concise, and thus does not bind a specific implementation, which could be done, for instance, with a transformation systems receiving as input refactoring templates.

In another approach, a language-independent way to represent variability is provided, and it is shown how it can be used to build J2ME game PLs product lines [29]. Our approach differs from such work because, although ours relies on language-specific constructs, it has the advantage of not having to specify join points in the base. Moreover, their approach, despite language-independent, considerably complicates understanding the source code due to the tags introduced to represent variability.

A recent work [28] reports the AOP refactoring of a middleware system to modularize features such as client-side invocation, portable interceptors, and dynamic types. Nevertheless, such work does not describe the refactorings abstractly and does not attempt to express them in terms of simpler programming laws as a way to guarantee behave preservation, as we do.

6.2 Programming Laws, Model Refactoring, and Optimization

Previous work [11] presented 30 aspect-oriented programming laws and showed how these could derive some aspect-oriented refactorings. In our work, we have explored the usefulness of such approach in validating extractive and evolutive refactorings for building product lines in the mobile game domain. Additionally, this task prompted not only an extension of the number of laws initially proposed, but also a more careful description of some subtle issues of these laws, such as handling AspectJ's precedence semantics, which were skipped in the original work. Finally, the experience in using the laws during derivation suggested that these be organized in a more concise notation, which could lead to the implementation of a generative library.

The process of defining programming laws and showing how these can be used to derive refactorings has also been addressed for for object-oriented languages [7]. Such research additionally formally proves not only the completeness of such set of laws, but also the correctness of each law, by relying on a weakest precondition semantics [9]. Our work, despite not formally proving the laws, still benefits from understanding coarse-grained transformations in terms of simpler ones.

If high-level algebraic specification of products are available, as described in [21], an efficient optimization algorithm could be applied in order to extract the product line core from these specifications with the Shared Class Extractor operator. However, the hypothesis of having this high-level specification may not be met in practice, in such a way that the domain engineer would need to address handling legacy software directly at the design or at the implementation level. Our approach addresses building a product line from existing design/implementation artifacts. Additionally, the variations handled by our approach are considerably crosscutting and may have fine-granularity, which is not the focus of the work mentioned.

6.3 Aspect composition

Section 3.2 shows that the laws compose in the sense that their consecutive application is equivalent to a coarse-grained transformation. Note, however, that the application of each law assumes a fixed context. During composition, this context can change from the application of one law to another, but this shows the laws assume a closed-world approach. Therefore, they are not appropriate for working exclusively with libraries, for example, but are suitable for PLs, since the assets are available in the PL scope and can be considered as the context. Moreover, these compositions of laws have shown to be useful for the case study in this work and in case studies in four other domains [11]. However, more general and functional composition among aspects has shown to be limited within AspectJ [22]. Nevertheless, we believe this is not a strong disadvantage because handling variability in PL does not necessarily need to be addressed by using aspect composition. In fact, feature interaction and interaction between the core and the extension –common PL phenomena– may prevent functional composition from being applied. Current work on XPI [25] and EJP [19] suggest that such interactions are not rare.

7 Conclusion

We present a method and a tool for creating and evolving product lines combining the reactive and extractive approaches. Our method uses a set of refactorings, which can be extended when necessary. We show that these refactorings can be derived from a combination of programming laws, allowing us to better understand these refactorings and increase the confidence that they are correct. This is specially relevant because it reduces the burden on testing, which is extremely expensive in the PL scenario. Our refactorings rely on AOP to modularize crosscutting concerns and to generalize the implementations of these concerns in order to increase code reuse.

Our evaluation with an existing mobile game suggests that we can benefit from extensive code reuse and easily evolve the PL to encompass other products while still maintaining code correctness, since the refactorings are derived from sound elementary programming laws. It also provides some examples of strategies on the applications of refactorings that manage to handle the implementation of variant features.

Although our case study has addressed only a fraction of the configurations from a concrete PL, such fraction exposes most variability issues in the domain. Further, although the evaluation is in the mobile game domain, we argue that the method and the issues addressed here are valid for mobile applications in general, of which mobile games are representative. We also believe that other variant domains could benefit from our method.

Acknowledgements

We would like to thank the anonymous referees for useful suggestions. This research was partially sponsored by CNPq (grants 481575/2004-9, 141247/2003-7), MCT/FINEP/CT-INFO (grant 01/2005 0105089400), and FACEPE.

References

1. Vander Alves. Identifying variations in mobile devices. *Journal of Object Technology*, 4(3):47–52, April 2005.
2. Vander Alves et al. Comparative analysis of porting strategies in j2me games. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 123–132, Budapest, Hungary, 2005. IEEE Computer Society.
3. Vander Alves et al. Refactoring product lines. *5th ACM International Conference on Generative Programming and Component Engineering (GPCE'06)*, To appear, 2006.
4. Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and evolving mobile games product lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 70–81, Rennes, France, Sep 2005. Springer-Verlag.
5. M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *Proceedings of the International Conference on Software Reuse (ICSR)*, 2004.
6. Pavel Avgustinov et al. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, 2005.
7. P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programming*, 52:53–100, oct 2004.
8. Paulo Borba, Augusto Sampaio, and Márcio Cornélio. A refinement algebra for object-oriented programming. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP'03)*, pages 457–482, Darmstadt, Germany.

9. Ana Cavalcanti and David Naumann. A weakest precondition semantics for refinement of object-oriented programs. *IEEE Transactions on Software Engineering*, 26(8):713–728, August 2000.
10. Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
11. Leonardo Cole and Paulo Borba. Deriving refactorings for AspectJ. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-oriented software development*, pages 123–134. ACM Press, 2005.
12. Leonardo Cole, Paulo Borba, and Alexandre Mota. Proving aspect-oriented programming laws. In *Foundations of Aspect-Oriented Languages Workshop at the 4th International Conference on Aspect-oriented software development*, pages 1–9. Iowa State University Technical Report, 2005.
13. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
14. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
15. Oberschulte C. Hanenberg S. and Unland R. Refactoring of aspect-oriented software. In *Net.ObjectDays*, Erfurt, Germany, September 2003.
16. Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, New York, NY, USA, 2005. ACM Press.
17. Charles Antony Richard Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.
18. Charles Krueger. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering.*, pages 282–293, Bilbao, Spain, October 2001.
19. Uirá Kulesza, Vander Alves, Alessandro Garcia, Carlos J. P. de Lucena, and Paulo Borba. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In *Proceedings of the 9th International Conference on Software Reuse (ICSR-9)*, Lecture Notes in Computer Science, pages 231–245. Springer-Verlag, Jun 2006.
20. Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *International Conference on Software Engineering 2006 (ICSE'06)*, Shanghai, China, 2006.
21. Jia Liu and Don S. Batory. Automatic remodularization and optimized synthesis of product-families. In *GPCE*, pages 379–395, 2004.
22. Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA, 2006. ACM Press.
23. Miguel P. Monteiro and Joao M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 111–122, New York, NY, USA, 2005. ACM Press.
24. Martin P. Robillard and Gail C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *In Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, May 2002.

25. Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, New York, NY, USA, 2005. ACM Press.
26. World Wide Web, http://www.eclipse.org. *Eclipse Projetct*, 2004.
27. World Wide Web, http://proguard.sourceforge.net/. *ProGuard*, 2005.
28. Charles Zhang and Hans-Arno Jacobsen. Resolving feature convolution in middleware systems. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 188–205, New York, NY, USA, 2004. ACM Press.
29. Weishan Zhang and Stan Jarzabek. Reuse without compromising performance: Industrial experience from rpg software product line for mobile devices. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, pages 57–69, 2005.