Modeling Features in Aspect-Based Product Lines with Use Case Slices: An Exploratory Case Study

Roberto E. Lopez-Herrejon[1] and Don Batory[2]

[1]Computing Laboratory, University of Oxford, England
[2]Department of Computer Sciences, University of Texas at Austin, USA
rlopez@comlab.ox.ac.uk, batory@cs.utexas.edu

Abstract. A significant number of techniques that exploit aspects in software design have been proposed in recent years. One technique is use case slices by Jacobson and Ng, that builds upon the success of use cases as a common modeling practice. A use case slice modularizes the implementation of a use case and typically consists of a set of aspects, classes, and interfaces. Work on *Feature Oriented Programming (FOP)* has shown how features, increments in program functionality, can be modularized and algebraically modeled for the synthesis of product lines. When AspectJ is used in FOP, the structure of feature modules resembles that of use case slices. In this paper, we explore the relations between use case slices modeling and FOP program synthesis and describe their potential synergy for modeling and synthesizing aspect-based product lines.

1 Introduction

A significant number of techniques that exploit aspects in the realm of design have been proposed in recent years [4]. One technique, proposed by Jacobson and Ng [15], is *use case slices*, which are modular implementations of use cases. Typically, the implementation of a use case slice consists of a set of aspects, classes, and interfaces. A similar structure appears when aspects are used to implement features [16][19], which are increments in program functionality, with *Feature Oriented Programming (FOP)* [10][11], a technology that studies feature modularity in program synthesis for product lines.

In this paper, we present a simple product line example and its implementation in AspectJ. This example helps us illustrate how use case slices can model features in aspect-based product lines and how features can be algebraically modeled for program synthesis. We analyze the relations between use case slices modeling and FOP program synthesis and describe how their potential synergy can serve as a foundation of a methodology for modeling and synthesizing aspect-based product lines.

2 Product Line Example

To illustrate the similarities between use case slices and features we use a simple product line based on the Extensibility Problem [17]. This problem has been widely studied within the context of programming language design, where the focus is achieving data

type and operation extensibility in a type-safe manner. Our focus is on designing and synthesizing a family of programs that we call the *Expressions Product Line (EPL)* [17]. Next we describe in detail this product line and its implementation using AspectJ.

2.1 Example Description

EPL supports a mix of new operations and datatypes to represent expressions of the following language:

```
Exp  :: = Lit | Add | Neg
Lit  :: = <non-negative integers>
Add  :: = Exp "+" Exp
Neg  :: = "-" Exp
```

Two operations can be performed on expressions of this grammar:

1) `Print` displays the string value of an expression. The expression `2+3` is represented as a three-node tree with an `Add` node as the root and two `Lit` nodes as leaves. Operation `Print`, applied to this tree, displays the string "`2+3`".

2) `Eval` evaluates expressions and returns their numeric value. Applying the operation `Eval` to the tree of expression `2+3` yields `5` as the result.

An extra class `Test` creates instances of the datatype classes and invokes their operations.

A natural representation for EPL is a two-dimensional matrix [17]. Rows represent datatypes and columns specify operations. Each matrix entry is a feature that implements the operation, described by the column, on the data type, specified by the row. As a naming convention throughout the paper, we identify matrix entries by using the first letters of the row and the column, e.g., the entry at the intersection of row `Add` and column `Print` is named `ap` and implements operation `Print` on data type `Add`. This matrix is shown in Figure 1 where feature names are encircled.

A program member of this product line is composed from the set of features that are at the intersection of the set of operations (columns) and datatypes (rows) selected for the program. EPL is formed by all the possible combinations of selections of rows and columns. For instance, the program that implements `Print` and `Eval` operations on datatypes `Lit` and `Neg` is composed with features `lp`, `le`, `np`, and `ne`.

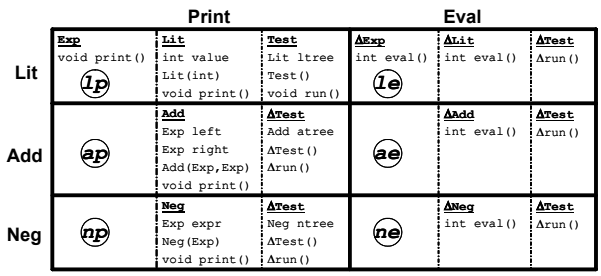

Figure 1. Matrix representation of EPL

2.2 AspectJ Implementation

Let us now analyze how the features of EPL are implemented in AspectJ [17]. Recall that feature `lp` implements operation `print` on datatype `Lit`. Thus the implementation of this feature contains: a) interface `Exp` that declares method `print`, b) class `Lit` with a `value` field, a constructor, and the implementation of `print` method, and c) class `Test` with a field `ltree` of type `Lit`, a constructor that creates an instance of `Lit` and assigns it to `ltree`, and method `run` that calls method `print` on `ltree`. See entry `lp` in Figure 1 for the short depiction of this feature's contents. `lp` can be implemented as follows[1]:

```
// Exp.java
interface Exp { void print( ); }

// Lit.java
class Lit implements Exp {
  int value;
  Lit (int v) { value = v; }
  void print() {
    System.out.print(value);
  }
}
```

```
// Test.java
class Test {
  Lit ltree;
  Test( ) { ltree = new Lit(10); }
  void run( ) { ltree.print( ); }
  void static main(String[] args) {
    Test test = new Test();
    test.run();
  }
}
```

Feature `lp` constitutes the base code in our product line because it contains only standard Java classes and interfaces which are used by all the other features of EPL.

Let us now consider the implementation of feature `le`. This feature implements operation `eval` on `Lit` datatype. It adds the definition of method `eval` to an existing interface `Exp` using an inter-type declaration as follows[2]:

```
// Exple.java
aspect Exple {
   abstract int Exp.eval();
}
```

We refer to this as an *interface extension* [10][11] which we denote with `ΔExp` in Figure 1. Similarly, we refer to the additions to existing classes as *class extensions* [10][11], which are also shown in Figure 1 with symbol Δ prefixed to the name of the class. Feature `le` makes class extensions for classes `Lit` and `Test`. It adds a new method to class `Lit` as follows:

```
// Litle.java
aspect Litle {
  int Lit.eval() { return value; }
}
```

1. Class members privileges are omitted for simplicity.

2. Aspect file names are formed with the name of the class or interface they are extending followed by the feature they help implement. This naming scheme was chosen to make the connection to the algebraic model described in Section 4 clearer.

We refer to this type of extension as *method addition* [10][11] and denote it in Figure 1 with the header of the method. Feature `le` also executes an additional statement in method `run` of class `Test` that calls method eval on field `ltree`. We call this a *method extension* [10][11] and denote it as `Δrun()` in Figure 1. The implementation uses a pointcut that captures the executions of method `run` and gets a reference to the object target of the execution, and an `around` advice that contains the additional statement as shown below:

```
// Testle.java
aspect Testle {
  pointcut LPRun(Test t): execution(void Test.run()) && target(t);
  void around(Test t) : LPRun(t) {
     proceed(t); System.out.println(t.ltree.eval());
  }
}
```

Seasoned AspectJ programmers may wonder at this point why the contents of the three aspects are not aggregated (copied) into a single one. In previous work we showed that composing aspects in this way is not equivalent to their separate file definitions under the current AspectJ precedence rules [18]. Additionally, keeping classes and interfaces extensions into separate aspects improves program understandability [6] and simplifies the algebraic composition model described in Section 4.

As another example, consider the implementation of feature `ap`. First this feature implements operation `print` on the `Add` datatype as follows:

```
// Add.java
class Add implements Exp {
  Exp left, right;
  Add (Exp l, Exp r) { left = l; right = r; }
  void print(){ left.print(); System.out.print("+"); right.print();}
}

// Testap.java
aspect Testap{
  Add Test.atree;
  pointcut APTest(Test t): execution(Test.new()) && target(t);
  void around(Test t) : APTest(t) {
    proceed(t); t.atree = new Add(t.ltree, t.ltree);
 }
 pointcut APRun(Test t):execution (void Test.run(..)) && target(t);
 void around(Test t) : APRun(t) { proceed(t); t.atree.print();}
}
```

Notice that `Testap` implements a *construction extension* denoted as `ΔTest()` in Figure 1. The implementation of the rest of the features is similar to the ones just described.

An EPL program is created by passing all the names of the files that implement its features to the AspectJ compiler or weaver `ajc` [5]. When several pieces of advice apply to the same join point an order of execution must be specified following AspectJ precedence rules as the order is in general undefined. For example, if in the program that implements both operations for `Lit` and `Add` (which we call `LitAdd`) we would like to

execute the method extensions to `run` in order `ap`, followed by that in `le`, and `ae`, we would need to define a precedence clause in an aspect as follows[3]:

```
aspect Ordering {
   declare precedence : Testae, Testle, Testap;
}
```

The whole composition of `LitAdd` becomes:

```
ajc Exp.java Lit.java Test.java Exple.java Litle.java Testle.java
Add.java Testap.java Addae.java Testae.java Ordering.java
-outjar LitAdd.jar
```

With this example, we present how use case slices can be used to model EPL features.

3 Use Case Slices

Use cases are a common technique to capture system functionality and requirements using UML [21]. However the implementation of use cases using traditional object oriented languages and techniques typically breaks use case modularity as their implementation is scattered and tangled in the modules supported by the underlying OO languages. This is the observation that Jacobson and Ng exploit to make the connection with the work on aspects [15]. They propose *use case slices* as a modularization unit to address these problems.

A use case slice contains ([15] pages 111-112):

- **Collaboration**. A collaboration is a set of UML diagrams (interaction, class, etc.) that describe how a use case is realized.
- **Specific Classes**. Classes that are specific to a use case realization.
- **Specific Extensions**. Extensions to existing classes specific to a use case realization.

A use case slice is modeled as a special kind of package with stereotype `<< use case slice >>`. The package has the following basic contents:

- Use case slice name.
- A collaboration symbol (a dashed ellipse) and its name.
- Specific classes. Denoted with the standard UML symbol for classes. These classes may have any relationships of standard class diagrams.
- Specific aspects. Denoted with a symbol similar to UML class. It has stereotype `<<aspect>>`. This symbol has two compartments, one for the pointcuts and one for the class extensions. Aspects may have the same relations between them as supported by AspectJ.

Let us illustrate a use case slice with feature `ap` as shown in Figure 2. Recall that this feature implements the `print` operation on the `Add` datatype. First, notice the name of the use case slice and its collaboration. Since `ap` adds new class `Add`, this class is represented using the standard class symbol. This feature also contains one constructor ex-

3. In [18] we describe several compositional problems that precedence clauses cause.

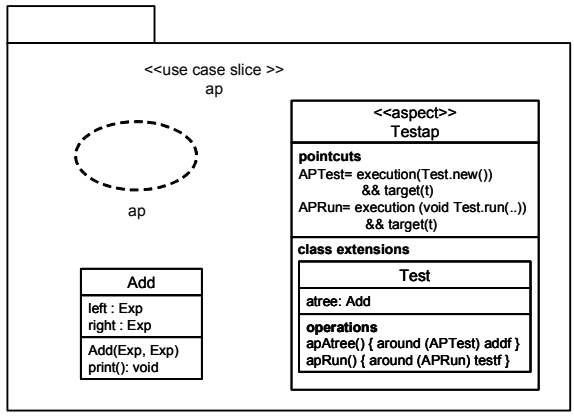

Figure 2. Use case slice for feature `ap`

tension and one method extension to class `Test`. The pointcuts compartment of the `Testap` aspect contains the definitions of pointcuts `APTest` and `APTRun`. The class extensions compartment contains class `Test` as all the extensions that this aspect implements are for this class. In the attributes compartment of the `Test` class the `atree` field appears as it is introduced by the aspect. In the operations compartment, the method extension and constructor extensions are shown. The extensions are given names for reference, `apAtree()` and `apRun()`, and specify the type of advice (`around`), the pointcuts they apply to (`APTest` and `APRun`) and a denotation of their operations, `addf` and `testf` (names chosen arbitrarily) for adding and testing a field (in this case `atree`).

Use case slices have the same relationships as use cases, `extend`, `generalization`, and `include` with a comparable semantics. This relationship can be used to describe how a program of the product line can be composed. To the best of our understanding, use case slices do not provide modeling support for the variability entailed by a product line design, thus a use case slice diagram conveys the design of a single member of a product line. Use case slices can be further modularized into *use case modules*, where each slice modularizes a different model of the use case lifecycle: analysis, design, implementation, testing, etc. [15] (Chapters 4 and 10).

In this section we described the basic ideas of use case slices. However, they can provide more sophisticated modeling functionality . For instance, their pointcuts, classes, and class extensions can be parameterized, using UML templates, to allow extra design flexibility [15]. In next section, we present how EPL can be algebraically modeled with FOP.

4 Feature Oriented Programming (FOP)

Feature Oriented Programming (FOP) is a technology that studies feature modularity and its use in program synthesis. FOP aims at developing a structural theory of programs to express program design, manipulation, and synthesis mathematically whereby program properties can be derived from a program's mathematical representation. In

this context, a program's design is an expression, program manipulation is expression manipulation, and program synthesis is expression evaluation. *AHEAD (Algebraic Hierarchical Equations for Application Design)*, is a realization of FOP that is based on a unification of algebras and step-wise development [8][11]. FOP research predates the work on use case slices and aspects.

4.1 AHEAD in a Nutshell

An AHEAD model of a domain is an algebra that offers a set of operations, where each operation implements a feature. We write `M = {f, h, i, j}` to mean model `M` has operations (or features) `f`, `h`, `i`, and `j`. AHEAD categorizes features as *constants* and *functions*. Constant features represent base programs, those implemented with standard classes and interfaces. For example:

```
f       // a program with feature f
h       // a program with feature h
```

Function features represent *program refinements* or *extensions* that add a feature to the program received as input. For instance:

```
i•x      // adds feature i to program x
j•x      // adds feature j to program x
```

where • means function application. The design of a program is a named expression which we refer as a *program equation*. For example:

```
prog1 = i•f     // prog1 has features f and i
prog2 = j•h     // prog2 has features h and j
prog3 = i•j•h   // prog3 has features h,j,i
```

4.2 An Algebraic Model of EPL

The AHEAD model of EPL is algebraically expressed as a set of features:

```
EPL = { lp, le, ap, ae, np, ne }
```

These features are themselves formed with classes, interfaces, class extensions, and interface extensions. They are denoted as follows (where subscripts identify the feature an element belongs to):

$lp = \{ Exp_{lp}, Lit_{lp}, Test_{lp} \}$ $le = \{ Exp_{le}, Lit_{le}, Test_{le} \}$

$ap = \{ Add_{ap}, Test_{ap} \}$ $ae = \{ Add_{ae}, Test_{ae} \}$

$np = \{ Neg_{np}, Test_{np} \}$ $ne = \{ Neg_{ne}, Test_{ne} \}$

Thus features are hierarchical modules that can contain any number of nested modules. Two features are composed by composing its elements by name (ignoring subscripts). The elements that do not have a match are simply copied to the result of the composition. For example, the composition of `ap•lp` is defined as follows:

$lp = \{ Exp_{lp}, Lit_{lp}, Test_{lp} \}$

$ap = \{ Add_{ap}, Test_{ap} \}$

$ap \bullet lp = \{ Exp_{lp}, Lit_{lp}, Add_{ap}, Test_{ap} \bullet Test_{lp} \}$

A similar composition scheme is only depicted throughout Chapter 4 in Jacobson and Ng's book [15], where it is denoted with symbol +, however its realization is not further described nor elaborated.

Features are implemented as hierarchies of directories and can contain multiple artifacts other than source code. Artifact types are distinguished by the names of the file extensions. Composition of non-code artifacts follows the same principles of source code composition [10] and feature elements are composed when they match both file name and extension. The *AHEAD Tool Suite (ATS)* provides tailored composition tools for different artifacts which are selected by ATS's composer tool according to the artifact type. Currently ATS supports composition of equation files, XML files, and grammar files [8]. Since AHEAD treats all artifacts from all life cycle stages equally, we find that the ideas of use case slides and use case modules are unified or indistinguishable in AHEAD.

Scalability is a prominent concern in any software project. We explain now how AHEAD addresses this concern. Normally, a program is specified in AHEAD by a single expression. By organizing feature models as matrices (or k-dimensional cubes), a program is specified by k expressions, one per dimension. This can drastically simplify program specification, from $O(n^k)$ to $O(nk)$ for k dimensions and n features per dimension [11]. This complexity reduction is key for the scalability of AHEAD's program synthesis. Such matrix (or cube) is called an *Origami Matrix*. An example is the EPL matrix in Figure 1. Each dimension of a matrix is represented with a model. In EPL, the dimensional models are:

```
Operation = { print, eval }
Datatype = { Lit, Add, Neg }
```

Each model lists the features in each dimension. To specify a program, one equation is defined per dimension. For instance, a specification of program `LitAdd` is:

$$\texttt{operation= eval•print} = \prod_{i \varepsilon (\texttt{eval,print})} \texttt{Operation}$$
$$\texttt{datatype = Add • Lit} = \prod_{j \varepsilon (\texttt{Add,Lit})} \texttt{DataType}$$

where $\prod_{i \varepsilon X}$ denotes dot composition of a given sequence X of features. If we denote `MLA` as the projected EPL matrix that forms the intersection of `Lit` and `Add` rows on both columns, `LitAdd` program can be algebraically expressed as:

$$\begin{aligned} P &= \prod_{i \varepsilon (\texttt{eval,print})} \prod_{j \varepsilon (\texttt{Add,Lit})} \texttt{MLA}_{\texttt{operation,datatype}} \\ &= \texttt{ae • le • ap • lp} \\ &= \{ \texttt{Add}_{ae}, \texttt{Test}_{ae} \} \bullet \{ \texttt{Exp}_{le}, \texttt{Lit}_{le}, \texttt{Test}_{le} \} \\ &\quad \bullet \{ \texttt{Add}_{ap}, \texttt{Test}_{ap} \} \bullet \{ \texttt{Exp}_{lp}, \texttt{Lit}_{lp}, \texttt{Test}_{lp} \} \\ &= \{ \texttt{Add}_{ae} \bullet \texttt{Add}_{ap}, \texttt{Lit}_{le} \bullet \texttt{Lit}_{lp}, \texttt{Exp}_{le} \bullet \texttt{Exp}_{lp}, \texttt{Test}_{ae} \bullet \texttt{Test}_{le} \bullet \texttt{Test}_{ap} \bullet \texttt{Test}_{lp} \} \end{aligned}$$

The algebraic representation of origami matrices has proven an useful abstraction to analyze matrix orthogonality, a property that guarantees that the same program is produced for any valid (conforming to design constraints [11]) composition order [9].

AHEAD has been successfully used to synthesize large systems (in excess of 250K Java LOC) from program equations [11]. Currently AHEAD does not support AspectJ, it uses a language called *Jak* that can express all the types of extensions required by EPL. We are working on extending and integrating an algebraic model of AspectJ [18] into

ATS. Nonetheless, the composition model described for EPL still holds. Furthermore, FOP ideas have been used to implement an AspectJ version of the core tools of AHEAD which generates 207+KLOC of which around 30% is aspect code [19].

5 Integrating Use Case Slices and Features

The last two sections explore two seemingly disjoint facets of aspect-based product line development. The first proposes modeling aspect-based features with use case slices while the second describes an algebraic foundation of program composition and synthesis.

On closer inspection there are several similarities. Use case slices consist of classes, interfaces and their extensions implemented with aspects; which is identical to the structure of features. Both features and use case slices can be nested hierarchically and also aim at modularizing non-code artifacts. Similarly, both have relative strengths and drawbacks which we analyze next.

One one hand, we presume that use case slice notation may be easy to adopt for aspect modeling as UML is a popular modeling language. However, we believe the research on use case slices lacks a clear composition model to map use case slices models to concrete working implementations. In terms of source code, the translation to AspectJ is missing an important compositional issue, precedence management. Similarly for other artifacts, we find unclear how such modularization is actually realized (implemented).

On the other hand, the strength of AHEAD is its composition model that supports scalable composition of multiple artifacts backed by an algebraic model. However, for programmers unfamiliar with algebraic notation it may be less intimidating to adopt a familiar modeling notation such as UML.

We believe that the differences and similarities described can be exploited for the development of an aspect-based product line methodology that profits from both lines of work. A feature modeling notation based on use case slices that can ease the adoption by programmers, and an underlying scalable and multi-artifact composition model for program synthesis.

Along the same lines, earlier work by Jacobson hints at the possibility of expressing use case models with a simple algebra of program extensions [14]. However this line of thought is not further pursued in the work of use case slices. We believe our work on AHEAD and FOP could provide a basis for an algebraic foundation for use case slices. We are unaware of any tools that support use case slices and generate AspectJ code from their models. In any case, such kind of tools would encounter the same sort of problems of program synthesis of multiple artifacts faced and solved by AHEAD.

6 Related Work

In UML 2.0 a collaboration is a set of class instances that play different roles [21]. In that sense it is closer to the notion of collaboration-based designs which are the origins of AHEAD [10]. Though use case slices also treat several types of UML diagrams as part of a collaboration.

A close line of work to use case slices is Theme [8]. A *theme*, is an element of design: a collection of structures that represent a feature [8]. Themes are classified into: *base themes* that share structure and behaviour with other themes, and *crosscutting themes* that correspond to aspects. Programs are built by composing themes with a set of binding specifications. Thus Theme and AHEAD classify features in a similar way, but their composition mechanism is significantly different. Also, to the best of our knowledge there is no tool support for this approach. It would be interesting to explore if the composition mechanism of Theme could be expressed in an algebraic notation similar to AHEAD's.

Several extensions of UML to model product lines have been proposed. One example is *Product Line UML-based Software engineering (PLUS)* [13] which is a method that brings FODA [12] modelling ideas to the realm of UML diagrams. PLUS models features as packages of use cases that are stereotyped with the kind of feature they implement such as optional, alternative, etc. Another example is the work of Ziadi and Jézéquel that describes extensions to model variability in class and sequence diagrams and an algorithm for product derivation based on UML model transformations [23]. To what extent this line of work could benefit from aspect research and algebraic modeling is an open question.

There are several pieces of work on aspect-based product line engineering. Anastasopoulus and Muthig propose criteria to evaluate AOP as a product line implementation technology [3]. Alves et al. study product line evolution and refactoring techniques applied to mobile games [2]. Loughran et al. merge natural language processing and aspect oriented techniques to provide tool support for analyzing requirements documents and mining commonality and variability for feature modeling [20].

7 Conclusions and Future Work

In this paper we compare and contrast use case slices and FOP as complimentary facets in the modeling and synthesis of aspect-based product lines. We briefly sketched how these two lines of work can serve as the foundation of a product line methodology that exploits their synergy, feature modeling based on use case slices and program synthesis based on FOP.

We plan to explore how to model algebraically and implement advanced use case slices functionality such as parameterized pointcuts. A promising venue is the work on *Aspectual Mixin Layers (AML)* which allows extensions of pointcuts and pieces of advice using mixin technology [5]. AML provide some support for the parameterization of use case slices. Similarly, the work by Trujillo et al. could be used as a basis for the composition of UML diagrams that are part of a use case slice collaboration [22].

8 References

1. AHEAD Tool Suite (ATS). http://www.cs.utexas.edu/users/schwartz
2. Alves, V., Matos, P., Cole, L., Borba, P., Ramalho, G.: Extracting and Evolving Game Product Lines. SPLC (2005)

3. Anastasopoulus, M., Muthig, D.: An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. ICSR (2004)
4. AOSD Europe. Survey of Analysis and Design Approaches. Deliverable D11.
5. Apel, S., Leich, T., Saake, G.: Aspectual Mixin Layers: Aspects and Features in Concert. ICSE (2006)
6. Apel, S., Batory, D.: When to Use Features and Aspects? A Case Study. GPCE (2006)
7. AspectJ, http://eclipse.org/aspectj/
8. Baniassad, E.L.A, Siobhán, C.: Theme: An Approach for Aspect-Oriented Analysis and Design. ICSE (2004)
9. Batory, D.: Feature Oriented Programming. Class Notes. UT Austin. Spring (2006)
10. Batory, D., Lopez-Herrejon, R.E., Martin, J.P.: Generating Product-Lines of Product-Families. ASE (2002)
11. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE TSE, June (2004)
12. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
13. Gomaa, H.: Designing Software Product Lines with UML. From Use Cases to Pattern-Based Software Architectures. Addison-Wesley (2004)
14. Jacobson, I.: Use cases and Aspects — Working Seemlessly Together. JOT. July (2003)
15. Jacobson, I., Ng, P.: Aspect-Oriented Software Development with Use Cases. Addison-Wewley (2004)
16. Lopez-Herrejon, R.E., Batory, D.: Using AspectJ to Implement Product-Lines: A Case Study. Tech. Report UT Austin CS. TR-02-45. September (2002)
17. Lopez-Herrejon, R.E., Batory, D., Cook, W.: Evaluating Support for Features in Advanced Modularization Techniques. ECOOP (2005)
18. Lopez-Herrejon, R.E., Batory, D., Lengauer, C.: A disciplined approach to aspect composition. PEPM (2006)
19. Lopez-Herrejon, R.E., Batory, D.: From Crosscutting Concerns to Product Lines: A Function Composition Approach. Tech. Report UT Austin CS. TR-06-24. May (2006)
20. Loughran, N., Sampaio, A., Rashid, A.: From Requirements Documents to Feature Models for Aspect Oriented Product Line Implementation. MDD in Product Lines at MODELS (2005)
21. Pilone, D., Pitman, N.: UML 2.0 In a Nutshell. A Desktop Quick Reference. O'Reilly (2005)
22. Trujillo, S., Batory, D., Diaz, O.: Feature Refactoring a Multi-Representation Program into a Product Line. GPCE (2006)
23. Ziadi, T., Jézéquel, J.-M.: Software Product Line Engineering with the UML: Deriving Products. FAMILIES project research book. To appear in Springer LNCS.