2006 International Conference on Software Engineering
Shanghai, China

Feature Oriented Refactoring of Legacy Applications

Jia Liu, Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78712 U.S.A.
{jliu, batory}@cs.utexas.edu

Christian Lengauer
Fakultät für Mathematik und Informatik
Universität Passau
Passau, Germany
lengauer@fmi.uni-passau.de

ABSTRACT

Feature oriented refactoring (FOR) is the process of decomposing a program into features, where a feature is an increment in program functionality. We develop a theory of FOR that relates code refactoring to algebraic factoring. Our theory explains relationships between features and their implementing modules, and why features in different programs of a product-line can have different implementations. We describe a tool and refactoring methodology based on our theory, and present a validating case study.[1]

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: Methodologies, Tools; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering; D.2.11 [**Software Architectures**]: Data abstraction.

General Terms

Design, Languages

Keywords

features, feature interactions, refactoring, program algebra, program synthesis, product lines

1 INTRODUCTION

A *feature* is an increment in program functionality [27]. It is a central abstraction in work on software product lines [4][10] (where each program of a product-line is differentiated from other programs by the set of features it has) and feature interactions (how features influence each other) [7][23][27]. Features are important because they are reusable with in a product-line and can be used to specify program variants *declaratively* — much like specifying PC configurations declaratively by selecting features via web-interfaces, customers want to do the same for specifying programs [6][10]. Concomitantly, program evolution can be described in a high-level and easy-to-understand way as the process of adding and removing features. Giving applications a feature-based design facilitates such extensibility. Designing an application from the ground-up using features is one approach [5]; an alternative is to refactor a legacy application. This paper is about the latter.

Feature oriented refactoring (FOR) is the process of decomposing a program into features. The challenge of FOR is two-fold. First, feature implementations often do not translate cleanly into traditional software modules, such as methods, classes, and packages. Aspects and refinements (e.g., fragments of methods, classes, and packages) are better suited [5][16][14]. This requires a theory of program structure that is based on features. Second, what makes FOR unusual is that feature implementations are not monolithic: the implementation of a feature can vary from one program to another. Stated differently, a feature can be implemented by several modules, some of which are conditional. Only if the right conditions hold — namely that certain other features are present in a program — are these modules actually used.

FOR manipulates program structure in a highly disciplined and sophisticated way. We develop a theory of FOR that relates code refactoring to algebraic factoring, thus providing us with a clean conceptual basis of program structure and manipulation. The theory defines the relationship between features and their implementing modules, and explains why features can have different implementations in different programs. We present a tool and refactoring methodology based on this theory and offer a case study as validation. The case study was used in a previous work on aspect refactoring [14]. Our work lays a mathematical foundation for a new generation of sophisticated program refactoring and synthesis tools.

2 A THEORY OF FOR

2.1 Problem Definition

Metaprogramming is the concept that programming is a computation, i.e., a program (source or executable) is data whose value is to be computed. It has been shown that a feature can be modeled as a function that adds a capability to a program [4][5]. This enables the development of a program to be defined as a computation, starting with a base program and applying a sequence of functions to add features with the desired capabilities. For example, if `B` is a base program, and `F` and `G` are functions that implement different features, a program `P` that is formed by base program `B` extended by the features of `F` and `G` is the expression:

```
P = F(G(B))
```

Feature oriented refactoring is the inverse of feature composition. The goal is to factor a program into an expression that has a base program (value or constant function) composed with one or more features (functions). FOR begins with program `P` and factors it into

ICSE'06, May 20–28, 2006, Shanghai, China.

the expression `F(G(B))`. In our approach, FOR is modeled by *algebraic factoring*, where the terms 'refactoring' and 'factoring' are synonymous. This connection with mathematics gives us a formal basis for defining feature composition and refactoring, where compositional reasoning is a form of algebraic reasoning.

2.2 Implementing Features

Consider program `R`, which is factored into a base program `B` with feature `H`, i.e., `R = H(B)`. Two things can happen when `H` is composed with `B`. First, `H` can add new classes, new methods, and new variables (it can not add classes or class members that are already defined in `B`). Second, `H` can integrate its functionality into `B` by modifying the methods of `B`. To keep the reasoning about features monotonic, we only allow features to extend methods of other features; overriding the definition of a variable or a method introduced in another feature is not allowed.

To illustrate, let `BASE` denote an elementary buffer with `get` and `set` methods. Let `RESTORE` denote a "backup" feature that remembers the previous value of a buffer. Figure 1a shows the `buffer` class of program `BASE` and Figure 1b shows the `buffer` class of program `R = RESTORE(BASE)`. The underlined code indicates the changes `RESTORE` makes to `BASE`. Namely, `RESTORE` adds two members, a `back` variable and a `restore` method, to the `buffer` class, and it also changes the `set` method. While this example is simple, it is typical of features. Adding a feature means adding new members to existing classes and changing existing methods. As programs and features get larger, features can add new classes to a program as well.

```
class buffer {
 int buf = 0;
 int get() {return buf;}
 void set(int x) {
   buf=x;
 }
}
```
(a)

```
class buffer {
 int buf = 0;
 int get() {return buf;}
 int back = 0;
 void set(int x) {
   back = buf;
   buf=x;
 }
 void restore() {
   buf = back;
 }
}
```
(b)

Figure 1. Buffer Variations

Features can be implemented in many ways. One way is to use the AHEAD Tool Suite [5]. The changes to the `buffer` class are defined as a refinement, which adds the `back` and `restore` members and extends the `set` method (Figure 2a). (Method refinement in AHEAD is equivalent to an AspectJ execution pointcut with a single join point. The `Super()` construct is the counterpart of AspectJ's `proceed()`). By composing the `RESTORE` feature of Figure 2a with the base program of Figure 1a, a program that is equivalent to that in Figure 1b is produced.

Another way is to use AspectJ [2]. The aspect `RESTORE` (Figure 2b) defines the `back` variable and `restore` method as introductions (i.e., inter-type declarations), and the modification of the `set` method as advice. By weaving the program of Figure 1a with the aspect of Figure 2b, a program that is equivalent to that in Figure 1b is produced.

Note: Generally, AspectJ aspects are not functions on programs, and are difficult to compose [19]. For the examples and case studies that we consider in this paper, AspectJ can be used. More on this in Section 5.

2.3 The Optional Feature Problem

An unusual characteristic of features, which makes FOR challenging, is that a feature cannot always be implemented by a single module, where a module contains any number of files. Let `LOG` be a feature that prints the state of a buffer. Figure 3a shows a `LOG` module consisting of a single file in AHEAD. Whenever a buffer method is called, the current and previous value of the buffer is printed.

Figure 3b shows a buffer program `T` that is both logged and restorable. It could be factored as `T = LOG(RESTORE(BASE))`. That is, by weaving the `BASE` program (Figure 1a) with `RESTORE` module (Figure 2a) and the `LOG` module (Figure 3a), a program that is equivalent to Figure 3b is produced.

A key motivation for feature refactoring is the ability to customize a program by removing unneeded features. Suppose we want a logged buffer, i.e., `LOG(BASE)`. We cannot build such a program simply by weaving the modules for `BASE` and `LOG`. The reason is that our implementation of `LOG` in Figure 3a assumes the presence of both `BASE` and `RESTORE` as it references variables and methods introduced by both (i.e., `buf`, `back`, `restore`). This restriction on the reusability of the `LOG` feature is an instance of the *optional fea-*

```
refines class buffer {
  int back = 0;
  void restore()
  { buf = back; }

  void set(int x) {
    back = buf;
    Super().set(x);
  }
}
```
(a)

```
aspect RESTORE {
  int buffer.back=0;
  void buffer.restore()
   { buf = back; }

  before(buffer t) :
    target(t) && execution
     (void buffer.set(int))
  { t.back = t.buf; }
}
```
(b)

Figure 2. Possible Feature Implementations

```
refines class buffer{
  void logit()
  { print(buf);
    print(back);
  }

  int get() {
    logit();
    return Super().
           get();
  }
  void set(int x) {
    logit();
    Super().set(x);
  }
  void restore() {
    logit();
    Super().restore();
  }
}
```
(a) Log Feature

```
class buffer {
  int buf = 0, back = 0;
  void logit() {
    print(buf);
    print(back);
  }
  int get() {
    logit();
    return buf;
  }
  void set(int x) {
    logit();
    back = buf;
    buf=x;
  }
  void restore() {
    logit();
    buf = back;
  }
}
```
(b) Logged and Restorable Buffer

Figure 3. Restorable Buffers

ture problem. **RESTORE** is an optional feature, but our implementation of **LOG** makes its presence mandatory.

Feature optionality is possible if a feature is implemented by multiple modules, some of which are conditionally used. Our **LOG** module can be factored into the **baseLOG** and **restoreLOG** modules of Figure 4, each consisting of a single file. If the **LOG** feature is present in a program, the **baseLOG** module is present. The **restoreLOG** module is present only if both **LOG** and **RESTORE** features are present.

```
refines class buffer{
  void logit()
  { print(buf);}

  int get() {
    logit();
    return Super().
           get();
  }
  void set(int x) {
    logit();
    Super().set(x);
  }
}
```
(a) baseLOG

```
refines class buffer {
  void logit() {
    Super().logit();
    print(back);
  }

  void restore() {
    logit();
    Super().restore();
  }
}
```
(b) restoreLOG

Figure 4. Factored LOG Modules

FOR requires a theory that relates features to modules, feature expressions to module compositions, and explains the conditional usage of modules. We develop such a theory in the next sections.

2.4 The Module Structure of Features

Features have a module structure that we need to make explicit. Figure 5 depicts the composition of **H(B)** : **B** is a base program that is represented by a single module **b**. Feature **H** has two modules. Module **h** contains the new classes, members, and methods that are added by feature **H**, and module ∂**b**/∂**H** (read as "changes to **b** by **H**") contains the changes **H** makes to the methods of **b**.

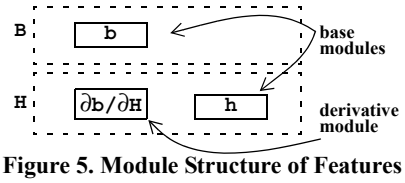


Figure 5. Module Structure of Features

We call **h** and **b** the *base* modules of **H** and **B**. We follow the convention that features and programs have names in **UPPERCASE**, and their base modules have names in **lowercase**. ∂**b**/∂**H** is called a *derivative module*. Derivative modules are differentiated from base modules in that they contain only method refinements; base modules contain classes and introductions. In AOP terminology [16], derivatives contain only advice. They can be implemented with aspects if we limit their join points to the base modules they modify (in Section 5.1 we discuss this in detail). While the name ∂**b**/∂**H** seems fancy, it is a portent of things to come.

> **Example**. In our buffer example, the base module of **BASE**, denoted **base**, is the **buffer** class of Figure 1a. The base module of **RESTORE**, denoted **restore**, is the class refinement of Figure 6a; it contains the **back** and **restore** introductions to the **buffer** class. The derivative module, ∂**base**/∂**RESTORE**, is Figure 6b; it defines a change to the **set** method of module **base**.

```
refines class buffer {
  int back = 0;
  void restore()
   { buf = back; }
}
```
(a) restore

```
refines class buffer {
  void set(int x) {
    back = buf;
    Super().set(x);
  }
}
```
(b) ∂base/∂RESTORE

Figure 6. restore and ∂base/∂RESTORE

We have a notation for the composition of **H** with **B**, namely **H(B)**. We need an expression that tells us how to compose modules **h**, **b**, and ∂**b**/∂**H** to implement **H(B)**. We define two operations on modules and follow rules proposed in [19].

The first is **+**, which we call *introduction sum*. **+** is a binary operation that aggregates base modules by disjoint set union. A base module is a set of unique variables and methods that belong to one or more classes. The base module **restore** has one variable (**back**) and one method (**restore**), both belonging to class **buffer**. Let **a** and **b** be base modules, which have disjoint sets of variables and methods. The introduction sum of **a** and **b**, denoted **a+b**, is formed by disjoint set union — **a+b** is itself a base module. Of course, modules **a** and **b** must be co-designed so that the introduction sum is meaningful. This is the case for feature refactoring of legacy applications.

> **Example**. Figure 7a shows base module **a** containing class **X** (with variable **xx** and method **xy**) and class **Z** (with variable **zz**). Figure 7b shows base module **b** containing class **X** (with variable **qq**) and class **W** (with method **tt**). Figure 7c shows **a+b**.

Introduction sum is associative and commutative. This follows as disjoint set union is associative and commutative.

The second operation, denoted by •, is function composition, which we call *weaving*. Normally • is used to weave the changes of a derivative module into a base module, yielding a woven base module. • can also be used to compose two derivative modules into a composite derivative module.

Recall that a derivative module consists of zero or more refinements (pieces of advice in AOP) that modify the methods of a given base module. ∂**b**/∂**H** is the set of changes made by **H** (consisting of zero or more pieces of advice) that alter the methods of module **b**. The expression ∂**b**/∂**H** • **b** means "weave the changes of derivative module ∂**b**/∂**H** into base module **b**", and when evaluated produces a woven base module. (Section 5 discusses our implementation).

```
class X {
  int xx;
  int xy(){..}
}

class Z {
  float zz;
}
```
(a)

```
class X {
  String qq;
}

class W {
  int tt(){..}
}
```
(b)

```
class X {
  int xx;
  int xy(){..}
  String qq;
}
class W {
  int tt(){..}
}
class Z {
  float zz;
}
```
(c)

Figure 7. Introduction Sum

Example. Evaluating ∂base/∂RESTORE•base yields:

```
class buffer {
  int buf = 0;
  int get() { return buf;}
  void set(int x) { back = buf; buf=x; }
}
```

With the + and • operations, we can now algebraically define the relationship between feature expression H(B) and the module expression that implements it:

$$[\ H(B)\ ] = h + \partial b/\partial H \bullet b \qquad (1)$$

where [E] is the module expression for feature expression E. That is, the result of composing base program B with feature H is the introduction sum of the base module h and the woven base module that is produced by weaving the changes H makes to b with b. The result of evaluating [RESTORE(BASE)] is the buffer class of Figure 1b.

Weaving has two important properties. First, • distributes over + [19], which is a fundamental concept in aspect weaving [2]. Let d be a derivative, and m be a base module that is the introduction sum a+b. We have:

$$\begin{aligned} d \bullet m &= d \bullet (a+b) \\ &= d \bullet a + d \bullet b \end{aligned} \qquad (2)$$

(2) follows because advising module m equals the advising of each of its parts. Second, let m and n be different base modules. Weaving a derivative of m into n makes no change to n; the reason is that a derivative of m makes changes only to m and to no other module. In AOP-speak, the pointcut of a derivative of m targets join points only in m (and none in n):

$$\partial m/\partial F \bullet n = n \qquad (3)$$

Given the above, let's see how the concept of derivatives scales. Suppose we add feature J to program H(B), to produce program J(H(B)). Figure 8 depicts the four modules that J might have. Base module j contains the new variables, methods, and classes that J adds to H(B). In addition, feature J may alter any existing module in H(B). That is, derivative module ∂h/∂J contains the changes that J makes to module h, ∂b/∂J contains the changes that J makes to module b, and $\partial^2 b/\partial J \partial H$ contains the changes that J makes to ∂b/∂H. Module $\partial^2 b/\partial J \partial H$ is called a *second order derivative*.

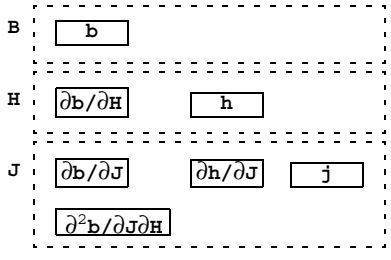


Figure 8. Feature Internal Structures

Example. Our buffer example is too simple to have second order derivatives, but our case study in Section 4 does. In general, $\partial^2 z/\partial X \partial Y$ is a module that encapsulates (before, after, around) advice of targeted methods in module z, and that references members in modules z, x, and/or y. In AHEAD, after advice is expressed by:

```
void methodZ() {            // a method of z
  Super().methodZ();        // proceed()
  code that references members of z, x, and/or y
}
```

Thus, classifying method extension code fragments to derivative modules is fairly simple; it can be done by examining the members of the code fragment that are referenced. The tool described in Section 3 that implements our theory relies on this.

The relationship between feature expression J(H(B)) and its module expression is:

$$\begin{aligned} [\ J(H(B))\ ] = {} & \partial^2 b/\partial J \partial H \bullet \partial b/\partial J \bullet \partial b/\partial H \bullet b \\ & + \partial h/\partial J \bullet h + j \end{aligned} \qquad (4)$$

That is, we weave the changes made by H and J into base module b, we weave the changes made by J into module h, and sum them with module j. Stated differently, *if we refactor a program* P *into the feature expression* J(H(B)), *we need to identify the contents of each of the modules in* (4), *and compose these modules according to* (4) *to rebuild* P. Any or all of the above modules may be *empty*, that is, they make no introductions or changes to existing modules.

Much better than inventing module expressions on a per-example basis is a theory that allows us to derive relations (1) and (4) from a small number of principles.

2.5 Feature Oriented Refactoring Principles

We postulate that derivatives are operators on module expressions. ∂/∂F is an operator that denotes how feature F changes a module. If m is a module, the change is denoted ∂m/∂F. This leads to a set of interesting ideas.

First, we can express changes made by multiple features. For example, the change feature F makes to the change feature Q makes to module m is a *second order derivative*:

$$(\partial/\partial F)\ (\partial m/\partial Q) = \partial^2 m/\partial F \partial Q \qquad (5)$$

Such derivatives have a simple interpretation: $\partial^2 m/\partial F \partial Q$ is a module that contains changes to m by the *combined* features F and Q. Only if features M, F, and Q are present in a program will module $\partial^2 m/\partial F \partial Q$ be used in that program's implementation. Such a module contains AOP (before, around, after, etc.) advice that modifies methods in module m and can reference members in modules m, f, and q, as discussed earlier.

Consider program P in Figure 9a which is formed by the composition G(F(E)). Base module e of feature E defines a class X with variable a and method Foo() (Figure 9b). Features F and G define variables b and c respectively in their corresponding base modules (Figure 9c-d). We want to extend Foo() by appending a statement "a+=b*c;", but only when both features F and G are present. This is expressed by placing this statement in a second order derivative $\partial^2 e/\partial F \partial G$ (Figure 9e). The original program P is reassembled by the composition $\partial^2 e/\partial F \partial G \bullet e + f + g$.

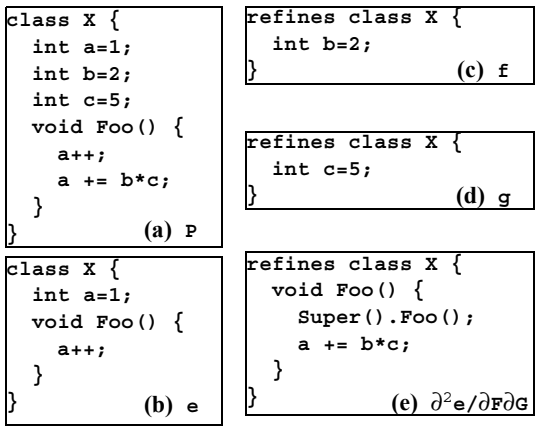


Figure 9. Program **P=F(G(E))**

nth order derivatives have a similar interpretation. Module ∂^n**b/** ∂**F**$_1$…∂**F**$_n$ extends methods in **a** and can reference members in modules **b**, **F**$_1$ … **F**$_n$.

Second, derivatives distribute over expressions involving **+** and **•**. Let **r** and **s** be modules and **F** be a feature:

$$\partial/\partial F\ (r+s) = \partial r/\partial F \bullet \partial s/\partial F \quad (6)$$
$$\partial/\partial F\ (r \bullet s) = \partial r/\partial F \bullet \partial s/\partial F \quad (7)$$

In general, the derivative of a composition of terms is a • composition of the derivative of these terms.

Third, using the + and • operations, we can pose a law that allows us to synthesize module expressions **(1)** and **(4)**:[2]

$$[H](x) = h + \partial x/\partial H \bullet x \quad (8)$$

That is, function **[H]**, with module expression parameter **x**, tells us how feature **H** is implemented in terms of modules. If **x** is primitive, e.g., **x=[B]**, we have:

$$[H(B)] = [H]([B]) = h + \partial [B]/\partial H \bullet [B]$$

If **x** is not primitive, i.e., **x=[A(P)]**, then **(8)** is used to evaluate **[A(P)]**. To expand **(8)** requires the use of **(2)-(8)**.

Equation **(8)** is useful: it tells us how to compose modules to build a program that is defined by a feature expression. It also tells us how to implement a particular feature: every implementation of feature **H** includes **H**'s base module (**h**) plus the changes **H** makes to every module of its input program **x**. The modules that encapsulate these changes are identified in the coefficient ∂**x/**∂**H**.

Recall Figure 8 which graphically depicts how program **J(H(B))** is implemented by modules. The module expressions that define the implementation of the three programs **B**, **H(B)**, and **J(H(B))** are derived successively:

$$[B] = b \quad \text{(def)}$$

$$[H(B)] = [H]([B])$$
$$= h + \partial [B]/\partial H \bullet [B] \quad \text{(def (8))}$$
$$= h + \partial b/\partial H \bullet b \quad \text{(def [B])}$$

$$[J(H(B))] = [J]([H(B)])$$
$$= j + \partial [H(B)]/\partial J \bullet [H(B)] \quad \text{(def (8))}$$
$$= j + \partial (h + \partial b/\partial H \bullet b)/\partial J \bullet (h + \partial b/\partial H \bullet b) \quad \text{(def H[B])}$$
$$= j + \partial h/\partial J \bullet \partial^2 b/\partial J \partial H \bullet \partial b/\partial J \bullet (h+\partial b/\partial H \bullet b) \quad \text{((6)(7))}$$
$$= j + \partial h/\partial J \bullet \partial^2 b/\partial J \partial H \bullet \partial b/\partial J \bullet h + \partial h/\partial J \bullet \partial^2 b/\partial J \partial H \bullet \partial b/\partial J \bullet \partial b/\partial H \bullet b \quad \text{((2))}$$
$$= j + \partial h/\partial J \bullet h + \partial^2 b/\partial J \partial H \bullet \partial b/\partial J \bullet \partial b/\partial H \bullet b \quad \text{((3))}$$

Note that the final expression corresponds to the right side of **(4)**, i.e., we have just derived law **(4)**.

Equation **(8)** also tells us that the module implementation of a feature may vary from one program to another in a product-line. Consider feature **J** and its module expression:

$$[J](x) = j + \partial x/\partial J \bullet x$$

The variability of **J**'s implementation lies in the "coefficient" ∂**x/** ∂**J**. That is, **J**'s module implementation in program **J(H(B))** is base module **j** plus the module expression ∂**[H(B)]/**∂**J**, which we previously found to be:

$$\partial [H(B)]/\partial J = \partial h/\partial J \bullet \partial^2 b/\partial J \partial H \bullet \partial b/\partial J$$

In program **J(C)**, for some other **C**, the difference would be whatever ∂**[C]/**∂**J** expands to. Our case study in Section 4 demonstrates this variance.

3 A PROCESS OF FOR

We have built a tool that implements our theory to refactor legacy Java applications. Our tool is an extension of the Eclipse IDE [25] and utilizes Eclipse's program analysis capabilities to transform a program's parse tree. It guides users through a five-step process, which is explained below.

Step 1 (choose feature expression). Given a legacy application **P**, a user defines a feature expression that specifies what features are present in the program and in what order they are composed. Such expressions are easy to formulate, given **P**'s documentation and knowledge of its functionality. The process is step-wise refinement: you start with a base program and incrementally add more functionality, observing dependency relationships such as feature **X** must be composed before feature **Y** if **X**'s functionality is needed by **Y**. Doing so linearizes the addition of features to the base program, thus leading to a feature expression.

Step 2 (label members). Our theory tells us that there are two kinds of modules: base modules and derivatives (i.e., changes to base modules). Base modules have the property that they contain distinct sets of variables (i.e., data members) and methods. Thus, any variable or method in **P** must belong to precisely one base module. To define this correspondence, we ask users to label each data member and method in **P** with the name of the feature (base module) to which it belongs. Our tool helps by letting users label a few members and it infers the labels of the remaining members. As our labeling algorithm is heuristic (and is a variation of labeling

2. **(8)** is a special case of the weaving rule **A(x)=a(i+x)** in [19]. Since **h** is not affected by the derivative (i.e. advice), the rule simplifies to **A(x)=i+a(x)**.

algorithms used previously by others [21]), several iterations may be necessary to achieve a partitioning (labeling) that is acceptable.

Step 3 (initial refactoring). An initial refactoring of methods into base modules and derivatives is now performed; this step is done automatically by our tool. For each feature, a base module is created to contain its data member and method definitions identified in Step 2. A method body may then be partitioned into a derivative according to the features it references. There are many variations of this process; we illustrate an interesting scenario where a method is extended by multiple features.

Suppose method `M` is introduced by feature $\mathbf{F}_i$ (it is labeled $\mathbf{F}_i$ in Step 2), and its body references members in features $\mathbf{F}_i$, $\mathbf{F}_j$ and $\mathbf{F}_k$ as Figure 10a indicates. Assume that features $\mathbf{F}_j$ and then $\mathbf{F}_k$ are composed in order after $\mathbf{F}_i$, which implies that `M` is refined by $\mathbf{F}_j$ and $\mathbf{F}_k$. Our tool automatically refactors this method into an empty method `M` in base module $\mathbf{f}_i$ and a refinement of `M` in derivative module $\partial^2\mathbf{f}_i/\partial\mathbf{F}_j\partial\mathbf{F}_k$ as indicated in Figure 10b-c. The original method of Figure 10a is reconstructed by evaluating $\partial^2\mathbf{f}_i/\partial\mathbf{F}_j\partial\mathbf{F}_k \bullet \mathbf{f}_i$. Our tool alerts users that it has detected that `M` and other methods like `M` are refined, and further refactoring may be necessary. (If left as is, the body of `M` will be empty in every program that does not have all three features $\mathbf{F}_i$, $\mathbf{F}_j$, and $\mathbf{F}_k$).

Step 4 (refactor derivatives). Here is where knowledge about program `P` kicks in. Suppose a user knows that features $\mathbf{F}_j$ and $\mathbf{F}_k$ are optional, and that the base implementation of method `M` is the line in Figure 10a that references only members of $\mathbf{F}_i$. Figure 10d shows the definition of `M` that is appropriate for base module $\mathbf{f}_i$.

Suppose further when $\mathbf{F}_j$ is added, the line that references members of $\mathbf{F}_j$ appears in `M`. Similarly, when $\mathbf{F}_k$ is added, the line that references members of $\mathbf{F}_k$ appears in `M`. When both $\mathbf{F}_j$ and $\mathbf{F}_k$ are present, the line referencing both members in $\mathbf{F}_j$ and $\mathbf{F}_k$ appears. So the contents of `M`'s body varies, depending on the presence or absence of $\mathbf{F}_j$ and $\mathbf{F}_k$. These variations are realized by refactoring module $\partial^2\mathbf{f}_i/\partial\mathbf{F}_k\partial\mathbf{F}_j$ of Figure 10c into the modules $\mathbf{f}_i$, $\partial\mathbf{f}_i/\partial\mathbf{F}_j$, $\partial\mathbf{f}_i/\partial\mathbf{F}_k$, and $\partial^2\mathbf{f}_i/\partial\mathbf{F}_k\partial\mathbf{F}_j$ in Figure 10d-g. The original method of Figure 10a is now reconstructed by evaluating:

$$\partial^2\mathbf{f}_i/\partial\mathbf{F}_k\partial\mathbf{F}_j \bullet \partial\mathbf{f}_i/\partial\mathbf{F}_k \bullet \partial\mathbf{f}_i/\partial\mathbf{F}_j \bullet \mathbf{f}_i$$

Our tool helps users perform this refactoring. Other refactorings are possible, but this is illustrative of some of the more complicated scenarios that are encountered.

Of course, we expect the original methods may not be easily partitioned without rewriting when feature references are tangled, as first observed by Murphy et al. [21]. Rearranging the order of statements may be needed before methods can be partitioned. As in [21], we ask users to rewrite methods manually. However, as our understanding of this process matures, we expect our tools to analyze and manipulate control flow and data flow graphs to suggest an automatic means of method rewriting and partitioning. Several iterations of this step may be necessary to achieve an acceptable refactoring.

Step 5 (reconstitute program). The previous step defines the base and derivative modules of the program. This step, which is done automatically by our tool, relies on our theory of FOR to translate the feature expression in Step 1 into module expressions that define each feature. (This is accomplished by using `(8)`). A common activity is to create variants of the original program with different sets of features. We create a feature expression for each variant, and features can be reused across different variants. Our theory tells us how to synthesize such a variant automatically by composing the modules defined in Step 4. The resulting program can then be executed to verify that the functionality of a particular feature was successfully isolated in base and derivative modules. In the next section, we review a case study that uses our theory and tool.

4 CASE STUDY: PREVAYLER

We refactored Prevayler [26], an open source Java application that implements an in-memory database and maintains object persistency. As everything is kept in RAM as though "you were just using a programming language" [26], it is much faster than traditional databases. Serializable transactions are supported and queries are run against Java objects. Prevayler has 28 classes and 9 interfaces defined in 2K lines of code. The classes have 101 variables and 181 methods. The interfaces have 21 methods.

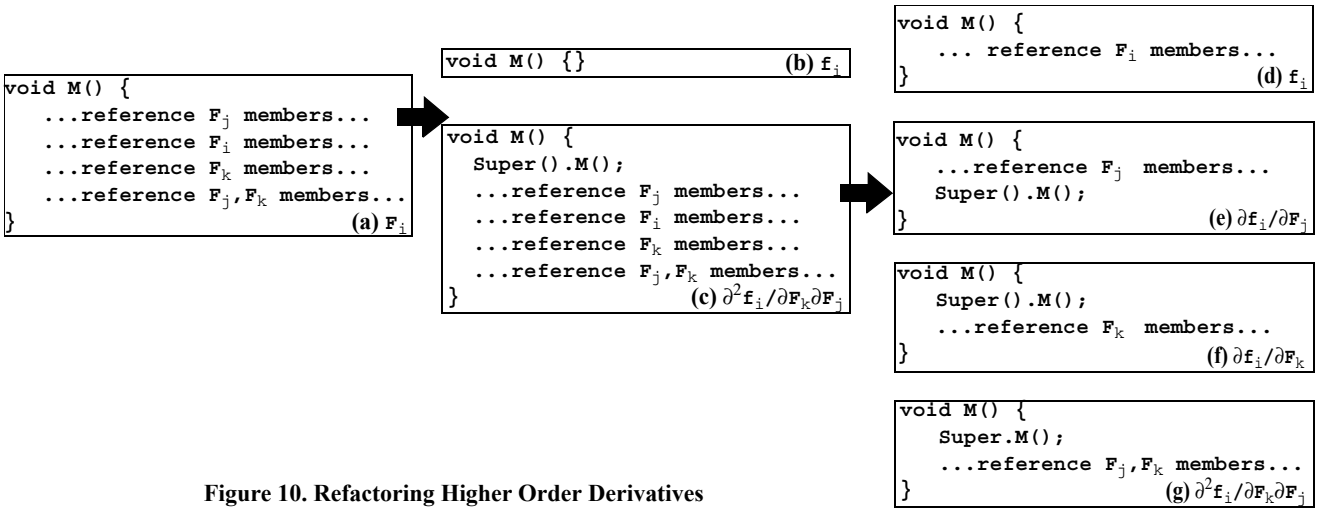
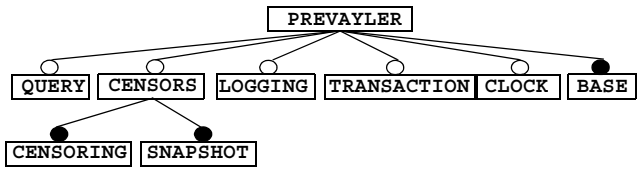


Figure 10. Refactoring Higher Order Derivatives



(a) Feature Diagram

```
// grammar
PREVAYLER :     [QUERY] [CENSORS] [LOGGING]
                [TRANSACTION] [CLOCK] BASE ;
CENSORS  :      CENSORING SNAPSHOT ;

// constraints
TRANSACTION implies CLOCK;
LOGGING implies TRANSACTION;
CENSORING implies (TRANSACTION and SNAPSHOT);
```

(b) GenVoca Grammar

Figure 11. Prevayler Feature Models

Step 1 (choose feature expression). We identified the following set of features in Prevayler by reviewing its manual and source code:

- **BASE**. Base program of the Prevayler framework.
- **CLOCK**. Provides time stamps for transactions.
- **TRANSACTION**. Allows database updating.
- **LOGGING**. Logs transactions.
- **SNAPSHOT**. Writes and reads database snapshots.
- **CENSORING**. Rejects transactions by certain criteria.
- **QUERY**. Retrieves database objects.

Modeling programs as features is not new; it is a core contribution of research in software product-lines [15][9]. We can use results from product-lines to express our feature model of Prevayler. Figure 11a shows a feature diagram: each leaf represents a primitive feature (i.e., the features we identified above), and non-leaves are compound. Solid circles indicate that a feature is required if its parent feature is selected; open circles indicates a feature is optional.

Alternatively, a feature model can be expressed as a GenVoca grammar [6], where primitive features are terminals. GenVoca grammars (like feature diagrams) are not context free. That is, there can be relationships between features in different subtrees and are expressed as additional propositional formulas, such as **CENSORING implies (TRANSACTION and SNAPSHOT)**, meaning if the **CENSORING** feature is present in a program, so too must **TRANSACTION** and **SNAPSHOT** be present. Figure 11b shows a GenVoca grammar. A sentence of this grammar defines a particular member of the Prevayler product-line (i.e., a particular version that can be synthesized from Prevayler features). Approximately 2^5 different versions can be created. For example, a fully-configured version of Prevayler is specified by:

```
PREVAYLOR = QUERY(CENSORING(SNAPSHOT(LOGGING(
        TRANSACTION(CLOCK(BASE))))))          (9)
```

It is not necessary to know all features up-front to perform a FOR. In fact, we identified some of the compound features first, and later

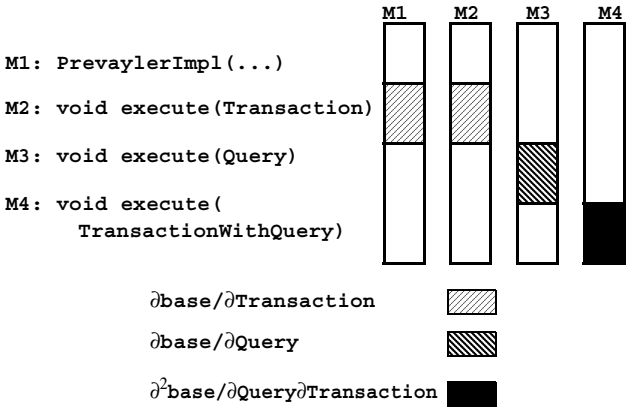


Figure 12. Cross-Cutting Derivatives

decomposed them. Our experience with Prevayler and other programs suggests a divide-and-conquer approach works best: begin with a feature expression of 4-5 terms, and later, decompose these terms if necessary.

Of all the steps in our refactoring process, this step took the longest (2.5 days) as we had to familiarize ourselves on how Prevayler worked.

Step 2 (labeling members). We partitioned the program by assigning every class member to a feature under the guidance of our refactoring tool. As expected, we had to correct previous labelings to better reflect the structure of features. We spent a half-day in this step.

Step 3 (initial refactoring). As this step is automatic, it took but a few minutes once member labels were assigned.

Step 4 (refactoring derivatives). We use one class — **PrevaylerImpl** — to illustrate how derivatives modify class members. Four constructors/methods in **PrevaylerImpl** are depicted in Figure 12. Each has a code fragment that is classified as a refinement in one of three derivatives: ∂**base/∂Transaction**, ∂**base/∂Query** and ∂^2**base/∂Query∂Transaction**. Modules ∂**base/∂Transaction** and ∂**base/∂Query** contain refinements of **base** by features **Transaction** and **Query** respectively. The refinement in ∂^2**base/∂Query∂Transaction** is used when both **Transaction** and **Query** are present. To build the system specified by **(9)**, all derivatives are composed. However, if a variant of Prevayler is created, say without the **Query** feature, then only the ∂**base/∂Transaction** module is used.

When we finished refactoring Prevayler, which took about 1.5 days, we obtained 7 base modules (one for each feature) and 9 derivatives. Table 1 lists all modules and the features to which they belong. According to our theory, a composition of *n* features may lead to $O(2^n)$ distinct modules. Experience suggests that interactions among features are sparse, and a *vast* majority of derivatives are empty. A model of *n* features maps to (approximately) 3**n* non-empty modules, a number that is easily manageable.

Feature	Base Module and Derivatives
BASE	base
CLOCK	clock, ∂base/∂CLOCK
TRANSACTION	transaction, ∂base/∂TRANSACTION
LOGGING	logging, ∂transaction/∂LOGGING
SNAPSHOT	snapshot, ∂base/∂SNAPSHOT
CENSORING	censoring, ∂transaction/∂CENSORING
QUERY	query, ∂base/∂QUERY, ∂^2base/∂QUERY∂TRANSACTION, ∂^2base/∂QUERY∂CLOCK, ∂transaction/∂QUERY

Table 1. Derivatives of Prevayler Features

Step 5 (reconstitute program). After we refactored Prevayler, we were able to implement feature reductions to the application. We created different versions by removing optional features from the original feature expression. We present two here. Suppose a user wants a version V1 that only has clock and query features:

```
V1 = QUERY(CLOCK(BASE))
```

Using the laws of Section 2, we expand [V1] and eliminate empty derivatives (i.e., modules whose names do not appear in Table 1), yielding:

```
[V1] = [QUERY(CLOCK(BASE))]
     = [QUERY]([CLOCK]([BASE]))
     = query + ∂²base/∂QUERY∂CLOCK•
       ∂base/∂QUERY• (clock + ∂base/∂CLOCK•(base))
```

The modules that implement the QUERY feature in V1 are underlined above. V2 is a slightly more complex version that adds TRANSACTION capabilities:

```
[V2] = [QUERY(TRANSACTION(CLOCK(BASE)))]
     = query + ∂transaction/∂QUERY •
       ∂²base/∂QUERY∂TRANSACTION •
       ∂²base/∂QUERY∂CLOCK • ∂base/∂QUERY •
       (transaction + ∂base/∂TRANSACTION •
       (clock + ∂base/∂CLOCK•(base)))
```

The QUERY feature, while reused in both versions, has different implementations in V1 and V2: it contains four derivatives in V2 — ∂base/∂QUERY, ∂^2base/∂QUERY∂CLOCK, ∂^2base/∂QUERY∂TRANSACTION, and ∂transaction/∂QUERY, and only two in V1 (∂transaction/∂QUERY and ∂^2base/∂QUERY∂TRANSACTION are absent). The value of our theory is that it is simple to specify a particular variant of a program; our theory automatically tells us what modules are needed and how to compose them to implement that variant. Tool support is essential, as manipulating these equations is error-prone.

5 RELATED WORK

5.1 Horizontal Decomposition

We selected Prevayler because it was used as a case study for an aspect-oriented refactoring method by Godil, Zhang, and Jacobsen [14][28]. Their method, called *Horizontal Decomposition (HD)*, decomposes programs hierarchically a la Dijkstra [11] using levels of abstraction and step-wise refinement. Program building blocks are features that use AspectJ as the underlying weaving technology. Aspects express changes made by a feature to existing classes and conventional techniques express classes that a feature adds. Together, a feature is implemented by a set of aspect files and a set of class files. A program has a core (which corresponds to our base program), and can have any number of features added to it. Features are composed with the base by super-imposition, i.e., aspects are woven into the base and classes are added to realize desired functionality. Figure 13 shows our graphical depiction of their SNAPSHOT feature. Design patterns were proposed as aids to help identify features in a refactoring process.

HD is closely related to our work. (We drew Figure 13 deliberately in a manner similar to Figure 5 to make this connection clearer). Hierarchically decomposing programs into features — a.k.a. *layers* — and step-wise refinement is the foundation of our model and GenVoca [4]. Our methodology could benefit from HD design patterns. The primary difference with HD is that we have an algebraic theory of feature composition and decomposition. The theory tells us properties that decompositions and their underlying base modules and derivatives must have, and these properties are *automatically* checked by our tools. And our theory formally explains why features can have different implementations in programs of a product-line. HD is informal and (as far as we can tell) relies on users to pick the correct modules to construct customized programs. Our theory automates this task.

HD sheds light on the core equation of our theory (8):

```
[H](x) = h + ∂x/∂H•x                                    (8)
```

An *interpretation* of a theory is the mapping of its concepts to a concrete realization. An HD interpretation of (8) assumes base module h to contain *only* the new classes added by feature H, while ∂x/∂H contains both advice and introductions that change x. This is an alternative and consistent interpretation of our theory.

Not all interpretations are equally powerful. HD does not allow base modules to contain introductions to previously defined classes (as we do). That is, an introduction into a base class of b by feature H must be stored in the derivative module ∂b/∂H. A refinement of this introduction by feature G must be placed in a second-order derivative module (e.g., ∂^2b/∂G∂H), making it harder to recognize that G is really changing H. By allowing base modules to have a finer granularity of introductions, we can make finer distinctions.

We refactored Prevayler prior to an in-depth study of HD. Not surprisingly, the feature set that we used was slightly different than that in [14]. Further, there is a subjective element in defining features and partitioning programs. Their definition of CLOCK, for example, is a superset of ours. Never-the-less, weaving several of

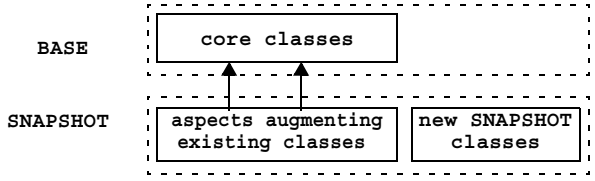


Figure 13. Horizontal Decomposition Structure

their aspects yields refinements that are equivalent to composing one or several of our derivatives. Also, we found refinements in their work that correspond to our notion of higher order derivatives. Consequently, we believe the HD results and ours are in alignment.

The main theoretical difficulty in a clean unification of HD and our work is aspect composition. In general, aspects (as defined in AspectJ) are difficult to compose [19]. Also, an aspect applies to all classes of a program. If an aspect is woven into a program of *n* classes, and subsequently *m* classes are added to the program, the previously woven aspect "comes alive" to weave these additional classes [20]. This behavior is difficult to model by a single function, and thus complicates program reasoning.

In the examples in this paper, these problems were avoided because advice was limited to the refinement of individual methods. In such cases, aspect weaving mimics function composition. An argument on the merits of functional semantics is discussed elsewhere [19].

5.2 Other Related Work

Space precludes a discussion of all the topics to which our work is related. We focus on a key set of contributions.

The inspiration for our work is Prehofer's lifters [22], which our theory formalizes. A preliminary presentation of our ideas was sketched in [18]; it lacked the theory and a more substantial case study that this paper provides. There is an enormous literature on feature interactions in telecommunications (e.g., [7][23][27]). Prior work emphasizes the dynamic or run-time impact features have on each other (e.g. [27]), rather than the structural interactions that we consider. There are a few examples of static interaction models but they too follow Prehofer's initial work [22].

We have built product-lines using AHEAD for many years [5], but never noticed that feature implementations varied among programs. The reason is that all AHEAD features had the form $\mathtt{(8)}$, where the $\partial\mathtt{x}/\partial\mathtt{H}$ coefficient is a constant. If other changes (derivatives) were needed, we simply included another "feature" when building a program, and defined conditions that automatically included that feature. We did not realize that these "extra" or "conditional" features were a symptom of a much deeper result about feature implementations, which this paper exposes.

Other work on feature oriented refactoring focused on identifying the code of a feature (for example, having tools display how feature code is distributed throughout a program) and factoring the code into a single module or aspect [21][24]. Unlike our work, there was (1) no underlying algebraic theory of composition (i.e., to know what to do with the feature after it has been identified and modularized), and (2) no formal notion of interactions among features. Despite these differences, labeling methods to identify feature contents and inferring other labels via program analysis is the approach that we used and that was pioneered in [24].

Another technique to identify features by running test cases has been proposed in [17]. Test cases are classified by the features they belong, and features can be identified by analyzing code blocks that are impacted by each group of test cases.

Feature oriented refactoring is closely related to the area of general refactoring of object oriented programs [13]. For example, creating class refinements is a crucial step in refactoring features from a legacy application. It uses similar techniques that implement extract subclass and push down method/field refactorings which improve OO designs. Recent work on OO refactoring includes, among others, utilizing generic classes [12] and class library migration [3].

One of the issues that our theory exposes is the need for separate compilation of modules [1][8]. The way we deal with module compilation is through a "big inhale", where all source files are parsed and types are resolved prior to weaving. As a consequence, we cannot incrementally evaluate an expression, one operation at a time.

6 CONCLUSIONS

Features play an important role in program evolution, a significant part of which is adding and removing features. Feature oriented refactoring is the process of decomposing a program into features, thus recovering a feature based design and giving it an important form of extensibility.

A distinguishing characteristic of features is that their implementation can vary from one program to another. To explain this, we developed an algebraic theory of FOR that exposes the highly regular structure that features impose on programs. Our theory relates code refactoring to algebraic factoring and defines the relationship between features and their implementing modules. It also dictates how features are constructed from modules for a particular program. We presented a tool and refactoring methodology based on this theory, and offered a case study as validation. We refactored the Prevayler application into a composition of features, and were able to automatically synthesize variants of it by removing optional features.

We believe our algebra lays a mathematical foundation for a new generation of sophisticated program refactoring and synthesis tools that can simplify program construction, evolution, and maintenance.

Acknowledgements. This research is sponsored by NSF's Science of Design Project #CCF-0438786. We thank Hans-Arno Jacobsen for bringing his work to our attention. We also thank Sven Apel, Alar Raabe, and Sebastian Scharinger for their helpful comments.

7 REFERENCES

[1] D. Ancona, G. Lagorio, and E. Zucca, "True Separate Compilation of Java Classes". *PPDP 2002*.

[2] AspectJ Manual, `www.eclipse.org/aspectj/doc/progguide/language.html`.

[3] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring Support for Class Library Migration". *OOPSLA 2005*.

[4] D. Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components". *ACM TOSEM*, October 1992.

[5] D. Batory, J.N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement". *IEEE TSE*, June 2004.

[6] D. Batory, “Feature Models, Grammars, and Propositional Formulas”. *SPLC 2005*.

[7] M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec, “Feature Interaction: A critical Review and Considered Forecast”. Computer Networks 41(1), 115-141, January 2003.

[8] L. Cardelli, “Program Fragments, Linking, and Modularization”. *POPL 1997*.

[9] K. Czarnecki and U. Eisenecker. *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, 2000.

[10] K. Czarnecki, S. Helsen, and U. Eisenecker, “Formalizing Cardinality-based Feature Models and their Specialization”. *Software Process Improvement and Practice, 2005 10(1)*.

[11] E.W. Dijkstra, “The Structure of THE Multiprogramming System”. *CACM*, May 1968.

[12] A. Donovan, et al. “Converting Java programs to use generic libraries”. *OOPSLA 2004*.

[13] M. Fowler, et al. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] I. Godil and H.-A. Jacobsen, “Horizontal Decomposition of Prevayler”. *CASCON 2005*.

[15] K. Kang, et al. “Feature Oriented Domain Analysis (FODA) Feasibility Study”. CMU/SEI-90 TR-21, November 1990.

[16] G. Kiczales, et al., “Aspect-Oriented Programming”. *ECOOP 1997*.

[17] D. Licata, C. Harris, and S. Krishnamurthi, “The Feature Signatures of Evolving Programs” (short paper). *ASE 2003*.

[18] J. Liu, D. Batory, and S. Nedunuri, “Modeling Interactions in Feature Oriented Designs”. *ICFI 2005*.

[19] R. Lopez-Herrejon, D. Batory, and C. Lengauer, “A Disciplined Approach to Aspect Composition”. *PEPM 2006*.

[20] M. McEachen and R.T. Alexander, “Distributing Classes with Woven Concerns - An Exploration of Potential Fault Scenarios”. *AOSD 2005*.

[21] G. C. Murphy, et al., “Separating Features in Source Code: An Exploratory Study”. *ICSE 2001*.

[22] C. Prehofer, “Feature Oriented Programming: A Fresh Look at Objects”. *ECOOP 1997*.

[23] S. Reiff-Marganiec and M.D. Ryan, ed., *Feature Interactions in Telecom. and Software Systems VII*, IOS Press, 2005.

[24] M. P. Robillard and G. C. Murphy, “Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies”. *ICSE 2002*.

[25] **`www.eclipse.org/`**.

[26] **`www.prevayler.org/`**.

[27] P. Zave, “Distributed Feature Composition: Middleware for Connection Services”. **`www.research.att.com/projects/dfc`**.

[28] C. Zhang and H.-A. Jacobsen, “Resolving Feature Convolution in Middleware Systems”. *OOPSLA 2004*.