Generating Product-Lines of Product-Families

Don Batory, Roberto E. Lopez-Herrejon, Jean-Philippe Martin
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
{batory, rlopez, jpmartin}@cs.utexas.edu

Abstract[1]

GenVoca is a methodology and technology for generating product-lines, i.e. building variants of a program. The primitive components from which applications are constructed are ***refinements*** *or* ***layers****, which are modules that implement a feature that many programs of a product-line can share. Unlike conventional components (e.g., COM, CORBA, EJB), a layer encapsulates fragments of multiple classes. Sets of fully-formed classes can be produced by composing layers. Layers are modular, albeit unconventional, building blocks of programs.*

But what are the building blocks of layers? We argue that facets is an answer. A facet encapsulates fragments of multiple layers, and compositions of facets yields sets of fully-formed layers. Facets arise when refinements scale from producing variants of individual programs to producing variants of multiple integrated programs, as typified by ***product families*** *(e.g., MS Office).*

We present a mathematical model that explains relationships between layers and facets. We use the model to develop a generator for tools (i.e., product-family) that are used in language-extensible ***Integrated Development Environments (IDEs)****.*

1. Introduction

Over the last thirty years, program modularity has been dominated by *object-orientation (OO)*: method, class, and package encapsulations are standard concepts. Over this same period, another form of program modularity has arisen. The concept is *feature refinement* — that is, a module that encapsulates the implementation of a *feature*, which is a product characteristic that customers view as important in describing and distinguishing programs within a family of related programs (e.g., a product-line) [15].

Feature refinement is a very general concept and many different implementations of it have been proposed, each with different names, capabilities, and limitations: layers [2], features [18], collaborations [25, 35, 21], subjects [16], aspects [19], and concerns [33]. Unlike traditional component technologies (such as COM, CORBA, and EJB), a feature refinement encapsulates not an entire method or class, but rather fragments of methods and classes. Figure 1 depicts a package of three classes, **`c1`**—**`c3`**. Refinement **`r1`** *cross-cuts* these classes, i.e., it encapsulates fragments of **`c1`**—**`c3`**. The same holds for refinements **`r2`** and **`r3`**. Composing refinements **`r1`**—**`r3`** yields a package of fully-formed classes **`c1`**—**`c3`**. Because refinements reify levels of abstraction, feature refinements are often called *layers* — a name that is visually reinforced by their vertical stratification of **`c1`**—**`c3`** in Figure 1. As refinements, layers, and features are so closely related, their terms are often used interchangably. In general, layers are modular, albeit unconventional, building blocks of programs.

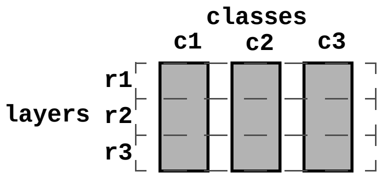


Figure 1. Classes and Refinements (Layers)

This raises an interesting question: if layers (features) are the building blocks of programs, what are the building blocks of layers (features)? We argue that an answer is a *facet*. The idea is simple: Figure 2 depicts a set of three layers, **`r1`**—**`r3`**. Facet **`f1`** *cross-cuts* these layers, i.e., it encapsulates fragments of **`r1`**—**`r3`**. The same for facets **`f2`** and **`f3`**. Composing facets **`f1`**—**`f3`** yields fully-formed layers **`r1`**—**`r3`**. Although it appears that Figure 2 is just Figure 1 turned on its side, where classes and facets are indistinguishable, this is not the case. Facets are *not* classes.

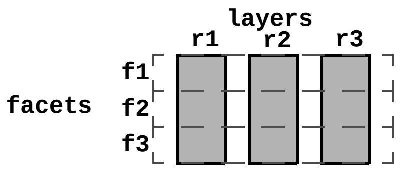


Figure 2. Facets

1. This work was supported by the U.S. Army *Simulation and Training, and Instrumentation Command (STRICOM)* contract N61339-99-D-10.

A year ago, we would not have believed facets to exist or, if they did, to have any utility. To our surprise, we now believe that they are very common. Facets arise when feature refinements scale beyond the confines of an individual program or package. As a perspective, contemporary models of feature refinements allow clients to customize *individual* programs; the set of all program variants that can be produced is a *product-line*. In contrast, a *product-family* is an integrated *suite* of programs, each program having different capabilities [9]. Microsoft Office is an example; it includes the Excel (spreadsheet), Word (text processor), and Access (database) programs. Given how common product-families are, an interesting question is: can feature refinements scale to define a product-line of product-families?

In this paper, we present new results on feature refinement modularity. We show that refinements do scale to product-families and there are interesting twists in doing so. Previously considered "atomic" refinements are revealed to be composed of more elementary refinements called *gluons*. Gluons are arranged in regular ways to form both "atomic" refinements and facets. We present a model of gluons, called *Origami*, that reveals software to have an elegant mathematical structure that leads to simpler designs and more powerful models of code generation.

We base our work on GenVoca, a methodology and technology for generating product-lines using feature refinements. This paper shows how GenVoca ideas scale to product-families, something that has not been demonstrated previously. Further, we argue that our results are directly applicable to other models, such as *Aspect-Oriented Programming (AOP)* [20] and *Multi-Dimensional Separation of Concerns (MDSC)* [33, 23, 24], and thus are not GenVoca-specific. We explore this connection further in Related Work. We begin with a motivating example that illustrates the phenomenon of facets.

2. A Motivating Problem

An *Integrated Development Environment (IDE)* is a suite of applications (i.e., a product-family) that allow users to write, debug, visualize, and document programs. Among the programs, here called *tools*, of an IDE are a compiler, debugger, editor, formatter, and document generator (e.g, Javadoc). Figure 3a depicts some of these tools, each of which is implemented in a different package.

The problem we consider is generating IDE tools that all work on the same language dialect or *Domain-Specific Language (DSL)*. The use of DSLs have shown benefits in terms of understandability, maintainability, and extensibility in software design and development processes [11]. Providing IDE tools to support DSL program compilation, editing, debugging, and document generation is essential

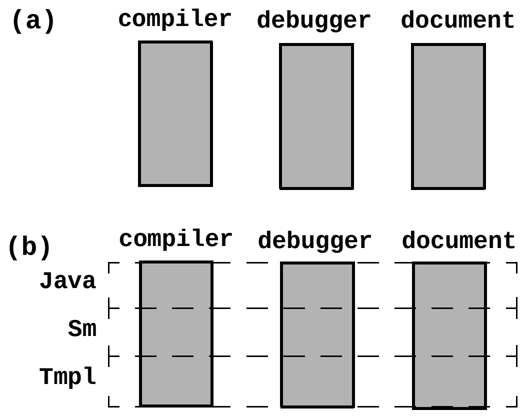


Figure 3. IDE Tools and Cross-Cutting Language Features

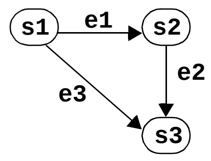
for the successful adoption of DSL technology. In particular, our work focuses on dialects of Java.

An example of a Java dialect is the one we are using to write fire-support simulators for the U.S. Army [6]. As a brief summary, fire support programs are a set of collaborating state machines. Figure 4a depicts a state machine of three states (**`s1`**, **`s2`**, **`s3`**) and three edges (**`e1`**, **`e2`**, **`e3`**), where an edge denotes a transition from one state to another. For example, edge **`e3`** begins at state **`s1`** and ends at state **`s3`**. Wide spectrum languages, like Java, are typically used to implement state machines. The resulting code, even when using the state machine design pattern [14], is often ugly, involving nested switch statements, large numbers of methods or classes. This places a burden on maintenance engineers because they must re-engineer the simple abstractions of state machines (e.g., Figure 4a) from the code in order to understand and modify it. In contrast, Figure 4b shows the specification of Figure 4a in our extended Java language. Highlighted are state declarations and edge declarations. We have found that state-machine-extended Java programs are about half the size of their pure-Java counterparts, and this in turn simplifies the writing, maintenance, and understanding of domain-specific programs. Similar benefits accrue when other extensions, such as templates, are added to Java.

In the future, we expect to work in other domains, each requiring their own specific extensions to Java. This means that we need to be able to construct IDE tools targeted for a particular Java dialect, or more generally, we need to define a product-line for a product-family of IDE tools. The novelty of our work is that we are using refinements (layers) as the unit of modularity.

Figure 3b revisits our IDE tools, but this time we expose the layers from which they were constructed. One layer, **`Java`**, encapsulates a cross-cut of the compiler, debugger, and document generation packages that is specific to the Java language. A second layer, **`Sm`**, encapsulates another cross-cut of these tools; the encapsulated code fragments

(a)



(b)

```
state_machine example {
   event_delivery receive_message(M m);
   no_transition { error( -1, m ); }
   otherwise_default { ignore_msg(m); }

   states s1, s2, s3;                          states

   edge e1 : s1 -> s2                          edges
   conditions !booltest() do
      { /* e1 action */ }

   edge e2 : s2 -> s3
   conditions booltest() do
      { /* e2 action */ }

   edge e3 : s1 -> s3
   conditions true  do
      { /* e3 action */ }

   // Java class data members and
   // methods from here

   boolean booltest() { ... }
   example() { current_state = start; }
}
```

Figure 4. State Machines in Extended Java

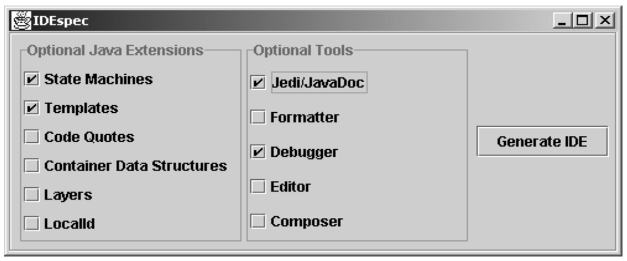
implement our state machine extension to Java. (That is, **Sm** extends the compiler tool to compile state machine specifications, it also extends the debugger so that it can debug state machine programs, etc.) A third layer, **Tmpl**, encapsulates the code fragments that implement our template extension to Java.

In principle, this is encouraging: layers (features) scale to product-families. That is, refinements scale to the encapsulation of fragments of multiple tools. Further, it appears that an IDE tool generator has a simple, declarative GUI front-end. Figure 5 suggests its basic outline: a client selects a set of optional language features and a set of tools (as not all might be needed), and by pressing the **Generate** button, the generator produces the requested set of IDE tools to work on the specified dialect of Java.

While the GUI is simple, the technology that underlies this generator is sophisticated. To understand how it works, we first review the GenVoca model and the Jakarta Tool Suite.

3. GenVoca

GenVoca is a design methodology for creating product-lines and building architecturally-extensible software —



Optional Java Extensions
- [✓] State Machines
- [✓] Templates
- [ ] Code Quotes
- [ ] Container Data Structures
- [ ] Layers
- [ ] LocalId

Optional Tools
- [✓] Jedi/JavaDoc
- [ ] Formatter
- [✓] Debugger
- [ ] Editor
- [ ] Composer

Figure 5. A GUI for an IDE-Tool Generator

i.e., software that is extensible via component additions and removals. GenVoca is an outgrowth of an old and practitioner-ignored methodology called *step-wise refinement* [12], which asserts that efficient programs can be created by revealing implementation details in a progressive manner. Traditional work on step-wise refinement has focussed on microscopic program refinements (e.g., **x+0** ⇒ **x**), for which one had to apply hundreds or thousands of refinements to yield admittedly small programs. While the approach is fundamental and industrial infrastructures are on the horizon [7, 27], GenVoca extends step-wise refinement by scaling refinements to a multi-class-cross-cut granularity, so that each refinement adds a feature to a program, and composing a few refinements yields an entire application.

3.1. Model Concepts

The central idea is programs are *values* and that refinements are *functions* that add features to programs. Consider the following constants that represent programs with different features:

```
f     // program with feature f
g     // program with feature g
```

A *refinement* is a function that takes a program as input and produces a refined (or feature-augmented) program as output:

```
i(x)  // adds feature i to program x
j(x)  // adds feature j to program x
```

A multi-featured application is specified by an *equation*, i.e., a named composition of functions. Different equations define a family of applications, such as:

```
app1 = i(f)    // app1 has features i & f
app2 = j(g)    // app2 has features j & g
app3 = i(j(f)) // app3 has features i, j, & f
```

Thus, by casually inspecting an equation, one can determine features of an application.

Note that there is a subtle but important confluence of ideas: a function represents both a feature *and* its implementation — there can be different functions that offer different implementations of the *same* feature:

```
k1(x)      // adds feature k with
           // implementation1 to x
k2(x)      // adds feature k with
           // implementation2 to x
```

When an application requires feature **k**, it is a problem of *equation optimization* to determine which implementation of **k** would be the best (e.g., provide the best performance)[2]. It is possible to automatically design software (i.e., produce an equation that optimizes some quantitative criteria) given a set of declarative constraints for a target application. An example of this kind of automated reasoning is in [5].

Although GenVoca constants and functions appear to be untyped, typing constraints do exist in the form of design rules. *Design rules* capture syntactic and semantic constraints that govern the legal composition of features [3]. It is not unusual that the selection of a feature will disable (or enable) the selection of other features. For this paper, design rules constrain the order in which features are composed. Details of their specification are beyond the scope of this paper and can be found in [3].

3.2. Model Implementation

Feature refinements are intimately related to *collaboration-based designs* [25, 30, 28]. A *collaboration* is a generic relationship among multiple classes. An individual class represents a particular role in a collaboration, and is a set of data members, methods, and method overrides that are needed to carry out this role. Because collaborations are defined largely in isolation of each other, they define features that are reusable, i.e., that can be used in the construction of many applications. A particular application is a composition of collaborations. Each class of an application plays one or more roles, where each role originates from a different collaboration.

A GenVoca constant is a set of classes. Figure 6 depicts a constant **i** with four classes ($\mathbf{a}_i$—$\mathbf{d}_i$). A GenVoca function is a set of classes and class refinements. A *class extension* is a subclass[3]: it encapsulates new data members, methods, and method overrides of its parent class. Figure 6 shows the result of applying function **j** to **i**: three classes are extended and another class is added. (That is, **j** encapsulates a cross-cut that includes classes $\mathbf{a}_j$, $\mathbf{c}_j$, and $\mathbf{d}_j$ that extend $\mathbf{a}_i$, $\mathbf{c}_i$ and $\mathbf{d}_i$ respectively, and adds class $\mathbf{e}_j$). Figure 6 also shows the application of function **k** to **j(i)**, resulting in two classes being extended. In general, a forest of inheritance hierarchies is created as layers are composed, and this forest grows progressively broader and deeper as the number of layers increase [4].

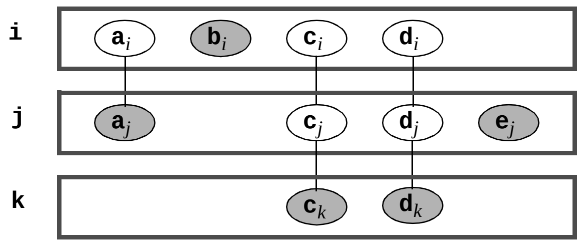


Figure 6. Implementing Refinements as Collaborations

Linear inheritance chains, called *refinement chains*, are common in this implementation method. The general rule is that only the bottom-most class of a refinement chain is instantiated, because this class implements all roles that were assigned to it. These classes are shaded in Figure 6.

For example, the bottom-most class of the **c** refinement chain plays the roles $\mathbf{c}_i$, $\mathbf{c}_j$, and $\mathbf{c}_k$ in the collaborations **i**, **j**, and **k** respectively.

Because GenVoca functions may be composed in arbitrary orders, class extensions can be implemented as mixins (modulo footnote 3). A *mixin* is a template: it is a class whose superclass is specified via a parameter. Mixins enable the order in which subclasses appear in a refinement chain to be permuted. More details on mixins and implementing collaborations as mixins are discussed elsewhere [30, 28, 13].

4. The Jakarta Tool Suite (JTS)

The *Jakarta Tool Suite (JTS)* is a suite of compiler-compiler tools for building families of language translators [4] that we used to implement our IDE tools. A language family is defined by a GenVoca model, which consists of a single constant — the base language — and functions that define optional extensions to the base. The family of Java dialects that can be synthesized by JTS is a GenVoca model, named **J,** consisting of the **Java** constant (representing the Java 1.4 language) and functions that add to Java embedded domain-specific languages for state machines (**Sm(x)**), container data structures (**P3(x)**), code fragments a la Lisp quote and unquote (**Ast(x)**), hygienic macros (**Gscope(x)**), and templates (**Tmpl(x)**), among others [4, 5, 29]:

```
J = { Java, Sm(x), P3(x), Ast(x), Gscope(x),
      Tmpl(x), ...}                          (1)
```

A particular dialect of Java is defined by an equation. The current dialect is called **Jak** (short for Jakarta), which is Java extended with state machines and templates:

2. Different equations represent different programs and equation optimization is over the space of semantically equivalent programs. This is identical to relational query optimization: a query is initially represented by a relational algebra expression, and this expression is optimized. Each expression represents a different, but semantically equivalent, query-evaluation program as the original expression.

3. More accurately, a class extension is a subclass that assumes the name of its parent class. This is different than typical subclassing, but is a simple way in which the contents of a class can be refined.

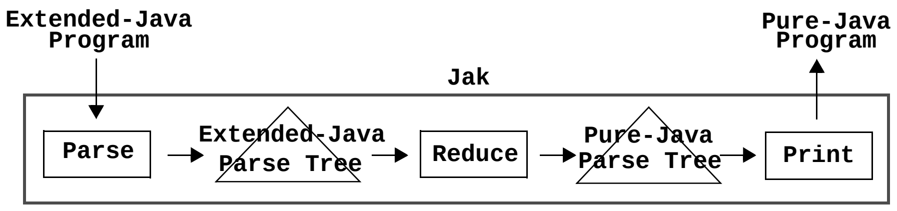


Figure 7. The Organization of the Jak Translator

```
Jak = Sm( Tmpl( Java ))                           (2)
```

The state machine and template features are independent of each other. As a consequence, the order in which **Sm** and **Tmpl** are composed doesn't matter. Thus, an equation equivalent to **(2)** is:

```
Jak = Tmpl( Sm( Java ))                           (3)
```

JTS converts such equations directly into a Java package using the ideas of Section 3 to implement a translator (preprocessor) for that language. The **Jak** preprocessor, like other JTS-produced preprocessors, translates an extended-Java program (with state machines and templates) into a program that represents its pure-Java counterpart. Figure 7 depicts its organization. An extended-Java program is parsed into an extended-Java parse tree. A reducer walks the tree, replacing each non-Java node or subtree with its pure-Java counterpart. The result is a pure-Java parse tree, which is then printed. The printed program is the Java translation of the extended-Java program. No matter what language extensions are added to Java, the organization of Figure 7 remains the same. This organization was inspired by Microsoft's IP [27].

Jak is only one of a number of IDE tools that must be customized to a particular language dialect. Another is a Javadoc-like tool that harvests comments from specific program constructs and displays them neatly on HTML pages. Obviously, Sun Microsystem's Javadoc [17] can't be used directly, as it only understands pure-Java programs (and documenting generated pure-Java programs typically isn't all that useful). So we created a language extensible version of Javadoc called *Jedi (Java Extensible DocumentatIon)*. **Jedi**, like **Jak**, is produced by JTS using a GenVoca model, called **D**. The lone constant is **JavaDoc**, which encapsulates the code that parsers, harvests, and documents comments in pure-Java programs. Functions of this model extend **JavaDoc** with the capabilities of producing HTML documentation for state machines (**SmDoc(x)**), templates (**TmplDoc(x)**), etc. In principle, the elements of models **J** and **D** are in one-to-one correspondence: each language extension in **J** has a corresponding documentation extension in **D**.[4]

```
D = { JavaDoc, SmDoc(x), TmplDoc(x), ... } (4)
```

A particular version of **Jedi** is specified as an equation:

```
Jedi = SmDoc( TmplDoc( JavaDoc ))          (5)
```

As before, the template and state machine layers of **Jedi** are independent, and thus can be composed in any order. Thus, an equation equivalent to **(5)** is:

```
Jedi = TmplDoc( SmDoc( JavaDoc ))          (6)
```

Figure 8 depicts the internal organization of **Jedi**. An extended-Java program is parsed into an extended-Java parse tree. A harvester walks the tree, harvesting comments prefacing particular language constructs (e.g., class, method, state machine declarations, etc.) and stores them in a comment repository. Finally, a doclet reads the contents of the comment repository, and formats harvested comments neatly on an HTML page, which users recognize as Javadoc-like output.

It is interesting to note that **Jak**, **Jedi**, and other IDE tools can be expressed directly by a single GenVoca model, **IDE_Model**, where different equations correspond to different tools. The primitives of this model are *tool features*. There is a lone constant **Parse**, which represents the parser for the given language dialect, and there are functions for reducing extended-Java constructs to pure-Java (**Reduce(x)**), for printing parse trees (**Print(x)**), for harvesting comments from parse trees (**Harvest(x)**), for

4. In practice, **J** and **D** need not be in correspondence. That is, there might be a language extension without a corresponding documentation extension, simply because that extension has yet to be built.

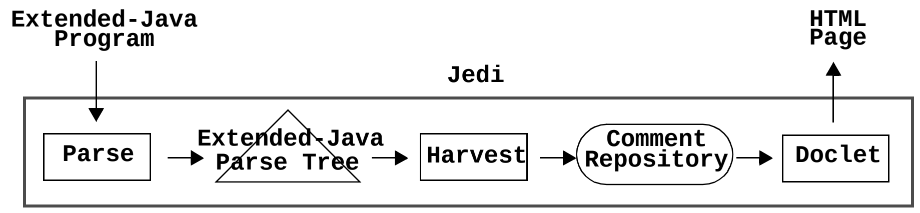


Figure 8. The Organization of the Jedi Translator

producing HTML documents from harvested comments (**Doclet(x)**), and so on.

```
IDE_Model = { Parse, Reduce(x), Print(x),
              Harvest(x), Doclet(x), ... }      (7)
```

Each IDE tool has an equation. The equations for **Jak** and **Jedi** are given below:

```
Jak  = Print( Reduce( Parse ))               (8)
Jedi = Doclet( Harvest( Parse ))             (9)
```

Even though the above equations look suspiciously like "functional" (e.g. Haskell) programs, they really do represent a composition of features that are implemented by the usual cross-cuts. Figure 9a shows the **Parse** layer to encapsulate a set of parser classes (only one class is shown), a set of parse tree node classes (again, only one is shown), and a **Main** class. The **Reduce** layer extends each parse tree node type with a reduction method (specific to that type), and extends the **Main** class with a call to reduce an extended-Java parse tree to a pure-Java parse tree. Finally, the **Print** layer extends each parse tree node type with a print method (specific to that node type) and extends the **Main** class with a call to print the reduced tree. Again, the terminals of the resulting refinement chains are the classes that are instantiated. Figure 9b shows the code added by each layer to the **Main** class.

> Note: the order in which tool features are composed is important. **Parse** must be first, followed by **Reduce** and then **Print**, or followed by **Harvest** and then **Doclet**. The reason is **Harvest** extends classes in **Parse**, and **Doclet** references methods in **Harvest**. The same applies to **Reduce** and **Print**. These constraints are examples of design rules.

Language extensibility is *not* part of the **IDE_Model**. In fact, astute readers may have noticed that our original descriptions of **Jak** and **Jedi** were based on GenVoca models of *language features*, and not *tool features*. Clearly these models are related, but how? Further, we know the **Jak** equations **(2)** and **(8)** must be equivalent, and so too the **Jedi** equations **(5)** and **(9)**. But how? An answer requires a closer look at the internals of these tools, which we do in the next section.

5. Gluons

Language features are orthogonal to tool features. This means that we can understand the modularity of **Jak** and **Jedi** in terms of matrices, where rows correspond to language features and columns correspond to tool features.

The matrix for **Jedi** is shown in Figure 10. Each matrix entry lists the name of a module that implements *a particular tool feature for a particular language feature*. For example, **Sharvest** is a module that implements the harvesting of comments from state machine specifications. **Jharvest** harvests comments from Java specifications. **Tdoclet** formats comments from template declarations on an HTML page. And so on. A composition of these modules implements **Jedi**.

	Doclet	Harvest	Parse
Java	Jdoclet	Jharvest	Jparse
Sm	Sdoclet	Sharvest	Sparse
Tmpl	Tdoclet	Tharvest	Tparse

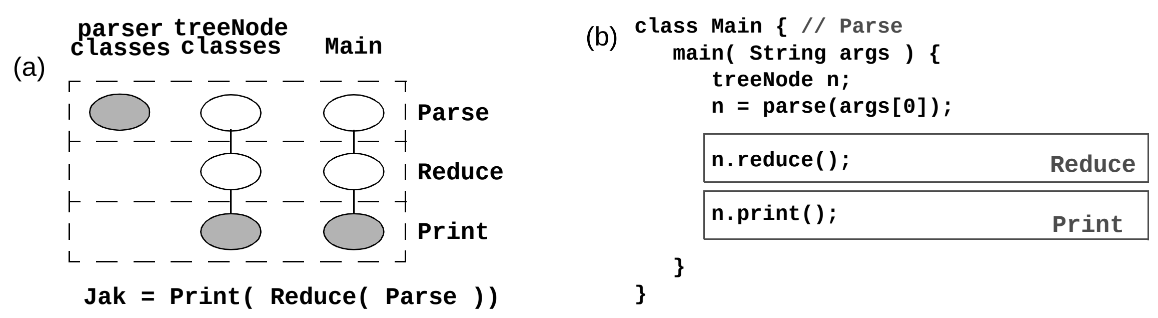
Figure 10. Jedi Matrix

The matrix for Jak is shown in Figure 11, and has a similar interpretation. There is a difference: there are no **Sm** and **Tmpl** row entries for the **Print** column. The reason is simple: consider the interpretation of **Sreduce**: it is a module that transforms parse trees on state machines into parse trees of pure Java. The **Jprint** module prints parse trees of pure Java. So once the **Sreduce** module performs its task, the **Jprint** module is invoked. Thus there is no need for a module that prints state machine parse trees. The same argument applies for templates. Once again, a composition of these modules implements the **Jak** tool.

	Print	Reduce	Parse
Java	Jprint	Jreduce	Jparse
Sm	–	Sreduce	Sparse
Tmpl	–	Treduce	Tparse

Figure 11. Jak Matrix

These matrices provide the first indication of facets. Let us call matrix entries *gluons* and consider the **Jedi** matrix of



```
class Main { // Parse
   main( String args ) {
      treeNode n;
      n = parse(args[0]);

      n.reduce();

      n.print();

   }
}
```

Figure 9. Cross-Cuts of Tool Features

Figure 10. Each row represents a language feature; its implementation is a composition of the gluons in that row. The **Java** language feature, for example, is defined by a composition of the **Jparse**, **Jharvest**, and **Jdoclet** gluons. The same for other rows.

By the same reasoning, each tool feature is represented by a column and is implemented by a composition of gluons in that column. For example, the **Harvest** tool feature is a composition of the **Jharvest**, **Sharvest**, and **Tharvest** gluons. The same for other columns.

Thus, if layers are rows of gluons, then facets are columns of gluons — columns cross-cut every row. Similarly, if layers are columns of gluons, then facets are rows of gluons — rows cross-cut every column. Thus, a facet is simply a feature along a dimension, and the implementation of a facet cross-cuts features of other dimensions.

Two questions remain. First, what are gluons? Very simply, they are elementary layers (refinements) that implement the intersection of pair of orthogonal features. Or more accurately, a gluon implements a *feature of a feature* or a *building block* of both a language feature and tool feature. A gluon is a module that encapsulates any number of classes and class extensions, and has straightforward implementation as a set of classes and mixins. Thus, we can represent each gluon as a GenVoca constant or function.

When we create the matrices of Figure 10 and Figure 11, we are decomposing a composite language feature (layer) or tool feature (layer) into more primitive layers — in essence, separating their concerns. The theoretical justification is simple: any function **F** can be the result of composing more primitive functions $F_1 \ldots F_n$, and any constant **C** could be the result of composing a more primitive constant C' with one or more functions $F_1' \ldots F_n'$:

$$F(x) = F_1(\ F_2(\ \ldots\ F_n(x)\ \ldots\ ))$$
$$C\ \ = F_1'(\ F_2'(\ \ldots\ F_n'(\ C'\ )\ldots))$$

Decomposing software is modeled by decomposing equations.

Second, we want to represent **Jak** and **Jedi** as equations that are compositions of gluons. Equations for **Jak** and **Jedi** are:

```
Jak = Jprint( Treduce( Sreduce( Jreduce(
     Tparse( Sparse( Jparse ))))))        (10)
Jedi = Tdoclet( Tharvest( Tparse(
     Sdoclet( Jdoclet( Sharvest(
     Jharvest( Sparse( Jparse )))))))))   (11)
```

These equations are *much* more complex than those of previous sections. Two questions immediately arise: (a) are they *correct* — are they legal compositions of gluons? and (b) are they *consistent* — do they represent tools that work on the same language dialect? Existing design rule checking algorithms can validate these equations [3], but there are no algorithms to check for *consistency.* In fact, without the techniques presented in the next section, it would take some time to write such equations manually and verify that they are consistent. We would expect the consistency problem to be much more difficult for larger sets of tools and more complex language dialects. Hence, automated support is required to write these equations and to ensure their consistency: we need a model of gluons.

6. Origami: A Model of Gluons

The notation that we have used prior to this section is consistent with previous work on GenVoca. However, the usual "functional" notation becomes cumbersome as equations become complicated. So we make a cosmetic switch in notation to simplify our upcoming discussions. Without loss of generality, instead of writing **A = B(C(D))** we write **A = B o C o D**, where **o** is the (function) composition operator.

GenVoca models are inherently one-dimensional; they are sets of constants and functions. In contrast, models of gluons are 2-dimensional — and generally n-dimensional — and need to be treated accordingly.[5] Consider the matrix of Figure 12, called an *Origami matrix*, where rows denote language features and columns are tool features. Elements of this matrix are gluons.

Adding new entries to this matrix is easy. When a new row is added, a gluon must be supplied for every existing column. For example, to add the container data structure (**Ds**)

5. All examples of gluons that are known to us can be expressed in 2-dimensions. We would expect examples yet to be discovered to have higher dimensionality. A 3-dimensional example would expose "facets of facets", and so on. Thus our model scales.

	Doclet	Harvest	Parse	Reduce	Print	...
Java	Jdoclet	Jharvest	Jparse	Jreduce	Jprint	...
Sm	Sdoclet	Sharvest	Sparse	Sreduce	-	...
Tmpl	Tdoclet	Tharvest	Tparse	Treduce	-	...
Ds	Ddoclet	Dharvest	Dparse	Dreduce	-	...
...	...	...	...	...	...	...

Figure 12. An Origami Matrix

language feature, we would have to add **Dparse** (a parser for container DSL specifications), **Dreduce** (reduction methods to transform container specification parse trees to Java parse trees), **Dharvest** (a harvester of comments on container specifications), and **Ddoclet** (a doclet that formats container comments). Some entries (such as the entry for the **Print** column) are “empty” because no code needs to be written to implement that functionality. In such cases, the identity function (denoted by “**-**”) is supplied.

Symmetrically, when a new column is added, a gluon must be supplied for every existing row. To add a new doclet that produces, say Word documents, we would add **Jword** (a doclet that formats Java comments in Word), **Sword** (a doclet that formats state machine comments in Word), **Tword** (a doclet that formats template comments in Word), and so on. Again, if no code needs to be written for a particular entry, an identity function is supplied.

An application (expression) is created by *folding* an Origami matrix (hence its name). Rows are folded together by composing the corresponding gluons in each column. Columns are folded together by composing the corresponding gluons in each row. Folding continues until a 1×1 matrix is produced; the entry of this matrix is the desired expression. (Unlike true Origami, rows and columns to be folded need not be adjacent. For our examples, we have arranged this matrix so that they are).

Rows and columns cannot be chosen at random for folding. Rows (columns) must be composed in design rule order. That is, if we are folding tool features, we must begin with the **Parse** column, and then fold/compose the **Harvest** column, and finally the **Doclet** column, just as design rules prescribe for the **IDE_Model**. Similarly, if we are folding language features, we must begin with the **Java** row, and then fold the **Sm** row and **Tmpl** rows in any order, as prescribed by the language feature models **J** and **D**. The reason for this is that language features and tool features are orthogonal.[6]

As an illustration, suppose we want to create an equation for current version of **Jedi**. We project this matrix of unnecessary rows and columns, leaving the rows for **Java**, **Sm**, and **Tmpl**, and the columns **Parse**, **Harvest**, and **Doclet** yielding Figure 13a. (Note that there can be different kinds of doclets — HTML, Word, etc. So part of this projection is selecting the appropriate tool features).

Figure 13b shows the result of composing the **Java** row with the **Sm** row. Figure 13c-d shows the result of composing the **Harvest** column with the **Parse** column, and this result with the **Doclet** column. A matrix of two rows and one column results. The final fold merges the remaining two rows to yield the expression of equation **(11)**. We leave it as an exercise for readers to discover the folding of equation **(10)**.

Other constraints may preclude certain foldings, but this is the essential idea. In the next section, we show how we can use Origami to produce sets of language-dialect consistent equations.

7. An Application of Origami

Recall the GUI for the IDE generator of Figure 5: users select a set of optional language features and a set of tools, and the generator produces this set of tools for the speci-

6. In effect, gluons make our model more powerful without additional complexity as the existing row and column rules can be used without modification.

(a)	Doclet	Harvest	Parse
Java	Jdoclet	Jharvest	Jparse
Sm	Sdoclet	Sharvest	Sparse
Tmpl	Tdoclet	Tharvest	Tparse

(b)	Doclet	Harvest	Parse
Sm o Java	Sdoclet o Jdoclet	Sharvest o Jharvest	Sparse o Jparse
Tmpl	Tdoclet	Tharvest	Tparse

(c)	Doclet	Harvest o Parse
Sm o Java	Sdoclet o Jdoclet	Sharvest o Jharvest o Sparse o Jparse
Tmpl	Tdoclet	Tharvest o Tparse

(d)	Doclet o Harvest o Parse
Sm o Java	Sdoclet o Jdoclet o Sharvest o Jharvest o Sparse o Jparse
Tmpl	Tdoclet o Tharvest o Tparse

(e)	Doclet o Harvest o Parse
Tmpl o Sm o Java	Tdoclet o Tharvest o Tparse o Sdoclet o Jdoclet o Sharvest o Jharvest o Sparse o Jparse

Figure 13. Folding an Origami Matrix

fied language dialect.[7] To see how the generator works, we begin with the Origami matrix of Figure 12 and eliminated all language feature rows that were not selected. Figure 14 shows this matrix for the current **`Jak`**/**`Jedi`** dialect.

Rows are folded in design rule order (i.e., **`Tmpl o Sm o Java`**). In general, our generator simply uses design rules to hard-code this ordering. The result is a $1 \times n$ matrix (i.e. a row) in Figure 15. Note the row's semantics. First, each column defines an equation for a tool feature:

```
Doclet  = Tdoclet o Sdoclet o Jdoclet
Harvest = Tharvest o Sharvest o Jharvest
Parse   = Tparse o Sparse o Jparse
...
```

That is, the **`Doclet`** equation is a composition of gluons that builds a doclet layer for the Java language that has been extended by state machines and templates. The **`Harvest`** equation defines a harvest layer for the Java language that has been extended by state machines and templates, and so on. Thus, by folding rows in design-rule order, *we have produced a set of equations for tool features that are consistent with respect to a particular language dialect.*

Second, the row itself is exactly the set of tool features that comprise the **`IDE_Model`**. Since we know the **`IDE_Model`** equations for each tool (e.g., **`(8)`**,**`(9)`**), we use these equations and plug in the generated definitions for their tool features. Thus, for each GUI-selected tool, we evaluate its equation, and send it to a generator to produce the Java package for that tool. In this way, our IDE generator produces language-dialect-consistent tools from a simple declarative specification.

8. Implementation

We currently have five tools that are language-dialect sensitive: **`Jedi`**, **`Jak`**, **`Mixin`** and **`Jampack`** (two different tools that compose **`Jak`** specifications), and **`UnMixin`** (a tool that propagates changes made manually in composed code back to their defining "gluon" layers). There are nine language features and nine tool features that span these tools. (We support several template features as well as additional language features, such as hygienic macros). Table 1 lists for each tool its size in Java LOC, and the number of gluons that define its equation. Note that equations provide *very* compact specifications of these tools.

Just with this set of tools (for which we expect many more), we are generating over 100K LOC. Without Origami, our tool equations are seemingly randomly-ordered compositions of gluons that are difficult to understand and update. Origami imposes a regularity in equation organization and layer modularization that enables us to generate product-families from simple specifications. Just as important, it helps us control the complexity of feature-refinement-based representations of product-families.

Tool	# of Gluons	Size in Java LOC
Jak	14	26K
Jedi	31	32K
Mixin	23	24K
JamPack	22	26K
UnMixin	20	23K

Table 1. Size of Generated IDE Tools

9. Relevance to Other Technologies

There are many non-GenVoca examples of Origami. One possibility is the internationalization of programs made during Windows OS installations. By selecting a particular language (or dialect), the GUIs of different Windows programs are modified to present commands in that language. Origami also has relationships to component-based software design.

Microsoft's *Component Object Model (COM)*, Sun's *Enterprise Java Beans (EJB)*, and CORBA are conventional component models [26] that deal with *interface-based programming* — clients program to standardized

7. We assume the set of language features is consistent. Design rule checking algorithms can be used to check consistency.

	Doclet	Harvest	Parse	Reduce	Print	...
Java	Jdoclet	Jharvest	Jparse	Jreduce	Jprint	...
Sm	Sdoclet	Sharvest	Sparse	Sreduce	-	...
Tmpl	Tdoclet	Tharvest	Tparse	Treduce	-	...

Figure 14. A Row-Projected Matrix

	Doclet	Harvest	Parse	Reduce	Print	...
Tmpl o	Tdoclet o	Tharvest o	Tparse o	Treduce o	Jprint	...
Sm o	Sdoclet o	Sharvest o	Sparse o	Sreduce o		
Java	Jdoclet	Jharvest	Jparse	Jreduce		

Figure 15. A Row-Folded Matrix

interfaces and components implement these interfaces [32]. This makes it easy to swap out one interface implementation (component) with another, say, for purposes of bug fixes, improved performance, or trying alternative implementations. Variations of interface-based programming are found in design patterns (e.g., OO decorators) and in common OO designs (e.g., frameworks) [14].

Not long ago, GenVoca was presented as example of interface-based programming.[8] The key design issue was choosing the methods of an interface. Those that were included were *fundamental* to the abstraction of that interface; those that were excluded were dismissed as not fundamental.

The problem that we and other engineers have noticed with interface-based designs is that they are brittle. We observed that the set of methods that we designated as fundamental was subjective — they were sufficient for our current needs. Over time, we longed for other methods to be included, and periodically we would indeed extend the set of methods in our interfaces. However, when new methods are added to a standardized interface, all components that export that interface had to be (manually) updated. After an extension, we would be happy for a while until we discovered a new set of methods that needed to be added, and the update cycle would repeat.

The problem with this, of course, is that we couldn't subsequently customize our interfaces or our components. It was simply too much work to eliminate unneeded groups of methods from interfaces and components. The impact of interface extensions is negative: interfaces become fat and components suffer code bloat. Other techniques have been developed to address this problem, but they too have limitations. COM, for example, requires that a new interface be published rather than changing an existing interface. While this works, it still requires a manually-introduced extension to each component that is to implement that new interface. The visitor design pattern allows almost arbitrary method extensions to existing components [14]. Access to private data members and methods of components is precluded to visitors, and this can be problematic. Also, it is useful for extensions to add new data members to components, and this too is problematic using visitors.

It is easy to recognize the concept of gluons and Origami in this situation. Each row represents either a standardized interface or a component that implements such an interface. Columns represent semantically cohesive groups of methods — features — where one column defines a "core" set of methods and other columns represent optional additional extensions to this set. Matrix (column) folding corresponds to the construction of interfaces and components that are customized for a desired set of interface extensions with their implementing components.

8. Which actually it still is. Layers have interfaces, although in recent papers including this one, this "feature" of layer implementation has been down-played. See [8, 22].

The need for Origami arises because abstractions change over time. Changes tend to be incremental and optional. That is, abstractions change by incremental leaps in understanding, and these leaps are needed for building specialized classes of applications. The contribution of this paper is a general model and a set of techniques that allow us to evolve both conventional components and implementations of feature refinements statically in an automatic and declaratively-specified way. Such flexibility is useful in generating software. For situations dealing with third-party components, where extensibility without recompilation is a major goal, it might not work as well. However, there is *no* requirement that feature refinements must be composed statically; they can be composed dynamically as well [31]. Unfortunately, dynamically-composable refinements are not as well-understood as statically-composable refinements.

10. Related Work

The idea that features have features is well-established in the product-line community. Feature diagrams, which are typically hierarchies of features, i.e., parent features are defined to have aggregate sets of child features, was first introduced in the FODA methodology [18] and has been improved by others [10]. Our contribution shows how the idea of features-of-features translates into product-family models based on feature refinements.

As mentioned earlier, there are other models of program development that are very similar to GenVoca, the most prominent of which are **`AspectJ`** and **`Hyper/J`**. **`AspectJ`** [1] offers two flavors of cross-cutting implementations: static and dynamic. Static cross-cuts are almost identical to GenVoca layers: they can add new data members and new methods to existing classes. Dynamic cross-cuts, where explicit pointcut-advice pairs are defined, can emulate the refinement (overriding) of methods offered by inheritance. What aspects *cannot* currently represent is the addition of new classes; in GenVoca terms, aspects only extend existing classes. (At least, we have been unable to add classes in aspect definitions that can be subsequently refined). With simple work-arounds, we have implemented GenVoca generators using **`AspectJ`**. These preliminary results suggest that compositions of layers can be modeled as compositions of aspects. Therefore, we believe that the Origami example in this paper could be implemented using **`AspectJ`** and thus our results are relevant to AOP in that they show how aspects can scale to product-families.

Admittedly, **`AspectJ`** can do more than just implement layers (modulo our comments above), and in fact, we are focussing on the least novel part of **`AspectJ`**. But it is also the case that what we and others have been able to do with GenVoca generators has never been done in AOP. Our work provides an opportunity to enhance AOP's appeal from a novel direction.

Our work is more closely related to *Multi-Dimensional Separation of Concerns (MDSC)*. MDSC is the idea that modularity relationships can be understood in terms of an n-dimensional space, called a *hyperspace*, of units [33, 23, 24]. A *unit* can be primitive (such as an individual method or variable) or compound (e.g., a class or package). Each dimension is associated with a set of similar concerns, such as a set of classes or a set of features; different values along a dimension are different members of this set (e.g., $\text{class}_1 \ldots \text{class}_n$ or $\text{feature}_1 \ldots \text{feature}_n$). A *hyperslice* is the set of units that pertain to a concern; it is an (n-1)-dimensional space where one coordinate value (e.g., a concern) is fixed. A *hypermodule* is a set of hyperslices and a set of integration relationships that dictate how the units of hyperslices are to be integrated or composed to form a program.

`Hyper/J` is the flagship tool for MDSC [34]. We have used **`Hyper/J`** to implement GenVoca product-line models. GenVoca layers have direct implementations as hyperslices, and layer compositions are hyperslice compositions. Again, we believe that the Origami model and its results are directly applicable to **`Hyper/J`**. Origami is a 2-dimensional example of MDSC, where both dimensions are features and units are gluons. Further, the strength of MDSC models is that they do not impose fixed modularization hierarchies, and this flexibility is present in Origami matrices. As with **`AspectJ`**, **`Hyper/J`** can do more than just compose hyperslices. Our contribution is that we can provide sophisticated examples of product-lines and product-families to **`Hyper/J`** researchers.

In summary, GenVoca, **`Hyper/J`** (MDSC), and **`AspectJ`** (AOP) have substantial overlaps. What distinguishes GenVoca and Origami is an algebra for organizing features into programs.

Other related work deals with tool integration [31]. Cross-cuts are problematic when new features can impact every product in a product-family. However, instead of designing a system to easily handle features, [31] explored how a product family can be designed in such a way that new features can be added by modifying a single class — a design that eliminates cross-cuts. The advantage is that it can be applied to legacy software and that it ensures that existing tools will be able to work with new additions to the program family without recompilation. The authors emphasize that in their design, the cost of evolutionary change is proportional to its apparent size in specification. The disadvantage is that this technique only applies to some features, so in fact their approach is complimentary to our own.

11. Conclusions

Features have proven their value in raising the level of abstraction in modularity in building and customizing individual programs. The question is: do features scale to larger program organizations, such as program families? We showed that they do in the context of GenVoca generators, which has not been done before. We discovered that features themselves have internal structures — features of features — which we called gluons. Gluons are arranged and composed in very regular ways, so that compositions of gluons yields both familiar and formerly "atomic" features, as well as an interesting and what we now believe is a common phenomena of facets. Facets cross-cut features and compositions of them yield fully-formed features. In essence, we have identified a new class of composition relationships among features that were not previously known.

There is anecdotal evidence that supports our work. Engineers have repeated the observation that there is something about program scale that introduces complexity one doesn't find in small programs. Our work reveals one reason: there are relationships and constraints that exist among gluons when building program families. If there is no way (or only ad hoc ways) of expressing and satisfying these constraints, it is no wonder why scaling programs introduces complexity. At least now we have a way to express and reason about such constraints. Undoubtedly there are even more relationships to be discovered.

The key to our success is *how* we represent and manipulate these relationships. Using GenVoca formulations allows us to capture these regularity relationships as matrices of functions and constants that can be folded into equations. That is, we can *reason* about software designs as equations. We explained that our results are not GenVoca-specific, in particular, how Origami has direct relationships to AOP and MDSC models. We believe Origami is important, because others will encounter it as feature refinement models scale to produce more complex systems.

Acknowledgements. We thank Jack Sarvela for pointing out the relationship of Origami to internationalization customizations of Windows programs. We also thank anonymous referees for helping us to better clarify our presentation.

12. References

[1] AspectJ. Programming Guide. **`http://aspectj.org/doc/proguide`**

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

[3] D. Batory and B.J. Geraci, “Composition Validation and Subjectivity in GenVoca Generators”, *IEEE Transactions on Software Engineering*, Feb. 1997, 67-82.

[4] D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages”, *5th Int. Conf. on Software Reuse*, Victoria, Canada, June 1998.

[5] D. Batory, G. Chen, E. Robertson, and T. Wang, “Design Wizards and Visual Programming Environments for GenVoca Generators”, *IEEE Trans. Software Engineering*, May 2000.

[6] D. Batory, C. Johnson, R. MacDonald, and D. von Heeder, “Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study”, to appear in *ACM TOSEM*.

[7] I. Baxter, “Design Maintenance Systems”, *CACM*, April 1992.

[8] R. Cardone, A. Brown, S. McDirmid, and C. Lin, “Using Mixins to Build Flexible Widgets”, *AOSD 2002*.

[9] K. Czarnecki and U.W Eisnecker. *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[10] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker, “Generative Programming for Embedded Software: An Industrial Experience Report”, GCSE/SAIG 2002.

[11] A. van Deursen and P. Klint, “Little Languages: Little Maintenance?”, *SIGPLAN Workshop on Domain-Specific Languages*, 1997.

[12] E.W.Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[13] Flatt, M., Krishnamurthi, S., and Felleisen, M. “Classes and Mixins”. *ACM Principles of Programming Languages*, San Diego, California, 1998, 171-183.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[15] M. Griss, “Implementing Product-Line Features by Composing Component Aspects”, *First International Software Product-Line Conference*, Denver, August 2000.

[16] W. Harrison and H. Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)”, *OOPSLA 1993*, 411-427.

[17] Javadoc — The Java API Documentation Generator. Sun Microsystems, **`http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/javadoc.html`**

[18] K.C. Kang, et al., Feature-Oriented Domain Analysis Feasibility Study, SEI 1990.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, *ECOOP 97*, 220-242.

[20] G Kiczales, E. Hilsdale, J. Hugunin, M. Kirsten, J. Palm, and W.G. Griswold. “An overview of AspectJ”. *ECOOP 2001*.

[21] M. Mezini and K. Lieberherr, “Adaptive Plug-and-Play Components for Evolutionary Software Development”, *OOPSLA 1998*, 97-116.

[22] S. McDirmid, M. Flatt, and W.C. Hsieh, “Jiazzi: new-Age Components for Old-Fashioned Java”, *OOPSLA 2001*.

[23] H. Ossher and P. Tarr. “Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software.” *CACM* October 2001.

[24] H. Ossher and P. Tarr, “Multi-dimensional separation of concerns and the Hyperspace approach.” In *Software Architectures and Component Technology* (M. Aksit, ed.), 293-323, Kluwer, 2002.

[25] T. Reenskaug, et al., “OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems”, *Journal of Object-Oriented Programming*, 5(6): October 1992, 27-41.

[26] R. Sessions, *COM+ and the Battle for the Middle Tier*, Wiley Computer Publishing, 2000.

[27] C. Simonyi, “The Death of Computer Languages, the Birth of Intentional Programming”, *NATO Science Committee Conference*, 1995.

[28] Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”. *ECOOP*, July 1998.

[29] Y. Smaragdakis and D. Batory, “Scoping Constructs for Program Generators”. *Generative and Component-Based Software Engineering (GCSE)*, September 1999.

[30] Y. Smaragdakis and D. Batory, “Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs”, to appear *ACM TOSEM*.

[31] K.J. Sullivan and Notkin, D., ``Reconciling Environment Integration and Software Evolution," *ACM TOSEM* July 1992.

[32] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1997.

[33] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton, Jr., “N Degrees of Separation: Multi-Dimensional Separation of Concerns”, *ICSE 1999*.

[34] P Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM Corporation, 2001. **`http://www.research.ibm.com/hyperspace`**.

[35] M. Van Hilst and D. Notkin, “Using Role Components to Implement Collaboration-Based Designs”, *OOPSLA 1996*, 359-369.