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. 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 synthesized 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* — 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) [18].

Feature refinement is a very general concept and many different implementations of it have been proposed, each with different names, capabilities, and limitations: layers [3], features [21], collaborations [28][39][24], subjects [19], aspects [22] and concerns [36]. 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 the stratification of **c1**—**c3** in Figure 1. As refinements, layers, and features are so closely related, their terms are used interchangeably. In general, layers are modular, albeit unconventional, building blocks of programs.

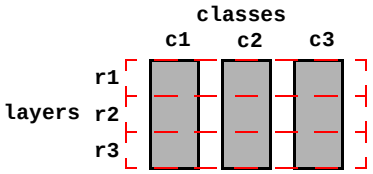


Figure 1: Classes and Refinements (Layers)

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.

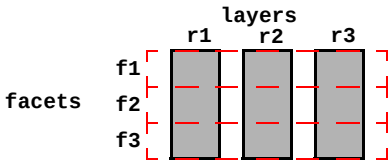


Figure 2: Facets

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 [11]. 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 also present our implementation, tools, and experiences with Origami.

Our work is based 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)* [23] and *Multi-Dimensional Separation of Concerns (MDSC)* [36][26][27], 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.

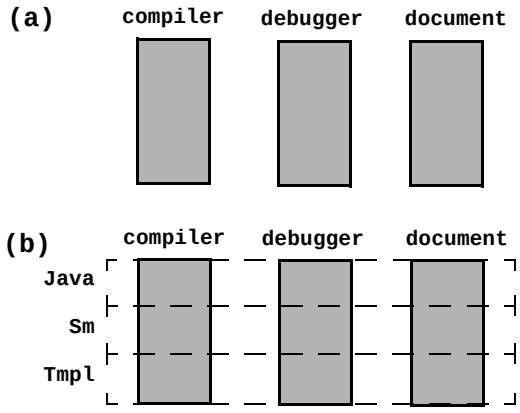
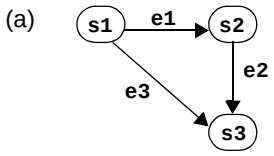


Figure 3: IDE Tools and Cross-Cutting Language Features

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 [13]. Providing IDE tools to support DSL program compilation, editing, debugging, and document generation is essential 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 [7]. 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



(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

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 [16], 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 sec-

ond layer, **Sm**, encapsulates another cross-cut of these tools; the encapsulated code fragments 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.

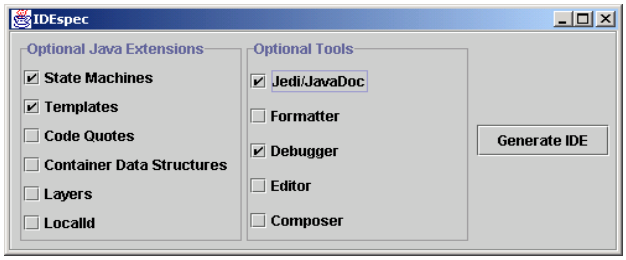


Figure 5: A GUI for an IDE-Tool Generator

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 — 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* [14], 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+1** ⇒ **inc(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 [9][30], 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 *constants* and 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 *GenVoca model* is a set of constants and functions. A multi-featured application is an *equation* that is a named composition of a model's constants and 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 the 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 *optimization* to determine which implementation of **k** is best (e.g., provides the best performance)[1]. 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 [6].

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 [4]. It is not unusual that the selection of a feature disables (or enables) the selection of other features [12]. 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 [4].

3.2 Model Implementation

Feature refinements are intimately related to *collaboration-based designs* [28][31][33]. 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.

1. 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.

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 (**a**$_i$—**d**$_i$). A GenVoca function is a set of classes and class extensions. A *class extension* is a subclass: it encapsulates new data members, methods, and method overrides of its parent class.[2] Figure 6 shows the result of applying function **j** to **i**: classes **a**, **c**, and **d** are extended (**a**$_j$, **c**$_j$, **d**$_j$), and class **e** is added (**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 [5].

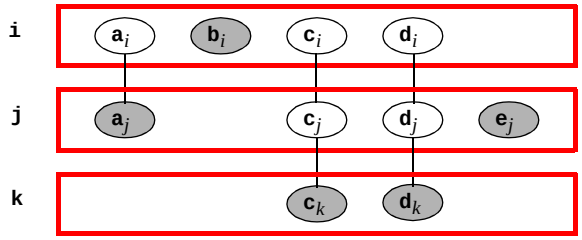


Figure 6: Implementing Refinements as Collaborations

Linear inheritance chains, called *refinement chains*, are common in this implementation method. The 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 **c**$_i$, **c**$_j$, and **c**$_k$ in the collaborations **i**, **j**, and **k** respectively.

Because GenVoca functions can be composed in arbitrary orders, class extensions are implemented as mixins. 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 [31][15][33].

2. 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.

4 The Jakarta Tool Suite (JTS)

The *Jakarta Tool Suite (JTS)* is a suite of compiler-compiler tools that we used to implement our IDE tools [5]. A family of translators 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 translators of Java dialects 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 ([5][6][32]):

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

A translator for particular dialect of Java is defined by an equation. The current dialect of Java is called **Jak** (short for Jakarta), which is Java extended with state machines and templates. Its translator, **j2j** (short for Jak-to-Java), is:

```
j2j = 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:

```
j2j = Tmpl( Sm( Java ))                  (3)
```

JTS converts such equations directly into a Java package using the ideas of Section 3 to implement a translator (preprocessor). **j2j**, 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 internal 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 [30].

j2j 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** [20] 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*

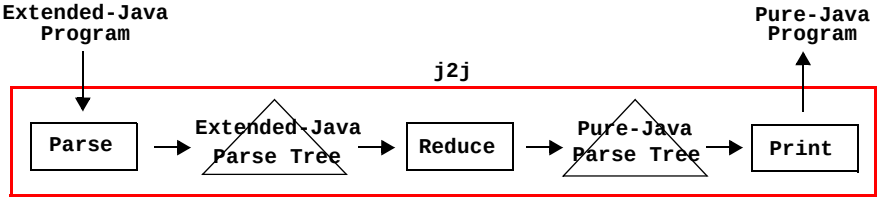


Figure 7: The Organization of the j2j Translator

Extensible DocumentatIon). **Jedi**, like **j2j**, 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: for each language extension in **J** there is a corresponding documentation extension in **D**.[3]

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

A particular version of **Jedi** is specified as an equation, e.g.,

```
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., interface declarations, class declarations, method declarations, and state machine declarations) 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 **j2j**, **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 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 **j2j** and **Jedi** are:

```
j2j  = 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 cross-cuts. Figure 9a shows that the **Parse** layer encapsulates 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.

3. 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.

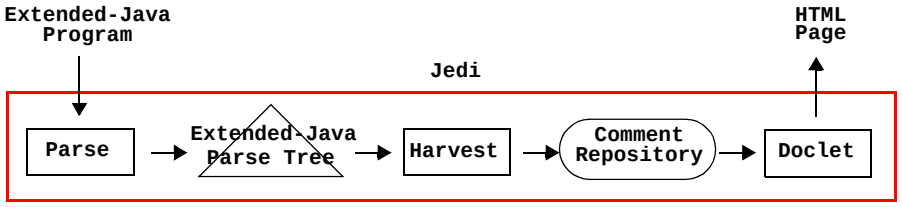


Figure 8: The Organization of the Jedi Translator

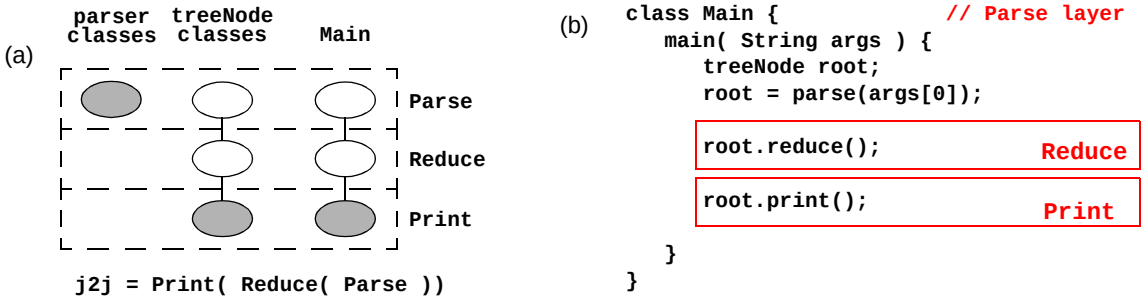


Figure 9: Cross-Cuts of Tool Features

Language extensibility is *not* part of the **IDE_Model**. In fact, astute readers may have noticed that our original descriptions of **j2j** 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 **j2j** 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 **j2j** 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 **j2j** 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 **j2j** tool.

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

Figure 11: j2j Matrix

These matrices provide the first indication of facets. Let us call matrix entries *gluons* and consider the **Jedi** matrix of 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 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 a language feature and a tool feature. A gluon is a module that encapsulates any number of classes and class extensions, and has straightforward implementation as a layer. 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** can 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 **j2j** and **Jedi** as equations that are compositions of gluons. Equations for **j2j** and **Jedi** are:

```
j2j = 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 [4], 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 worse 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. 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**) 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, the 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 the 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.

To illustrate, suppose we want to create an equation for **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 specified language dialect.[4] To see how the generator works, we begin with the Origami matrix of Figure 12 and eliminate all language feature rows that were not selected. Figure 14 shows this matrix for the current **Jak** 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. Each column defines an equation for a tool feature:

4. 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	-	...
Ds	Ddoclet	Dharvest	Dparse	Dreduce	-	...
...	...	...	...	...	...	...

Figure 12: An Origami Matrix

(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

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

That is, the **Doclet** equation is the 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*.

The row of Figure 15 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 expressions 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 and Experience

In this section, we describe our implementation, tools and experiences with Origami.

8.1 The AHEAD Origami Matrix

AHEAD is the successor to GenVoca [8]. Initially all of AHEAD tools were built using JTS; AHEAD tools have since been bootstrapped. Both JTS and AHEAD synthesized AHEAD tools by folding a 3-dimensional Origami matrix (Figure 16). The matrix itself is sparse, where only one "plane" in the third dimension is non-empty. The dimensions are (Language Features $\times$ Tool Features $\times$ Language Features).

The AHEAD matrix is represented as a pair of 2-dimensional matrices. The frontal matrix is called **Tools**; the horizontal matrix is **Ast**. We will explain the **Tools** matrix first in Section 8.2 and the **Ast** matrix later in Section 8.3.3.

The units along the Tool Features dimension are listed in Table 1. All tools that can be synthesized from this matrix share the same

	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 Sm o Java	Tdoclet o Sdoclet o Jdoclet	Tharvest o Sharvest o Jharvest	Tparse o Sparse o Jparse	Treduce o Sreduce o Jreduce	Jprint	...

Figure 15: A Row-Folded Matrix

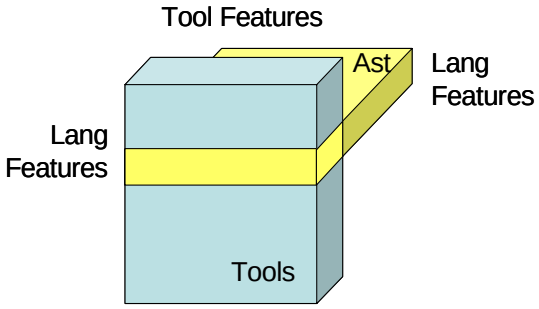


Figure 16: The AHEAD Origami Matrix

grammar and parser, which is denoted by **Base**. The remaining units of Table 1 graft on tool-specific semantics to the parser. Thus, AHEAD tools operate on **Jak** files yet have different functionality (e.g., some compose **Jak** files (**jampack**, **mixin**), another translates **Jak** files to Java (**j2j**), and another propagates changes in composed specifications back to the uncomposed specifications (**unmixin**)).[5]

Unit	Description
Base	parser/grammar shared by all tools
j2j	Jak-to-Java translation tool
JamPack	compressed composition tool
Mixin	uncompressed composition tool
UnMixin	update propagation tool
mmatrix	reflection tool

Table 1. Units of the Tool Features Dimension

The units along both Feature Language dimensions are listed in Table 2. The base language is **Java**; other units extend Java in some way, typically by adding new grammar rules. The extensions include an embedded DSL for state machines (**SmDsl**), extensions for metaprogramming (**AstDsl**), forms of hygienic macros (**LocalIdDsl**, **Gscope**), and refinements (**ComposeIntDsl**, **ComposeSmDsl**, **ComposeClassDsl**). Both Language Feature dimensions in the AHEAD matrix are identical.

8.2 Obe and the Tools Matrix

The **Tools** matrix defines the relationship between Language Features and Tool Features, exactly as the Origami theory prescribes. We wrote a program, called **obe** *(Origami Browser and Editor)*, which enables users to create, browse, fold, and edit 2-dimensional matrices. Users can define and document the units of each dimension and can define a number of files in each matrix element. The element files that we currently use are **entry.notes** and **gluon.expression**. **entry.notes** is simply a text file that we use to document an entry. **gluon.expression** is a file that specifies an expression — a composition of layers. We found that although gluons are indeed layers, in practice, gluons could be synthesized by composing even *more* primitive and reusable layers. So instead of replicating these primitive layers or their compositions in matrix entries, a more practical implementation of Origami is for its entries to define expressions for gluons. This is the approach taken in **obe**.

Unit	Description
Java	base Java language
AstDsl	AST constructors and escapes
Gscope	Generation scoping
BaseDsl	misc additions to Jak grammar
SourceDsl	SoUrCe statement
LocalIdDsl	layer scoping construct
LayerDsl	layer statement
ComposeClassDsl	statements for refining and composing classes
ComposeIntDsl	statements for refining and composing interfaces
ComposeSmDsl	statements for refining and composing state machines
antStuff	adds feature that allows tool to be invoked by ant

Table 2. Units of the Language Feature Dimensions

For a given matrix, users select the file they want to view. Figure 17a depicts the contents of the **Tool** matrix for the **gluon.expression** view. A matrix entry with "**x**" denotes the presence of a **gluon.expression** file; a blank denotes its absence. By clicking an entry, the contents of the selected file are displayed and can be edited. So in Figure 17, the [**Java**,**Base**] entry of **Tools** has a **gluon.expression** file, as do all other rows in the **Base** column except the **antStuff** row. Similar interpretations are given to other entries.[6]

	Base	j2j	JamPack	Mixin	mmatrix	UnMixin	SUMcols
Java	X	X	X	X	X	X	
AstDsl	X	X	X	X	X	X	
GscopeDsl	X	X	X		X		
BaseDsl	X	X	X	X	X	X	
SoUrCeDsl	X		X		X		
LocalIdDsl	X	X	X	X			
LayerDsl	X	X	X	X			
SmDsl	X	X					
ComposeClass...	X	X	X	X	X	X	
ComposeIntDsl	X	X	X	X	X	X	
ComposeSmDsl	X	X	X	X	X	X	
antStuff		X	X	X	X	X	
SUMrows							

Figure 17: Gluon View of the Tools Matrix

The **Tools** matrix is folded in two steps. First, a composition of Language Feature units defines the row composition order. The matrix that results from the folding is a single row, whose col-

5. The Tools Feature dimension of Table 1 reflects its state as of December 2002. Additional units have since been added to build **Jedi** and other AHEAD tools. However, the essential ideas remain the same.

6. Readers may have noticed an additional row (**SUMrows**) and column (**SUMCols**) in an **obe** matrix, beyond the units we have defined per dimension. **obe** allows matrices to be composed, and the result of composing selected rows appears in the **SUMcols** column and the result of composing selected columns appears in the **SUMrows** row.

umns are **{Base**, **j2j**, **jampack**, **mixin,** …**}**, as expected. To build a particular tool, columns of this synthetic row are composed. Each AHEAD tool is defined by its own equation. The **mixin** tool, for example, is defined by:

```
mixinTool = Mixin o Base
```

This equation specifies a column folding that pairs the **Base** parser (shared by all tools) with the **Mixin** layer that defines the semantic actions of the **mixin** tool. The same idea holds for all other tools.

8.3 The Ast Matrix

The unusual part of the AHEAD matrix is the **Ast** matrix. This matrix captures interactions among Language Features (in particular the **AstDsl** feature) and Tool Features. The contents of the **AstDsl** row in the **Tools** matrix is a function of the Language Features that are composed. This row is computed by folding the **Ast** matrix. Virtually all of our language additions to Java have no interaction with each other, because they are orthogonal to (or independent of) each other. The lone exception is **AstDsl** which adds metaprogramming constructs like *abstract syntax tree (AST)* constructors and escapes to Java. We first outline metaprogramming constructs that we have added to Java, and then show how AST constructs require the **Ast** matrix.

8.3.1 Metaprogramming Additions to Java

Consider the following code snippet. It assigns to variable **c** the code fragment that defines an empty class **foo**. A code fragment surrounded by delimiters **code{...}code** is a *code constructor*:

```
c = code{ class foo{ } }code;
```

When **c** is printed, the program:

```
class foo{ }
```

is generated. Programs that generate other programs are *metaprograms* or *generators*.

An important concept in metaprogramming is *staging* [38]. A typical metaprogram has two stages: the generator runs and then the generated program runs. A generator-generator has three stages: the generator-generator runs, the generated generator runs, and then the generated program runs. It is easy to recognize fragments of code that run at particular stages. For example, the snippet below shows code that runs in three stages:

```
s =                          // stage 1
  code{ f =                  // stage 2
       code{ a = b; }code;   // stage 3
  }code;
```

The assignment of a code fragment to **s** is done in the generator-generator (stage 1), the assignment of a code fragment to **f** is done by the generated generator (stage 2), and the assignment to variable **a** is by the generated program (stage 3).

In addition to code constructors, there are escapes. A *code escape* allows one to "pop-up" one stage within a code constructor. In the code snippet below, we create the boolean test "**a>b**" and insert it into an if statement using an escape denoted by **$code()**:

```
t = code{ a>b }code;
i = code{ if ( $code(t) ) foo(); }code;
```

$code(t) means substitute the value for code variable **t** that was defined in the previous stage — i.e., value "**a>b**". Thus, the above assignment for **i** is equivalent to:

```
i = code{ if ( a>b ) foo(); }code;
```

When a statement is parsed, it is a simple matter to determine the stage of each token. At the top of a parse tree, the stage is set to one. As the tree is traversed, the stage is incremented each time a code constructor is entered and is decremented upon exit. Conversely, upon entry to a code escape, the level number is decremented, and upon exit, the level number is incremented.

In general, the language that is used at each stage need not be the same. For example, a Java program could generate a Pascal program. However, the form of staging used in AHEAD is more typical: the same language — in our case, the **Jak** language — is used in all stages. As mentioned earlier, the Language Feature that adds code constructors and code escapes to the Java language is **AstDsl**.

8.3.2 Transforming Abstract Syntax Trees

AHEAD tools perform transformations on ASTs that are returned by a parser. These transformations are implemented by traversals that walk the tree and modify its content. One such traversal might be to harvest the names of all variables defined in an AST (i.e., parsed program). Harvesting variable names is a particularly simple traversal: when a parse tree node for a variable definition is encountered, the name of the variable is extracted and saved in a container.

A key assumption of traversal algorithms is that the nodes on which they perform their tasks are at stage 1. Nodes appearing at higher-stages are *not* harvested or transformed. Consider the class below: the only variable of this class is **s** (because it is at stage 1). Variable **b** exists at stage 2, and is *not* a variable of the **example** class. (It is a variable of a *generated* class, not a variable of the generator).

```
class example {
   void foo() {
      treeNode s = code{ int b; }code;
   }
}
```

A traversal in JTS and AHEAD is a method that is present in all AST nodes. This is possible because all AST node classes are descendant from a single class (**AstNode**); a traversal is a method that is added to **AstNode** and is thereby inherited by all of its subclasses. The default behavior of a traversal method is to do nothing except invoke the traversal method of the children of a node instance. Recall the example of harvesting variable names. The default **harvestName** traversal method might be:

```
void harvestName( int stage, Set result ) {
   for each subtree s {
      s.harvestName( stage, result );
   }
}
```

A particular harvesting or transformation task is introduced by overriding the traversal method for a node type. If an AST node type defines a variable name, its overriding **harvestName** method might be:

```
void harvestName( int stage, Set result ) {
   if (stage == 1) {
      harvest name and add it to result;
   }
   super.harvestName( stage, result );
}
```

where **super.harvestName(stage,result)** invokes the default behavior of harvesting names from the subtrees of that node. Note that harvesting occurs only when **stage == 1**.

In general, implementing traversals in this manner works well. Language Feature additions to Java add new AST node classes to the **AstNode** hierarchy, and default behaviors for traversals are inherited. When features don't interact, default behavior is appropriate. The problem comes when default behavior is inappropriate, and this occurs specifically when stage numbers are to be updated.

To see how we express feature interactions, keep the following in mind. An **AstDsl** layer introduces the **CodeConstructor** and **CodeEscape** classes as subclasses of **AstNode**. Each instance of **CodeConstructor** represents an AST of a code constructor expression. Similarly, each instance of **CodeEscape** represents the AST of a code escape expression.

Suppose a Language Feature (layer) **L** adds a traversal method **T** to **AstNode**. For traversal **T** to work correctly for the **CodeConstructor** and **CodeEscape** classes, we cannot use the default behavior of **T**. Instead, **T** must be overridden in **CodeConstructor** to be:

```
void T( int stage, Set result ) {
   super.T( stage+1, result );
}
```

so that the stage of the code constructor's expression is one higher the current level. Similarly, method **T** in **CodeEscape** must be overridden by:

```
void T( int stage, Set result ) {
   super.T( stage-1, result );
}
```

meaning that stage of the expression enclosed by the escape is one level lower than the current level. The collection of all these method overrides is encapsulated in a single layer **L***AstDsl*.[7]

7. Stated another way, a traversal can be thought of as a visitor [16]. **L***AstDsl* encapsulates a refinement of a visitor.

	Base	j2j	jampack	mixin	unmixin	mmatrix	SUMools
Java							
AstDsl		X					
GscopeDsl							
BaseDsl		X	X	X	X	X	
SoUrCeDsl							
LocalIdDsl			X	X			
LayerDsl							
SmDsl							
ComposeClass...			X				
ComposeIntDsl							
ComposeSmDsl							
antStuff							
SUMrows							

Figure 18: Gluon View of the Ast Matrix

There are specific conditions when **L***AstDsl* is used: **L***AstDsl* must appear in an equation iff Language Feature **AstDsl** is present (meaning that the **CodeConstructor** and **CodeEscape** classes are present) and Tool Feature **L** is present (meaning traversal method **T** is present). If either **AstDsl** or **L** is absent, **L***AstDsl* is not used. **L***AstDsl* encapsulates the interaction of Language Feature **AstDsl** with Tool Feature **L**.

8.3.3 The Role of the Ast Matrix

The **Ast** matrix encapsulates the interactions between **AstDsl** and all Tool Features. Figure 18 is the **obe gluon.expression** view of this matrix. The "**x**" entries identify the layers that introduce traversals (and hence, tool-interaction layers). For example, the **LocalIdDsl** layers for **jampack** and **mixin** add traversal methods.

Recall that the rows of the **Tools** matrix are folded by an equation that is a composition of Language Features. This same expression folds the rows of the **Ast** matrix; the computed row becomes the **AstDsl** row of the **Tools** matrix.

Stated another way, the Language Feature equation specifies the language features that will be in the **Jak** language. Each Language Feature identifies a unique row in the **Ast** matrix. This row enumerates the layers that define the interaction of code constructors and escapes with each Tool feature. By composing rows of the **Ast** matrix, the resulting entry in each Tool Feature column is the composition of all feature-interaction layers needed for the targeted variant of **Jak**.

8.3.4 Results

Folding the AHEAD matrix synthesizes five tools that are language-dialect sensitive: **j2j**, **mixin**, **unmixin**, **mmatrix**, **and jampack**. Table 3 lists for each tool its size in Java LOC, and the number of layers that define its equation. Just with this set of tools (and we expect many more), we are generating well over 100K LOC. Without Origami, our tool equations are seemingly randomly-ordered compositions of layers that are difficult to understand and update. Origami imposes a regularity in equation

organization that enables us to generate product-families from simple specifications. Even more important, it helps us control the complexity of feature-refinement-based representations of product-families.

Tool	# of layers	Size in Java LOC
j2j	27	31K
mmatrix	22	28K
mixin	26	28K
jampack	30	30K
unmixin	21	27K

Table 3. Size of Generated IDE Tools

8.3.5 Experiences and Future Work

Tool support for Origami is essential. **obe** only works for 2-dimensional matrices, and generalizations to n-dimension matrices (n>2) are needed. Conveniently visualizing multi-dimensional spaces remains an open problem, despite considerable prior research (e.g. [17]). In the meantime, we are exploring alternative encodings of matrices that are based purely on equational representations.

While our first examples of Origami demonstrate that feature refinements scale, it is equally important to illustrate microscopic examples as well, to demonstrate that Origami is applicable to arbitrary levels of abstraction. Finding good examples are the subject of current research, and preliminary examples are being distributed with the current release of AHEAD [2].

Finally, we believe the role of Origami will be essential to future models of feature refinements. The reason is that they lead to simple and appealing declarative languages (e.g., like the GUI of Figure 5) for specifying members of a product-family. Without Origami, the implementation of such languages/GUIs is not obvious and is subject to many ad hoc decisions. Origami provides an elegant way to structure orthogonal sets of features.

9 Relevance to Other Technologies

There are many non-GenVoca examples of Origami. One 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 software component models [29] that deal with *interface-based programming* — clients program to standardized interfaces and components implement these interfaces [35]. 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) [16].

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 non-essential.

The problem that we and other engineers 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 [19]. 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 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 to implement that new interface. The visitor design pattern allows almost arbitrary method extensions to existing components [16]. 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.

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

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 [10][25].

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 [34]. 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 [21] and has been improved by others [12]. 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 seem 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 [36][26][27]. 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., $class_1 \ldots class_n$ or $feature_1 \ldots feature_n$). 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 [37]. 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 [37]. Cross-cuts are problematic when new features impact every product in a product-family. However, instead of designing a system to easily handle features, [34] 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 AOP.

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 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 gratefully acknowledge the support of the U.S. Army *Simulation and Training Command (STRICOM)* contract N61339-99-D-10 and the *University of Texas Center for Agile Technology (UT:CAT)* for their support of our work. 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 clarify our presentation.

12 References

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

[2] AHEAD Tool Suite (ATS). **`http://www.cs.utexas.edu/users/schwartz`**

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

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

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

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

[7] 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*.

[8] D. Batory, J.N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement", *ICSE 2003*.

[9] I. Baxter, "Design Maintenance Systems", *CACM*, April 1992.

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

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

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

[13] A. van Deursen, P. Klint, "Little Languages: Little Maintenance?", *SIGPLAN Workshop on Domain-Specific Languages*, 1997.

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

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

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

[17] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D.Reichart, M. Venkatrao, F. Pellow, H. Pirahesh: Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Mining and Knowledge Discovery* 1(1): 29-53 (1997)

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

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

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

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

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

[23] G Kiczales, E. Hilsdale, J. Hugunin, M. Kirsten, J. Palm, W.G. Griswold. "An overview of AspectJ". *ECOOP 2001*.

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

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

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

[27] 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.

[28] 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.

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

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

[31] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers". 12th European Conference on Object-Oriented Programming, *ECOOP*, July 1998.

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

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

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

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

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

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

[38] W. Taha. "Multi-Stage Programming: Its Theory and Applications". Ph.D. thesis, Oregon Graduate Institute of Science and Technology, 1999.

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