IBM Systems Journal ■ Thursday, 6 April 2006
3:25 pm ■ Allen Press, Inc.
ibms-45-03-03 ■ Page 1

Multilevel models in model-driven engineering, product lines, and metaprogramming

Don Batory

Model-driven engineering (MDE) aims to raise the level of abstraction in program specification and increase automation in program development. These are also the goals of *product lines* (a family of related programs) and *metaprogramming* (programming as computation). We show that the confluence of MDE, product lines, and metaprogramming exposes a multilevel paradigm of program development, and further, we can use object-oriented design techniques to represent programs, the metaprograms that produced these programs, and the meta-metaprograms that produced these metaprograms, recursively. The paradigm is based on a small number of simple and well-known ideas, scales to the synthesis of applications of substantial size, and helps clarify concepts of MDE.

INTRODUCTION

Over the last decade, there has been an increasing desire in both research and practice to abandon the manual development of programs in favor of more automation.[1] Work on software product lines is an example.[2,3] A *product line* is a family of similar programs. Individual programs differ by the features that they support, where a *feature* is an increment in program functionality. By modularizing features, programs in a product line are produced by composing features[4]; that is, the process of developing a complex program can be reduced to the comparatively simple activities of feature selection and composition. Software tools automate the composition process.

More broadly, research on product lines and generative, transformational, and component-based programming[5] are progressing toward the goal of making programming a *computation*. This requires a fundamental shift in perspective on program design and development. Programs themselves become objects, and operations on programs are methods of such objects. *Metaprogramming* is the concept that programming is a computation.

Model-driven engineering (MDE) is an emerging approach to software development that centers on higher-level specifications of programs in *domain-specific languages (DSLs)*, greater degrees of auto-

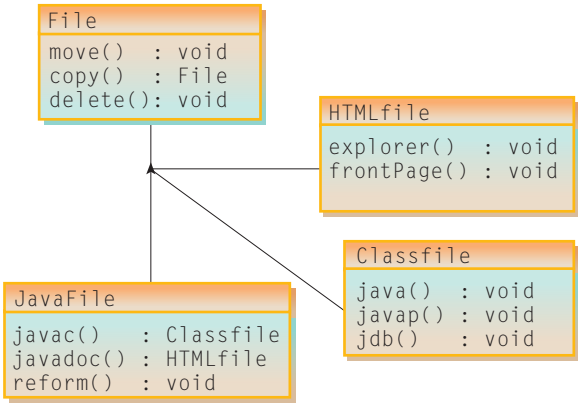

Figure 1
File types and tools

mation in software development, and the increased use of standards.[6] Among the tenets of MDE is the use of models to represent a program. A *model* is a specification written in a DSL that captures particular details of a program. As an individual model represents a limited range of information, a program specification is often defined by several models. A model can be computed from other models, and the process of building programs is one of transforming high-level models into executables (considered as yet other models).

Although MDE and metaprogramming are not identical, they do share concepts and goals; namely, that programs are first class (i.e., as objects or models), operations can be performed on them (i.e., as methods or transformations), program development can be a computation, and programs have multiple representations.

In this paper, it is argued that product lines enable both MDE and metaprogramming to converge on a multilevel paradigm of program design. This paradigm not only uses *object oriented (OO)* design techniques to represent programs that manipulate *everyday objects* (e.g., employees, books, ledger sheets), but also uses OO techniques to represent the *metaprograms* that produced these programs, and the meta-metaprograms that produced these metaprograms, recursively. The paradigm is based on a small number of simple and well-known ideas, scales to the synthesis of applications of substantial size, and helps clarify concepts of MDE.

MULTILEVEL DESIGNS

Long before MDE, software engineers realized that a program has many different representations, a simple Java** application, `.class` files, and `.html` files (produced by the `javadoc` tool).[7] A more elaborate application might have performance models (represented as Mathematica files), makefiles (in Extensible Markup Language [XML] format), formal models (as a state machine in an XML Metadata Interchange [XMI] file), and so on. Each representation is written in a language specific for its purpose—Java is good for source code, HyperText Markup Language (HTML) is good for documentation, and so forth.

Representations can be derived from other representations (e.g., a `.class` file is derived from its corresponding `.java` file by `javac`), or a representation may express unique information about a program (e.g., Mathematica files define a performance model that is not automatically derivable from source files). From this perspective, the software engineering community has been practicing a primitive form of MDE for years.

If we treat files (i.e., representations) as objects that are instances of file types, an OO design emerges. ***Figure 1*** identifies the file types that are encountered in the development of a Java program. File methods are implemented by Java *tools* that either transform a file into another representation (e.g., `.javac` is a method that maps a `.java` file to a `.class` file) or that modifies the file (e.g., `reform` is a pretty printing tool that transforms unruly `.java` files into beautifully formatted files[4]). Even inheritance relationships exist: operating systems provide a standard set of operations (move, copy, delete) on files of all types. Specialized file types are distinguished by different file extensions, and have their own methods (tools).

A makefile is a program that operates at this level of abstraction. It consists of one or more scripts that create objects (i.e., files) by invoking methods (i.e., tools) in a particular order, and whose goal is to maintain the consistency of these objects whenever an object is modified. (In effect, a makefile is a *metaprogram*—a program that produces a program.) Makefiles are written in a special language that is not object-oriented, but that could be given a class structure. ***Figure 2A*** shows the skeleton of an `ant` build script, and ***Figure 2B*** shows its corresponding

class structure (i.e., an `ant` project is a class, `ant` tasks are methods, and property definitions are members). Although this correspondence is loose, it is not difficult to recognize an OO class structure in other non-Java artifacts as well. We will return to this observation later in the section entitled "AHEAD."

Given the above, there are clearly two different levels of abstraction in program design. The *meta-application level* deals with the construction, manipulation, and synthesis of *application level* artifacts (files, etc.). Although there are OO languages to express programs at the application level, there is no language that unifies the concepts of files with objects, file instances with file types, tools with methods, and execution scripts (e.g., makefiles) as bodies of methods. (Perhaps Smalltalk and Lisp are exceptions, as they were both programming languages and programming environments.) Not surprisingly, programs at the meta-application level are developed by using tools and design techniques that are reminiscent of those used at the application level in the 1960s. The primary reason that OO techniques now dominate programming is that they impose more structure on programs. More structure means program complexity is better controlled, accidental complexity is reduced, and greater opportunities for automation and analysis arise (e.g., tools that restructure programs using design patterns). On the other hand, programs at the meta-application level are very simple—almost trivially so—compared to their counterparts at the application level. Perhaps this, for no other reason, explains why tools and design techniques at the meta-application level have not progressed.

MDE may be a driving force to change this. MDE explicitly embraces the idea that programs have multiple representations, thus complicating activities at the meta-application level. There are potentially many more program representations, called *models*, to keep track of; there are many more operations that can be performed on models to modify them or to derive other models. Keeping track of all these representations and relationships itself demands a modeling and programming language. Instead of inventing yet another set of concepts and terminology for this purpose—which itself would be enormously time consuming—why not consider OO as a starting point? The benefits seem clear.

First, OO provides a standard vocabulary and concepts for expressing program designs at any level

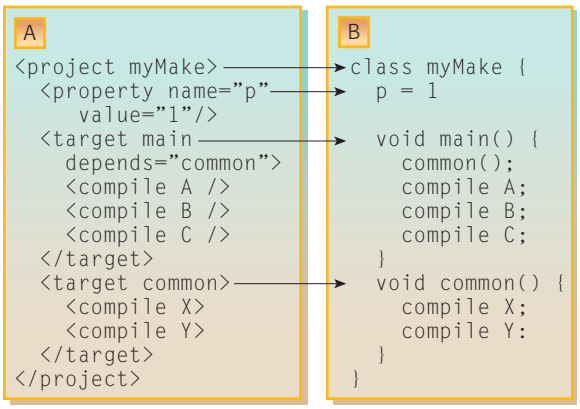

Figure 2
Makefile-class concept correspondence

of abstraction. Enabling researchers and developers to communicate with an established terminology would be extraordinarily beneficial, potentially saving the MDE community years of work.[8,9]

Second, OO provides a reasonable way to think about programs. It formats our thinking so that abstractions are more clearly distinguishable from their possible implementations. For example, today there is an increased interest in graph transformations to manipulate MDE models that are encoded as graphs.[10] While this is fine, let us not forget that there are many program representations that are graphs whose operations are not implemented by graph transformations. A Java program is clearly a graph of classes, yet no `javac` compiler that I know of is implemented by graph transformations.

These points are elaborated in the following sections by briefly describing projects whose authors unknowingly have used multilevel models to blend model-driven, metaprogramming, and product-line development. The purpose of this paper is to make explicit the notion and value of multilevel models and how to express them.

Let us begin by focusing on product lines whose programs have only one representation, namely source code. This requires the use of an elementary idea, called mixins, which was my first introduction to the idea of programs as objects and operations that map such objects. Once the power of mixins is appreciated, changing to the synthesis of programs with multiple representations is easier to understand.

MIXINS: FUNCTIONS THAT MAP CLASSES

A *mixin* is a class whose superclass is specified by a parameter. An elementary mixin (where I take liberties on syntax) that adds a color attribute with `set` and `get` methods to its input class is:

```
class addColor<class base> extends base {
   int color = 0;
   int getColor() { return color; }
   void setColor(int c) { color = c; }
}
```

With the above, colors could be added to skyscrapers, T-shirts, vertices, or any other classes whose objects need color:

```
cSkyscraper extends addColor<Skyscraper>;
cTShirt     extends addColor<TShirt>;
cVertex     extends addColor<Vertex>;
```

The motivation for mixins is code reuse: mixins encapsulate stereotypical extensions of classes that can be reused many times. From a design perspective, a mixin is a function that takes a class as input and produces an extended class as output. The `addColor` mixin places virtually no constraints on the classes that it extends. Although this is typical for classical mixins,[11,12] experience suggests this is uncommon.

A more typical example shows that a mixin not only can add new data members and methods to a class, but also it can *extend* existing methods. The following mixin can be applied to stack classes to count the number of times the `pop()` method has been invoked:

```
class countPop<class aStack>
extends aStack {
   int    cntr = 0;
   void   resetCntr() { cntr = 0; }
   int    getCntr() { return cntr; }
   Object pop() {
     cntr++;
     return super.pop();
   }
}
```

Note that `countPop` cannot be applied to just any stack, but only stacks that implement the `Object pop()` method. Additionally, `countPop` should only be applied to stacks that do not already have `cntr`, `resetCntr()`, or `getCntr()` members, as overriding them may break methods that depend on the overridden semantics.[13] This example is more revealing about the nature of mixins (or more generally, about the nature of functions that map programs). Mixins are not conceived in isolation, but rather are carefully designed with other mixins and base classes so that they are compatible. Mixins that invoke or extend base class methods, which is the norm, demand coordinated designs. This intimacy is further amplified by the fact that mixins should be designed to extend a particular family of classes, so that the result of applying a mixin to a class yields another class within the same family. This latter point is crucial, as it is the key to synthesizing a family of classes by mixin composition. This family is a product line.

As an example, Berger created a mixin library, called *Heap Layers*, to define a product line of memory allocators.[14] It had a few base allocators (`mallocHeap`, `mmapHeap`) and a set of over 20 mixins to extend these allocators (`debugHeap`, `profileHeap`, ...). The design of a particular allocator starts with a base allocator, and then, a sequence of mixins is applied to give the target allocator its desired features. Memory allocator `a1` allocates space from a memory-mapped heap and includes checks for a variety of allocation errors:

```
class a1 extends debugHeap<mmapHeap>;
class a2 extends profileHeap<a1>;          (1)
```

whereas allocator `a2` additionally collects and outputs fragmentation statistics. Of course, the examples in Reference 14 are much more elaborate, but I hope that the idea is clear. The reported benefits of this design are that sophisticated allocator implementations can be developed quickly and cheaply, and performance is comparable to hand-coded allocators. These benefits are typical of this genre of work.

Before we proceed, let us reflect on a tenet of OO design. The methods and internal representation of state for an object are designed with each other in mind—they are not created in isolation. Prior to OO, languages like C allowed programmers to separately define `struct`s and functions over instances of `struct`s. The idea of a class as a conceptual module that integrates both a `struct` and its functions was missing. By analogy, if mixins are added to Java, they are likely to have free-standing definitions in a program, much like generics have now in Java 1.5; therefore classes and mixins (functions on classes)

are defined separately; there is no module concept that integrates base classes and their mixins. Yet our experience tell us that the designs of base classes and mixins are indeed intimately connected; their connections are no different than members and methods of an everyday OO class; that is, base classes and mixins must agree on the names and semantics of methods, the representation of state, and so forth. The level of modularity of mixins needs to be raised from being free-standing functions on classes to *methods of a class of classes* (or more generally, a *class of programs*).

In other words, by adding mixins as stand-alone entities like class and generic declarations, OO language researchers are introducing structured-programming language concepts for the meta-application level in OO programming languages. Rather than repeating the structured versus OO arguments of the 1980s–90s, we should use OO concepts at the meta-application level immediately.

We can understand the Heap Layers mixin library as a two-level OO design. The bottom or application level defines the class structure of memory allocation programs. Superimposed on this design is a higher or meta-application level that consists of a single OO class whose instances are different memory allocator programs. Basic allocators (`mallocHeap`, `mmapHeap`) are produced by constructors, and mixins are methods (i.e., functions) that add features to these programs.[15] Thus, the Heap Layers mixin library has the following meta-application-level class structure:

```
class MemAlloc {
  private MemAlloc_Representation state;
  static MemAlloc mallocHeap(){...}//constructors
  static MemAlloc mmapHeap() {...}

    // optional features
    MemAlloc adaptHeap() {...}
    MemAlloc chunkHeap() {...}
    ...
}                                                (2)
```

That is, all instances of the `MemAlloc` product-line share a common internal representation and are expressions (i.e., metaprograms), such as:

```
MemAlloc a1,a2;
a1=MemAlloc.mmapHeap().debugHeap();
a2=a1.profileHeap();                             (3)
```

Note that `MemAlloc` program representations could be abstract syntax trees, and `MemAlloc` methods would manipulate these trees. To relate the class definition `a2` of (1) with the metaprogram variable `a2` of (3), we might need to invoke a `toSource()` method on the `a2` variable to produce its source-code representation. The use of `toSource()` is implied in our discussions. Although there are earlier examples of two-level designs,[16–18] the Heap Layers mixin library is ideal for our discussion because a memory allocation program is just a single class, and mixins are perfect for extending programs that can be defined by a single class. But memory-allocation programs lack scale in two important ways. First, most programs are rarely simple enough to be single classes, and the changes made by most program extensions are rarely limited to a single class. How can mixins scale to encapsulate changes to larger programs? Second, most meta-application-level models of product lines are also not defined by a single class; they, too, can have multiple classes. Both questions are addressed in the next section.

Mixin layers: Functions that map programs

Smaragdakis showed how mixins could scale to modularize extensions of multiclass programs.[19,20] Instead of storing the classes of a program in a package, program classes could be listed as inner classes of a single class. Suppose `myProgram` is defined by the set of classes `[A...Z]`. We could encode `myProgram` as a single class with `A...Z` as inner classes:

```
class myProgram {
   class A { ... }
   ...
   class Z { ... }
}
```

A mixin `myMixin` could then be defined to modularize changes to `myProgram`, such as adding new classes (e.g., class `AA`) and extending previously defined classes (e.g., class `Z`):

```
class myMixin<class base> extends base {
   class AA { ... }
   class Z extends base.Z { ... }
}
```

`myMixin` is a *mixin layer*—it is a mixin that modularizes *a program extension*, that is, the

addition of classes and extensions to existing classes. Semantically, a mixin layer defines a stereotypical extension that can be applied to a set of programs. In a product-line context, a mixin layer implements a feature. A program of a product line has a particular set of features; it is synthesized to a base program by applying a sequence of mixin layers.

This approach to the design and synthesis of programs is called *GenVoca*.[20,21] A GenVoca model `M` of a product line has one or more base programs `b1...bm` (called *constants*) and one or more mixin layers (called *functions*) `f1...fm`. A design for a program `p` in the product line of `M` is an expression, such as `p = f2(f4(b1))`. When GenVoca models are viewed from a meta-application-level perspective, most are single classes. The constants correspond to constructors, and functions correspond to methods. The meta-application-level class structure of a typical GenVoca model `M` is:

```
class M {
   private_M Representation state;
   static M b1(){...}// constructor #1
   static M b2(){...}// constructor #2
   ...
   M f1(){...} // method or feature #1
   M f2(){...} // method or feature #2
   ...
}
```

That is, all instances of `M`'s product line share a common internal representation and are expressions (i.e., metaprograms), such as:

```
M p = M.b1().f4().f2();
```

The Heap Layers mixin library is an example of a GenVoca design.

Genesis was a product line of relational database systems; it was also the first example of a GenVoca design.[21,22] Genesis could synthesize a product line of file management systems, relational storage systems, and relational database systems through feature (mixin-layer) composition. Unlike memory allocation programs, each Genesis system contained many different classes, and its meta-application model also had multiple classes. Genesis is only sketched in the following paragraphs; the actual design is more complicated.

A *file management system (FMS)* is a program that performs low-level activities to store and retrieve tuples of individual relations. Tuples are stored in primitive file structures, such as `BTree`, `Hash`, and `Heap`s. Alternatively, inverted files could be used, where tuples are stored in one structure (such as `Heap`) and index records are stored in another (such as `Hash`). Optionally, tuples could be compressed. An FMS had a single meta-application-level class structure:

```
class FMS {
   private FMS_Representation state;
   static FMS BTree () {...} // constructors
   static FMS Hash () {...}
   static FMS Heap () {...}

  {...} // optional features
   FMS indexed( FMS indexfile )
   FMS compress () {...}
}
```

Particular FMS designs are instances of class `FMS`. For example, `fs1` defines an FMS that stores compressed tuples on a heap, and `fs2` is an inverted file where tuples are stored on a heap and index records are stored in `Btree` structures:

```
FMS fs1. fs2;
fs1 = FMS.Heap().compress();
fs2 = FMS.Heap().indexed( FMS.BTree() );
```

A relational storage system (RSS) differs from an FMS in that it supports relational join operations in addition to relation retrievals. The features of RSS include different join algorithms (`mergejoin`, `hashjoin`, `nestedloops`) and an adapter to translate RSS interface calls into FMS calls (`adaptfms`). An RSS has a meta-application-level class structure:

```
class RSS
   private RSS_Representation state;
   static RSS adaptfms ( FMS f ) {...} // constructor

  // optional features
   RSS mergejoin() {...}
   RSS hashjoin() {...}
   RSS nestedloops() {...}
}
```

As an example, `r1` is a relational storage system that uses nested loops and hash joins to process rela-

tional joins, and stores relations in an `fs1` file system:

```
RSS r1 = RSS.adaptfms( fs1 )
                    .nestedloops().hashjoin();
```

Finally, a *relational database system (RDS)* layers a query language on top of an RSS. Genesis provides two different languages, `Quel` and `SQL` (Structured Query Language). An RDS has the meta-application-level class structure:

```
class RDS {
   private RDS_Representation state;
   static RDS Quel ( RSS r ) {...} // constructors
   static RDS SQL ( RSS r ) {...}
}
```

Relational system `s1` presents an SQL front end to relational storage system `r1`:

```
RDS s1 = RDS.SQL ( r1 );
```

Using the `FMS`, `RSS`, and `RDS` classes, architects could synthesize different file management systems, relational storage systems, and database systems. Although three meta-application-level classes have been sketched, Genesis has many more.

There is nothing magical about mixin-based technologies. They reduce to a simple development concept: when a feature is added to a program, new packages, classes, data members, and methods can be added, and existing methods can be extended.

I no longer use the traditional programming language concept of a mixin as it has, among other problems, complications with fix-points, the inability to add new constructors, and the inability to rename variables. I have since replaced it with a mixin-like technology that provides more general capabilities.[4] Other technologies, such as aspects, could be used to achieve these and even greater effects.

DESIGNS WITH THREE OR MORE LEVELS

Both Heap Layers and Genesis are examples of a two-level design. Three and higher-level designs are indeed possible and are modeled no differently than the application and meta-application levels; that is, we leverage OO's contribution to scale invariance to describe more abstract levels of program description.

Suppose we want to synthesize the `MemAlloc` class of Heap Layers, to customize it with a particular set of constructors and methods. Conceptually, this is the process of creating a *product line of product lines (PLoPL)*. A PLoPL is modeled by a class on a third or meta-meta-application level.

The idea is simple: create a class `MAPL` (memory allocation product line). We have a constructor (`base`) that generates a `MemAlloc` class that has only a program-representation data member. We then define methods (i.e., mixins) of this class to add specific constructors and methods to the target `MemAlloc` class:

```
class MAPL {
   primitive MAPL_Representation state;
   static MAPL base(); // constructor

   // features that add constructors
   // and methods to a MAPL object
   MAPL addConstructorMallocHeap();
   MAPL addConstructorMmapHeap();
   MAPL addMethodAdaptHeap();
   MAPL addMethodChunkHeap();
   ...
}
```

Thus, the `MemAlloc` class (2) that we considered earlier would be synthesized by a meta-meta-application-level expression:

```
MAPL MemAlloc =
   MAPL.base().addConstructorMallocHeap()
    .addConstructorMmapHeap()
    .addMethodAdaptHeap()
    .addMethodChunkHeap(). ...;
```

For details on how such models are implemented, see Reference 23. Interestingly, PLoPLs arise naturally in another form, called *staged configuration* models, where a product line is progressively simplified in different stages for different sets of customers.[24] In principle, levels higher than three may exist; they, too, would be described as other levels. In summary, multilevel models allow us to define a hierarchy of product lines. An individual application is defined at the bottom level. A product line of applications is defined at the meta-applica-

A
```
// tokens
INT {0-9}*;

// rules
list : value
     | list ',' value
     ;

value : INT ;
```

B
```
// tokens
ID {a-Z}{a-Z0-9}*;

// rules

value : super
      | ID
      ;
```

C
```
// tokens
INT {0-9}*;
ID {a-Z}{a-Z0-9}*;

// rules
list : value
     | list ',' value
     ;
value : INT
      | ID
      ;
```

Figure 3
Grammars and an extension

tion level. A PLoPL is defined at the meta-meta-application level, and so on. So far, we have considered the synthesis of programs that have only a single representation (e.g., source code). We remove this restriction in the next section.

AHEAD

Consider a parser. It is specified by at least two different program representations at the application level. There is a grammar representation that defines the grammar of the language and a source-code representation that defines the semantic actions that are to be performed when a rule of the grammar is matched. Although both representations can reference each other, neither is fully derivable from the other. (In the language of MDE, our application is specified by two different models, each written in its own DSL.)

When a feature is added to a language, its parser must be updated. This means that the grammar must be extended (with new tokens and rules that define the language extension), and the source code must be extended (with the semantic actions for these new rules). From previous sections, we know how to extend the source-code representation of a program—use a mixin-like technology to add new members and new methods to existing classes, add new classes, and extend existing methods. But how are grammars extended?

In the section "Multilevel designs," we noted that a class structure could be imposed on non-Java artifacts. With a class structure, we could use mixin-like technologies to extend these artifacts as well. This idea is called the *Principle of Uniformity*—give all artifacts (program representations) a class structure, and extend them analogously to code.[4]

Let us illustrate this with a simple grammar. ***Figure 3A*** shows the Backus Naur Form (BNF) of a list of integers. We can impose a class structure: a grammar is a class, tokens are data members, and rules are methods. A grammar can be extended by adding new tokens, new rules, and extending existing rules.

Figure 3B shows an extension of this grammar that generalizes a list to include identifiers. The `value : super` construct means extend the righthand side of the previously defined `value` production. Readers may recognize that this extension is a *mixin for a grammar*. The expected result of composing the base grammar (Figure 3A) with its mixin extension (Figure 3B) is shown in ***Figure 3C***.

The big picture is simple. Programs have many different representations. When a feature is added to a program, any or all of its representations may change. For example, the source code of a program changes (to implement that feature), its performance model changes (to profile that feature), its documentation changes (to document that feature), and so on.

The relationships between program representations have a simple OO description. Suppose a program has two representations: `C` (for code) and `G` (for grammar). A GenVoca product line of such programs is described by one or more base programs `b1`, `b2`, ... and one or more optional features f1, f2, ... We can define a meta-application class for each representation:

```
class C {
   private C_Representation state;
   static C b1_C() {...};  // constructors
```

```
    static C b2_C() {...};
    C f1_C() {...};  // optional features
    C f2_C() {...};
    ...
}
class G {
    private G_Representation state;
    static G b1_G() {...};  // constructors
    static G b2_G() {...};
    G f1_G() {...}; // optional features
    G f2_G() {...};
    ...
}
```

A product line of programs that have consistent `G` and `C` representations is a class whose objects maintain a `G` and `C` state, and when a method is invoked (i.e., a feature is added), both representations are updated. This is expressed by another meta-application-level class `P`:

```
class P {
    private G g;
    private C c;
    static P b1() {  // constructors
        P p = new P();
        p.g = G.b1_G();
        p.c = C.b1_C();
        return p;
    }
    ...
    P f1() {  // optional features
      P p = this.clone(); // copy
      p.g = g.f1_G();
      p.c = c.f1_C();
      return p;
    }
    ...
}
```

Note that the code of class `P` above is so predictable that it could be produced automatically. Thus, program `p`, which is base program `b1` extended with feature `f1`, is the expression (metaprogram):

```
P p = P.b1 ().f1();
```

Internally, `p` maintains two consistent representations: a `C` representation and a `G` representation, the values of which are:

```
p.c = C.b1_C().f1_C();
p.g = G.b1_G().f1_G();
```

These ideas extend to an arbitrary number of program representations and have been implemented in the AHEAD Tool Suite.[4] See Reference 25 for an example of these ideas. Although the emphasis of AHEAD is more on a mathematical description of meta-level models and their composition, it is not difficult to recognize this correspondence.

AN MDE CASE STUDY

Bold Stroke is a product line written in several million lines of C++ that supports a family of mission-computing avionics for a variety of military aircraft.[26] Recent work by Gray et al. used Model-Integrated Computing (MIC) by Sztipanovits and Karsai at Vanderbilt University,[27] which is a specific platform for MDE, to demonstrate how models could capture the design of the Bold Stroke code base and how simple model edits could be automatically translated into substantial code-base edits.[28] The manner in which artifacts of a MIC design process are related and manipulated is ad hoc, requiring manual adaptations due to the lack of an OO design (i.e., objects and methods) at the meta-application level.[29] Consequently, the elegance of MIC designs is obscured. We use Bold Stroke as a case study, and we invite readers to compare our explanation with Reference 28 to judge the difference.

Briefly, Gray et al. use three different representations of Bold Stroke: the C++ code base, its *Embedded-Systems Modeling Language (ESML)* model representation,[30] and an XML representation for configuring the start-up of Bold Stroke. The XML representation is derived directly from the ESML model by the `ConfigurationInterface` model interpreter (i.e., transformation). Features can be added to Bold Stroke: among them are different kinds of locks for supporting concurrency (e.g., `externalLock`, `internalLock`, and `noLock`) and logging a "black box" data recorder (`logging`). Each feature is specified by a separate transformation. We capture this meta-application design by a single class:

```
class BoldStrokeApp {
  private CppCode C;  // code repr
  private ESML M;  // model repr
  BoldStrokeApp () {...}  // constructor
  XML ConfigurationInterface () {
    return M.deriveXMLconfiguration();
```

```
  }
  BoldStrokeApp externalLock() {...} // features
  BoldStrokeApp internalLock() {...}
  BoldStrokeApp noLock() {...}
  BoldStrokeApp logging(options) {...}
}
```

A `BoldStrokeApp` object maintains two different representations: an ESML model and C++ code. Both are data members. A version `v` of Bold Stroke that has external locks and logging is:

```
BoldStrokeApp v = new BoldStrokeApp().
        externalLock().logging(options);
```

When the `externalLock` method is invoked, both representations change. Let `UpdLM()` be a method that updates an ESML model by weaving in external-lock declarations, and let `UpdC(ESML x)` be a method that, given an ESML model `x`, updates the C++ code to conform to `x`. The `externalLock` method becomes:

```
BoldStrokeApp externalLock() {
  BoldStrokeApp e = this.clone();
  e.M=M.UpdLM();       // weave locks into model
    e.C=C.UpdC(e.M); // weave lock code
    return e;
}
```

Gray et al. use the term *2-level weaving* to describe the weaving of the model and then the use of the woven model to determine how to weave the code base. This idea is easily captured above. Features are added to Bold Stroke by sequentially invoking feature methods. Gray et al. noticed an interesting optimization: there is no need to incrementally update the code representation after each model update. Instead, model updates can be batched, and the code representation is updated only when it is needed. We can express this idea by adding a `getCode` method to class `BoldStrokeApp`:

```
CppCode getCode() {
  return C.UpdC(M); // weave code updates here
  }
```

This allows us to simplify the definition of `externalLock` (and other feature methods) by updating only the model representation:

```
BoldStrokeApp externalLock() {
  BoldStrokeApp e = this.clone();
  e.M = M.UpdLM();
  return e;
}
```

This optimization does not synchronize the model and code representations. Model changes are collected, and modifications to the code are made only when the `getCode` method is invoked. This is a common OO coding idiom. Of course, considerable work is needed to implement these methods; but the simplicity of design is now evident.

Reference 28 is a good example that demonstrates a point of this paper: not only is OO good for defining the structure of models, it is also good for defining the methods (transformations) on these models as well. By modeling transformations as methods, the simplicity and elegance of MDE designs are revealed. Not doing so makes it difficult for authors to convey their ideas and for readers to understand them.

OTHER MDE CONCEPTS

There are many concepts and proposals in MDE that can be expressed in an OO manner. To illustrate one particular example, we note that the MDE literature is steeped in discussions of models, metamodels, and meta-metamodels.[31,32,33] Briefly, a model is a representation of a program and is an instance of a metamodel (that is, a model is an object and the metamodel is its class). Recursively, a metamodel is an instance of a meta-metamodel. Generally, recursion stops at the meta-metamodel level, as it is an instance of itself.[34]

Metamodels are paramount to MDE; they are the definitions of DSLs. The ability to create custom languages for particular domains and the ability to write models (specifications) in these languages is the key to MDE's success, and naturally is a primary focus of tool development in MDE platforms.[1,10,27]

Given the above, what is the relationship between our meta-application level and the more established MDE concept of metamodels? Here is a simple way to understand their connection. Many people find the concept of models, metamodels, and meta-metamodels intimidating, but these really are fancy names for familiar and well-understood concepts in databases. A database schema is a metamodel; an instance of a schema is a database. Every DBMS (database management system) internally has a

cast-in-concrete meta-metamodel, called the *information schema*, whose instances are database schemas.[35,36] If readers think in terms of databases, the metaconcepts of MDE are easy to appreciate.

To leverage this analogy, think of a database as an object, where different database instances of the same schema are different objects. The state of a database object is defined by its tuples. Transactions are methods on database objects that update its state or that create new databases.

Now, revise the preceding paragraph by replacing "database" with "model," "schema" with "metamodel," and "transactions" with "transformations." Paraphrasing:

A *model* is an object, where different *model* instances of the same *metamodel* are different objects. The state of a *model* object is defined by its tuples. *Transformations* are methods on *model* objects that update its state or that create new *models*.

Doing so, an analogy to describe fundamental ideas in MDE emerges. ***Figure 4*** lists the correspondence of database, MDE, and OO terminologies, where each row of terms are different names for the same concept (i.e., a model is an object; an object is a database).

These are the essential ideas behind the pioneering work in MIC.[27] A MIC model is literally stored as a database of tuples, and the schema of the database is its metamodel.[37] MIC now uses a graph transformation technology to define methods (i.e., tools) that map models.[27] Previously, low-level C++ code was used to write such tools.

The connection to our meta-application level is that MDE platforms enable an architect to define individual classes and their object instances at the meta-application level. Object state is defined by a model, the schema of a model is a metamodel, and object methods are transformations.

MDE platforms aim to replace traditional programming languages with DSLs. An object technology is needed to relate these classes, their instances, and their methods. This is our meta-application level. Product lines fit naturally into an OO meta-application level as additional methods (that correspond to features) in meta-application-level classes.

Databases	MDE	OO
information-schema	meta-meta-model	meta-class
schema	meta-model	class
database	model	object
transaction	transformation	method

Figure 4
Terminology correspondences

Sadly, what makes all this confusing and complicated is that different names have been given by different communities to the same concept. By using a single name per concept, the ideas become clearer.

CONCLUSIONS

The history of software development is marked by a series of advances that progressively raise the level of abstraction in which programs are specified; as the level increases, so does the degree to which software development can be automated.

MDE is an emerging technology that has the potential to go far beyond today's OO languages and program-development technologies. In this paper it was argued that MDE should be a next-generation OO technology that focuses on programming at the meta-application level (and higher levels), where objects are programs and methods are operations that map programs. At this level, program design and development is a computation, historically called metaprogramming. The synergistic value of the convergence of these ideas is apparent in product lines, where features are operations that transform programs (i.e., methods of program objects), and program development is function (method) application.[38]

Case studies were presented which demonstrated that a multilevel design paradigm for product lines in MDE is feasible, that it scales to the synthesis of large applications, and that it helps clarify concepts of MDE. I hope these ideas take us a step closer to realizing a broader vision of software development.

ACKNOWLEDGMENTS

This research is sponsored by NSF's Science of Design Project #CCF-0438786. I thank many people for their helpful comments on drafts: Jeff Gray (University of Alabama) who recommended that I

look at his work on Bold Stroke, Rick Snodgrass (University of Arizona) for the name "information schemas" as the database term for "schema of schemas," Gabor Karsai (Vanderbilt University), Emery Berger (University of Massachusetts), Steve Cook (Microsoft Corporation), and my colleagues at the University of Texas, Greg Lavender, William Cook, and Srinivas Nedunuri.

**Trademark, service mark, or registered trademark of Sun Microsystems, Inc.

CITED REFERENCES AND NOTES

1. J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi, *Software Factories: Assembling Applications with Patterns*, Models, *Frameworks and Tools*. Wiley, 2004.
2. K. Czarnecki and U. Eisenecker, *Generative Programming Methods, Tools, and Applications.* Addison-Wesley, Boston, MA, 2000.
3. D. M. Weiss and C. T. R. Lai, *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, Reading, MA, 1999.
4. D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Transactions on Software Engineering (TOSEM)* (June 2004), pp. 355–371.
5. R. Laemmel and J. Sariava, Visser J. (editors), *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering*, Braga, Portugal (July 2005). To appear in Springer Lecture Notes in Computer Science series, 2006.
6. A. W. Brown, G. Booch, S. Iyengar, J. Rumbaugh, and B. Selic, "An MDA Manifesto," Chapter 11 in *Model Driven Architecture Straight from the Masters*, D. S. Frankel and J. Parodi, Editors, Meghan-Kiffer Press, Tampa, FL, 2004.
7. P. Zave, "A Compositional Approach to Multiparadigm Programming," *IEEE Software* **6**, No. 5, 15–25 (September 1989).
8. Different names for the same concept at the application and meta-application levels will always increase complexity and the potential for confusion, as the section "Other MDE Concepts" illustrates.
9. It could be argued that the ideas expressed in this paper could have just as easily been cast in terms of concepts from functional programming languages. Although this may be true, there is considerable evidence that inheritance is not part of the functional-programming-language paradigm. It is an essential idea of mixins, mixin layers, and program extensions which are central to this paper.
10. A. Agrawal, "Graph Rewriting And Transformation (GReAT): A Solution for the Model Integrated Computing (MIC) Bottleneck," *Automated Software Engineering 2003*, pp. 364–369.
11. G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Meta-Object Protocol*, MIT Press, 1991.
12. D. A. Moon, "Object-Oriented Programming with Flavors," *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, and Applications (OOPSLA) 1986*, pp. 1–8.
13. Languages with mixins typically satisfy the first constraint by requiring the base class to implement an interface that contains the Object pop () method. It is not always clear how the second constraint is expressed.
14. E. D. Berger, B. G. Zorn, and K. S. McKinley, "Composing High-Performance Memory Allocators," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2001,* pp. 114–124.
15. Constructors and methods can have additional parameters. See Reference 14 for details.
16. D. Batory, V. Singhal, J. Thomas, and M. Sirkin, "Scalable Software Libraries," *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering 1993,* Los Angeles, CA (December 7–10 1993), pp. 191–199.
17. I. Holland, "Specifying Reusable Components Using Contracts," *European Conference on Object Oriented Programming (ECOOP) 1992*, pp. 287–308.
18. M. VanHilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs," *Proceedings of ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '96)*, San Jose, CA (October 1996), pp. 359–369.
19. Y. Smaragdakis and D. Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs," *ACM Transactions on Software Engineering and Methodology (TOSEM)* **11**, No. 2, 215–255 (April 2002).
20. Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers," *Proceedings of the Twelfth European Conference on Object Oriented Programming (ECOOP '98),* Brussels, Belgium (July 20–24, 1998), *Lecture Notes in Computer Science* **1445**, Springer-Verlag (1998), pp. 550–570.
21. D. Batory and S. O'Malley, **"**The Design and Implementation of Hierarchical Software Systems with Reusable Components,**"** *ACM Transactions on Software Engineering and Methodology (TOSEM)* **1**, No. 4, 355–398 (October, 1992).
22. D. Batory, "Concepts for a Database System Compiler," *Proceedings of the ACM SIGSOFT Symposium on Principles of Database Systems (PODS)* Austin, TX, (March 21–23, 1998), ACM, New York, pp. 184–192.
23. D. Batory, J. Liu, and J. N. Sarvela, "Refinements and Multidimensional Separation of Concerns," *ACM SIGSOFT Symposium on the Foundations of Software Engineering 2003*, Helsinki, Finland, September 1–3, 2003, ACM, New York, pp. 48–57.
24. K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models," *Software Process Improvement and Practice* **10**, No. 2, 143–169 (2005).
25. O. Diaz, S. Trujillo, and F. I. Anfurruita, "Supporting Production Strategies As Refinements of the Production Process," *Software Product Line Conference (SPLC) 2005*, pp. 210–221.
26. D. Sharp, "Component-Based Product-Line Development of Avionics Software," *Software Product Line Conference (SPLC) 2000*, pp. 353–370.
27. J. Sztipanovits and G. Karsai, "Model Integrated Computing," *IEEE Computer* **30**, No. 4, 110–111 (April 1997).
28. J. Gray, J. Zhang, Y. Lin, S. Roychoudhury, H. Wu, R. Sudarsan, A. S. Gokhale, S. Neema, F. Shi, and T. Bapty, "Model-Driven Program Transformation of a Large Avionics Framework," *Generative Programming and Component Engineering (GPCE) 2004*, pp. 361–378.
29. J. Gray, personal communication, Sept. 2005.

30. www.isis.vanderbilt.edu/projects/mobies/downloads.asp
31. OMG/MOF, "Meta Object Facility (MOF) Specification," OMG Document AD/97-08-14, http://www.omg.org.
32. J. Bezivin, "Model Driven Engineering: Principles, Scope, Deployment, and Applicability," in Reference 5.
33. J. Bezivin, "From Object Composition to Model Transformation with the MDA," *Technology of Object-Oriented Languages and Systems (TOOLS'USA)* (August 2001), p. 350.
34. This three-level model is consistent with prior work on metaclasses in OO programming languages. Forman and Danforth's text (Reference 35) provides a good explanation. Reflection is a form of metaprogramming when a program is allowed to change itself.
35. ISO/IEC 9075. *Database Language SQL*, International Standard ISO/IEC 9075:1992, American National Standard X3.135-1992, American National Standards Institute 1992.
36. DBMSs store schema descriptions as tuples in database relations called information schema tables. Tuples in such tables are called metadata. (See Reference 35.)
37. G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits, "Composition and Cloning in Modeling and Metamodeling," *IEEE Transactions Control Systems Technology* **12**, No. 2, 263–278 (March 2004).
38. I. Forman and Scott Danforth, *Putting Metaclasses to Work*, Addison-Wesley, 1999.

Accepted for publication January 3, 2006.

Don Batory
Department of Computer Sciences, 2.124 Taylor Hall, The University of Texas at Austin, 78712. Don Batory holds the David Bruton Centennial Professorship in the Department of Computer Sciences at The University of Texas at Austin. He is an Associate Editor of *Journal of Aspect Oriented Development* and was Associate Editor of *IEEE Transactions on Software Engineering* (1999–2002) and Associate Editor of *ACM Transactions on Database Systems* (1986–1992). He is a leading researcher on feature-oriented program development. Over the last 10 years, he and his students have received over eight Best-Paper-in-Conference awards for their work in automated and component-based program development. He has given numerous tutorials on feature-oriented programming and is an industry consultant on product-line architectures. ■