First Software Product-Line Conference (SPLC1) 2000

# Object-Oriented Frameworks and Product-Lines[1]

Don Batory[i], Rich Cardone[i], & Yannis Smaragdakis[ii]

*Department of Computer Sciences, The University of Texas, Austin, Texas 78712* [i] *&*
*College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332* [ii]
*{batory, richcar}@cs.utexas.edu, yannis@cc.gatech.edu*

**Keywords**: object-oriented frameworks, refinements, components, product-line architectures, GenVoca.

**Abstract**: Frameworks are a common object-oriented code-structuring technique that is used in application product-lines. A *framework* is a set of abstract classes that embody an abstract design; a *framework instance* is a set of concrete classes that subclass abstract classes to provide an executable subsystem. Frameworks are designed for reuse: abstract classes encapsulate common code and concrete classes encapsulate instance-specific code. Unfortunately, this delineation of reusable v.s. instance-specific code is problematic. Concrete classes of different framework instances can have much in common and there can be variations in abstract classes, all of which lead to unnecessary code replication. In this paper, we show how to overcome these limitations by decomposing frameworks and framework instances into primitive and reusable components. Doing so reduces code replication and creates a component-based product-line of frameworks and framework instances.

## 1 INTRODUCTION

A *product-line architecture (PLA)* is a design for a family of related applications. The motivation for PLAs is to simplify the design and maintenance of program families and to address the needs of highly customizable applications in an economical manner. Although the idea of product families is old (McIlroy, 1968; Parnas, 1976), it is an area of research that is only now gaining importance in software design (Bosch, 1998; DeBaud, 1999;

1. This work was supported in part by Microsoft, Schlumberger, the University of Texas Applied Research Labs, and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

Gomaa et al., 1994; Cohen and Northrup, 1998; Batory 1998; Weiss and Lai, 1999; Czarnecki and Eisenecker 1999).

A *framework* is a collection of abstract classes that encapsulate common algorithms of a family of applications (Johnson and Foot 1998). Certain methods of these classes are left unspecified (and hence are "abstract") because they would expose details that would vary among particular, fully-executable implementations. Thus a framework is a "code template"—key methods and other details still need to be supplied, but all common code is present in abstract classes. A *framework instance* provides the missing details. It is a pairing of a concrete subclass with each abstract class of the framework to provide a complete implementation. These subclasses encapsulate implementations of the abstract methods, as well as other details (e.g., data members specific to a particular framework instance).

Frameworks often arise in PLA implementations. This is hardly unexpected: frameworks are appropriate for reusing software parts and specializing them in multiple ways for distinct applications. Members of a product-line can be created by refining framework classes with appropriate concrete implementations. Frameworks are a fundamental technique because of their simplicity and generality—from an implementation standpoint, frameworks are just a coordinated use of inheritance. Since inheritance is a fundamental mechanism in object-oriented languages, the applicability of the framework approach is wide.

The factoring of common algorithms into reusable, abstract classes greatly reduces the cost of software development when building a new product-line application (i.e., when creating a framework instance). Unfortunately, this delineation of reusable vs. instance-specific code has problems. Concrete classes of different framework instances can have much in common. This situation is typically addressed by either copying code (with predictable maintenance problems) or redeveloping concrete classes from scratch (which is costly). A different problem arises with abstract classes: they can have variations. Much of the code in abstract classes would be common across variations, but some parts would be radically different. Variability is typically handled by replicating the abstract classes of frameworks and hard-coding the changes into a new framework. Framework proliferation ensues, again incurring maintenance problems.

These problems are real. In a recent discussion with a member of IBM's SanFrancisco project,[2] these limitations of frameworks have become apparent. While code replication is not yet burdensome because SanFrancisco is

still new, difficulties are anticipated in the future. We believe that these problems arise in other projects that use OO frameworks.

A simple way to state the above problem is that product lines with optional features will not be concisely expressed using frameworks. Such frameworks suffer from either "overfeaturing" (Codenie, De Hondt, Steyaert, and Vercammen 1997) —a lot of not-entirely-general functionality is part of the framework— or replication of the same code across many instances. In this paper we present a general technique to solve this problem. The solution is effected by *relaxing the boundary between a framework (the common part of the product line) and its instantiations (the product-specific part)*. More specifically our technique is based on assembling both the abstract classes of frameworks *and* the concrete classes of their instances from primitive and reusable components. We show that the level of abstraction that delineates abstract classes from concrete classes can be drawn in different ways, and by decomposing frameworks and their instances in terms of our components, variations in abstract and concrete classes can be handled without code replication. A particular framework or framework instance is created by a composition of components; variations in frameworks and their instances are expressed as different component compositions. Our approach can be used to express any framework, but requires more language support than plain frameworks—instead of regular inheritance, parameterized inheritance is needed. Nevertheless, this support is readily available in production languages like C++. An example in an extended version of the Java language is given to illustrate our ideas.

## 2 GENVOCA AND COLLABORATION-BASED DESIGNS

In this section we offer an overview of some design-level ideas that underlie our work. Many terms and concepts used in the rest of the paper are defined here.

**Collaboration-Based Designs**. It is well-known in object-oriented design that objects are encapsulated entities that are rarely self-sufficient. The semantics of an object is largely defined by its relationship with other objects. Object interdependencies can be expressed as collaborations. A

2. IBM SanFrancisco is a Java-based set of components that allows developers to assemble server-side e-business applications from existing parts, rather than building applications from scratch.

*collaboration* is a set of objects and a protocol (i.e., a set of allowed behaviors) that determines how these objects interact. The part of an object that enforces the protocol of a collaboration is called the object's *role* in that collaboration (Reenskaug, et al, 1992; VanHilst and Notkin 1996; Smaragdakis and Batory 1998).

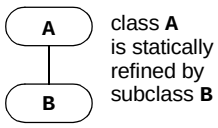
A *collaboration-based design* expresses an application as a composition of separately-definable collaborations. In this way, each object of an application represents a collection of roles describing actions on common data. Each collaboration, in turn, is a collection of roles that encapsulates relationships across its corresponding objects.

> **Example**. I (Batory) am an author of this paper. This relationship is defined by a collaboration of two objects, one in the role of author and another in the role of manuscript. I am also a parent—a collaboration of two or more objects, one in the role of parent and others in the role of children. I am also a car owner—a collaboration of two or more objects, one in the role of owner and others in the role of possession. And so on. I am a single object where relationships that I have with other objects are expressed through collaborations where I play a specific role in each collaboration.
>
> The collaborations mentioned above are generic; they are not specific to me but are general relationships that can be defined in isolation of each other. Furthermore, such collaborations are applicable in different contexts to different entities. For example, a corporation can own a car, and so too can a government entity. The relationship between owner and possession is the same in all cases, but the ownership roles are played by very different classes of objects (e.g., people, corporations, government). Symmetrically, a possession could be a car, dog, or building; the possession role can also be played by very different classes of objects. The same holds for roles in other collaborations.

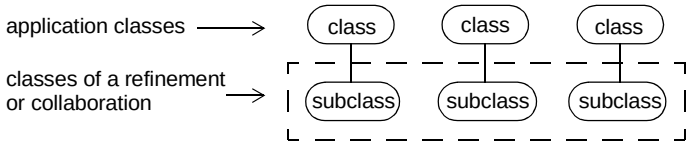
**GenVoca**. GenVoca is a design methodology for building architecturally-extensible software—i.e., software that is extensible via component addition and removal (Batory and O'Malley 1992). GenVoca is a scalable outgrowth of an old and practitioner-ignored methodology of program construction called *step-wise refinement*. GenVoca freshens this methodology by scaling refinements to a *component* or *layer* (i.e., multi-class-modularization) granularity, so that applications of enormous complexity can be expressed as a composition of a few refinements rather than hundreds or thousands of small refinements (c.f. (Partsch and Steinbruggen, 1983)).

**Relationship**. GenVoca product-lines and collaboration-based designs are intimately related: a GenVoca refinement is a collaboration. An application is constructed by composing reusable refinements (collaborations). Refinements can be composed dynamically at application run-time or statically at application compile-time. To address the problems of OO frameworks noted in the Introduction, we consider only statically composable refinements in this paper.

Consider how refinements are expressed statically in OO languages. A *static refinement* of an individual class adds new data members, new methods, and/or overrides existing methods. Such changes are expressed through subclassing: class **A** is refined by subclass **B**:




Both collaboration-based designs and GenVoca deal with *large-scale refinements*: such refinements involve the addition of new data members, new methods, overriding existing methods, etc. simultaneously to *several* classes:




The encapsulation of these “refining” subclasses in the above figure defines both a GenVoca *component* or *layer* and a collaboration. (Note that we are showing only subclassing relationships in this figure; there can be any number of “horizontal” interrelationships among individual subclasses).

> **Example**. Have you ever added a new feature to an existing OO application? If so, you discover that changes are rarely localized. Multiple classes of an application must be updated simultaneously and consistently for the feature to work properly. Similarly, if one subsequently removes that feature, all of its updates must be simultaneously removed from all affected classes. It is this collection of changes that we want to encapsulate as an application building block.

Each subclass of a layer encapsulates a role of a collaboration-based design. For a collaboration-based design to be "hooked" into an application, each role must be bound with an existing class of the application. We will see in Section 3 that layers can be expressed as *templates* and that such binding is accomplished via *template parameterization*. Thus, a layer defines a collaboration, while a layer instance additionally defines role/class bindings.

**Compositions**. When a layer (collaboration) is composed with other layers, a forest of subclassing (inheritance) hierarchies is created. As more layers are composed, the hierarchies become progressively broader and deeper. Figure 1 illustrates this concept: layer **`L1`** encapsulates three classes. Each of these classes root a subclassing hierarchy. Layer **`L2`** encapsulates three classes, two classes refine existing classes of **`L1`** while a third starts a new hierarchy. Layer **`L3`** also encapsulates three classes, two of which refine classes of **`L1`** and **`L2`**. Finally, layer **`L4`** encapsulates two classes, both of which refine existing classes.

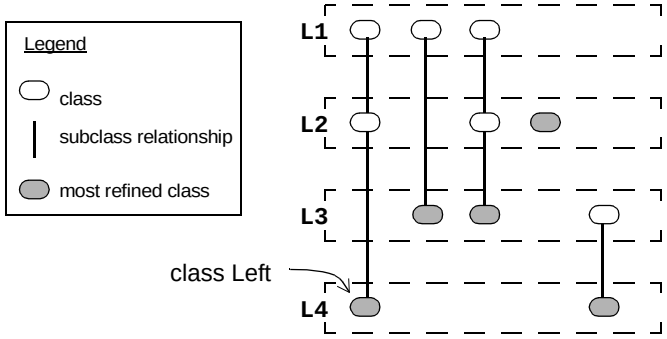



Figure 1 Creating Inheritance Hierarchies by Composing Layers

Each inheritance or refinement chain of Figure 1 represents a *derivation* of its terminal class. That is, each terminal class (shaded in Figure 1) is a product of its superclasses, where each superclass defines a role in some collaboration. In general, the classes that are instantiated by an application are the terminal classes, because these classes encode all the roles that are required of application objects. (For example, an object of class **`Left`** in Figure 1 plays three distinct roles which originate from collaborations **`L1`**, **`L2`**, and **`L4`**). The non-terminal (non-shaded) subclasses represent intermediate derivations of application classes. Thus, the composition of layers **`L1`** through **`L4`** yields five classes (i.e., those that are shaded in Figure 1); the unshaded classes represent the "intermediate" derivations of these shaded classes. If the resulting complexity of class hierarchies is a concern, preprocessors can be built to "accordion" (compact) inheritance chains so that

only the terminal classes remain. The resulting classes would be *conceptually* layered, but not *physically* layered (Habermann, Flon, and Cooprider, 1976). In general, collaboration-based designs ultimately reduce complexity by shifting the design emphasis from small-scale components (individual classes) to large-scale ones (entire collaborations).

**Product-Lines**. A layer implements a *feature* (*aspect*, *capability*) that can be shared by many applications of a product-line. An application that supports a given set of features is defined by a composition of layers that implements those features. Thus, layers are the basic building blocks for families of applications. In general, *n* layers can be composed in excess of *n*! ways, because the order of composition matters and layer replication is possible.[3]

## 3 MIXIN-LAYERS

A *mixin-layer* is a template representation of a GenVoca refinement (Smaragdakis and Batory, 1998; Findler and Flatt, 1998). Templates are important in our methodology because they allow writing source-code components that can be used in multiple contexts. We will use *Jak*, a superset of Java that supports templates, to explain the basic technique (Batory, Lofaso, and Smaragdakis, 1998).[4] Mixin-layers are an interesting meld of parameterized inheritance, inner classes, and standardized naming conventions.

**Mixins**. A *mixin* is a class whose superclass is specified by a parameter (Bracha and Cook, 1990).[5] Below we define a mixin **`M`** whose superclass is defined by parameter **`S`**. **`AnyClass`** is a Java interface that all classes implement:

```
class M <AnyClass S> extends S { ... }
```

3. So it is not uncommon that different applications of a product-line can be assembled by composing exactly the same layers in different orders (Batory and O'Malley, 1992; Hayden, 1998). Figure 1 provides an illustration: the order of **`L2`**-**`L4`** could be permuted, provided that **`L4`** is "below" **`L3`**.

4. We will not offer a strict definition of the *Jak* language, but its diversions from Java are few and, we hope, mostly self-explanatory. The reader can think of the semantics of our parameterization mechanism (templates) as those resulting from straightforward textual substitution. We will not address the potential problems of an actual textual substitution implementation as these are orthogonal to the topic of this paper.

5. C++ has a different meaning of "mixin" that is *not* equivalent to our use.

Mixins provide the capability of creating customized inheritance hierarchies when they are composed.

**Nested Classes**. Class declarations in Java can be nested inside other class declarations, and are inherited like data members and methods. Consider the following example:

```
class Parent { class Inner { ... } }
class Child extends Parent { }
```

**`Child`** is a subclass of **`Parent`**. Although no **`Child.Inner`** class is explicitly defined, such a class does exist as it is inherited from **`Parent`**. Nested classes emulate the encapsulation of multiple classes within a package, except this representation allows "packages" to appear as nodes in inheritance hierarchies.

**Combining Ideas**. A *mixin-layer* is an implementation of a collaboration. It is a mixin with nested classes, where each nested class corresponds to a role of a collaboration. A general form of a mixin-layer **`M`** is a *Jak* template that has *n+1* parameters: one parameter **`S`** that defines the superclass of **`M`**, plus *n* additional parameters {$\mathbf{r}_1 \ldots \mathbf{r}_n$} that define the specific classes the collaboration's role classes are to refine:

```
class M <AnyClass S, AnyClass r1, ... AnyClass rn>
extends S {
   class role1 extends r1 { ... }
   ...
   class rolen extends rn { ... }
   ...
   // additional non-refining classes (if any)
}
```

When a domain is decomposed into primitive collaborations, experience has shown that different collaborations use the same names for roles, and classes that have the same role names refine each other when their collaborations are composed. While the above template for mixin-layer **`M`** is general, a much more compact form standardizes names of inner classes and eliminates role-class parameters to yield a template with a single parameter **`S`**, the mixin-layer's superclass:

```
class M <AnyClass S> extends S {
   class role1 extends S.role1 { ... }
   ...
   class rolen extends S.rolen { ... }
   ...
   // additional non-refining classes (if any)
}
```

This form of a source-code component is interesting because it enforces a clear structure. Mixin-layers are written in such a way that they are composable with each other. Yet, composing layers entails fixing the superclass of *all* classes inside a layer. Thus, layers are simple to use (e.g., they have a single parameter), but can affect large portions of a software application.

**Collaboration Typing**. Astute readers may have noticed that mixin-layer **M** should not have an unconstrained parameter; substituting an arbitrary class for parameter **S** is unlikely to work. A legal instantiation of **S** must satisfy constraints, e.g., it must have inner classes {**role**$_1$ … **role**$_n$}. This gives rise to the notion that collaborations (layers) are typed and so too are their parameters. Unfortunately, Java currently offers no support for type-checking nested classes. For instance, it is not possible to use an **implements** clause to express the requirement that a class should contain a nested class that supports a certain interface. Therefore, we limit ourselves to very simple properties that can be expressed in the Java type system. Namely, we use an empty interface **R**. Collaborations that "implement" **R** are said to be of type **R**; parameters of type **R** will be legally instantiated only by collaborations of type **R**. Thus the set of layers of Figure 1 can be represented as:

```
interface R { }                  // empty

class L1
   implements R { ... }          // mixin-layer L1
class L2 <R x> extends x
   implements R { ... }          // mixin-layer L2
class L3 <R x> extends x
   implements R { ... }          // mixin-layer L3
class L4 <R x> extends x
   implements R { ... }          // mixin-layer L4
```

Although typing collaborations in this manner goes a long way toward ensuring that parameters have legal instantiations, additional properties are needed to distinguish the case where components with identical interfaces have different semantics (Batory and Geraci, 1997; Smaragdakis and Batory, 1998). For the purposes of this paper (and without loss of generality), we make the simplifying assumption that typing is sufficient.

**Compositions**. Refinements are composed by instantiating one mixin-layer with another as its parameter. The two classes are then linked as a parent-child pair in an inheritance hierarchy. The final product of a collaboration composition is a class **Fig1** with the general form (expressed in Jak):

```
class Fig1 extends L4< L3< L2< L1 >>>          (1)
```

That is, **L4**, **L3**, …, **L1** are mixin-layers, "**<...>**" is the *Jak* operator for template instantiation, and **Fig1** is the name given to the class that is produced by this composition. (This particular composition corresponds to Figure 1). The classes of **Fig1** are referenced in the usual way, namely **Fig1.role**$_1$ defines the application class **role**$_1$, etc. Readers who are familiar with GenVoca will recognize such compositions as *type equations*, which has an alternative and more compact syntax:

```
Fig1 = L4< L3< L2< L1 >>>  // type equation of (1)  (2)
```

The space of all type equations corresponds to all applications that can be synthesized in this product-line.

## 4 LIMITATIONS OF OO FRAMEWORKS

A common case where frameworks prove to be too rigid is that of optional features. If a set of features are often but not always used, they cannot be encoded in the framework. (Otherwise, they will burden or render incorrect any framework instances not needing these features.) Thus, such features need to be encoded independently (i.e., replicated) in each framework instance that uses them. We will show in this section that using mixin-layers as building blocks for frameworks and their instances, we can encode an optional feature as a mixin-layer and include or exclude it at will from a specific composition.

Recall that a framework is a set of classes. For simplicity, our prior discussions assumed that all framework classes are abstract, but in general they need not be. Non-abstract classes could encapsulate a capability that is shared by (and can be optionally extended by) all framework instances. We will proceed under this more general setting. We also assume that mixin-layers have *no* variations (e.g., no optionally-selected algorithms) and that their collaborations are "monolithic". Variations in product-line applications arise *only* from variations in compositions of mixin-layers. We will relax this assumption later.

To see the relationship between mixin-layers and frameworks, consider Figure 2a which replicates the inheritance hierarchies of Figure 1. Suppose we drew a line between layers **L2** and **L3**, where classes above the line define the classes of a framework. In Figure 2a, there would be four such classes {**A**$_1$, **A**$_2$, **A**$_3$, **A**$_4$}. Note that these classes correspond to the "most refined" classes of the refinement chains that lie above the line. The most refined classes that lie below the line define the concrete classes of a frame-

work instance. In Figure 2a, there would be four such classes {**C**$_1$, **C**$_2$, **C**$_3$, **C**$_5$}. (Note that for this framework instance, **A**$_4$ need not be subclassed/ refined). If we had a language preprocessor that would "accordion" (compact) refinement chains so that only the most refined classes remained, Figure 2b shows the result of this compaction. Readers will recognize Figure 2b as an encoding of a framework's classes and its instance classes.

Two points are worth noting. First, the classes of the framework of Figure 2 are defined by the type equation **F** = **L2<L1>**. An instance of this framework is any type equation whose innermost term is **F** (e.g., **Fig1** = **L4<L3<F>>**). *From this we can conclude that mixin-layers are building blocks of both frameworks and framework instances.*

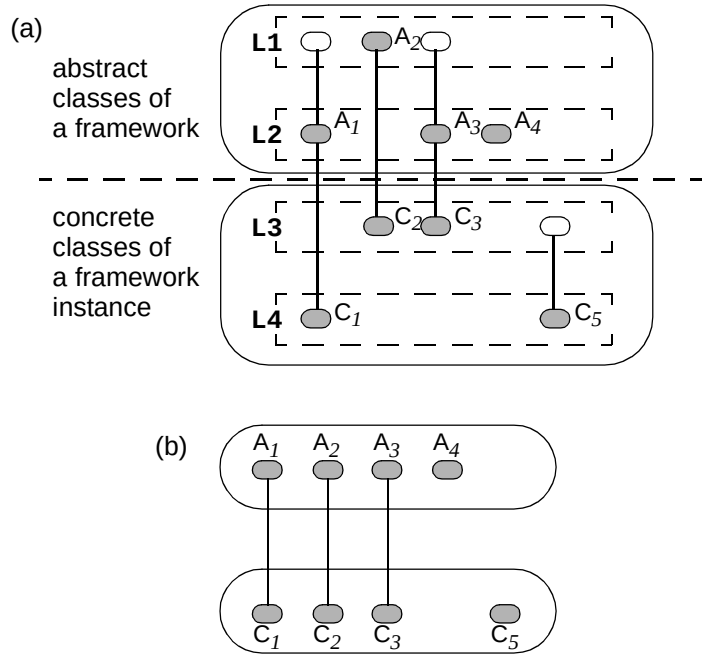



Figure 2: Refinement Hierarchies and Framework Instances

Second, we could have drawn our line in between any two adjacent layers and produced a different framework and one of its instances. (There is nothing special about the boundary between **L2** and **L3**). If the boundary is raised, the framework will become more general (as framework classes are simplified), but more code will need to be written for a framework instance. If the boundary is lowered, framework classes will encapsulate more features at an expense that the framework may be too specific (i.e., have too many features) to be used for a particular application.

But why partition along layer boundaries? Why not partition *across* layer boundaries? To see the answer, consider Figure 3 which defines the partitioning between framework and instance classes by crossing multiple layer boundaries. The framework of Figure 3 would consist of classes {**A**$_1$, **A**$_2$, **A**$_3$}. The framework instance that is depicted would consist of concrete classes {**C**$_1$, **C**$_3$, **C**$_4$, **C**$_5$}, where **C**$_3$ includes superclass **K**$_3$, and **C**$_5$ includes superclass **K**$_5$.

Look carefully at what Figure 3 implies: *any* instance of the framework of Figure 3 *must* replicate classes **C**$_3$ (which includes **K**$_3$), **C**$_4$, and **K**$_5$. The reason is simple: all collaborations encapsulate the implementation of some primitive feature that is shared by many applications of a product line. If the classes of a framework implement only part of features **L2** and **L3** (which they do in this case), then any legal instance of this framework must supply the missing parts. Classes **C**$_3$, **K**$_3$, **C**$_4$, and **K**$_5$ are the missing parts and these parts do not vary (as we assumed at the beginning of this section). The framework of Figure 3 is a bad design because it forces the same code to be replicated in every framework instance. The only framework designs for which this isn't true are those that partition framework code from instance code along layer boundaries. Stated another way, the classes of a framework must fully implement an integral number of collaborations otherwise code replication in framework instances will occur.

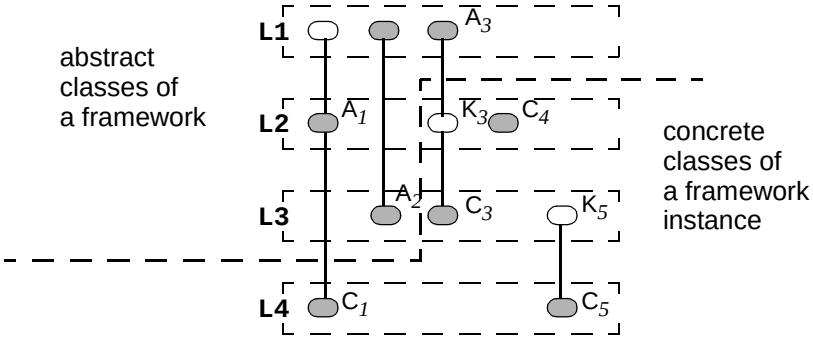



Figure 3: Partitioning Against Layer Boundaries

From our experience, the above model of framework construction covers a majority of situations that are encountered in practice. That is, variations in frameworks and their instances can be explained as the composition of monolithic mixin-layers. Occasionally however, mixin-layers and their collaborations do have variations. If there are few variations, one could define a separate mixin-layer for each variant (see example of the next section). If there are a large number of variations, it is not difficult at all to create a generator of mixin-layers that will produce the desired variant of a

collaboration from a high-level specification (Batory, Singhal, Sirkin, and Thomas, 1993). Doing so retains the building-block nature of mixin-layers (as we have been advocating) while forming a more compact encoding of the library of mixin-layers that must be maintained. By following either of these approaches, we can restructure the problem so that "good" framework designs can always be expressed as a partitioning *along* layer boundaries and not *across* layer boundaries.

Without loss of generality, let us assume "good" designs whose framework-instance partitioning corresponds to a layer boundary. We can now understand the problems of OO frameworks that we noted in the Introduction. Variations in framework classes arise whenever the type equation of a framework changes. Consider framework **F** whose type equation is **F** = **L2<L1>**. Any change to this equation—by swapping components (**F1** = **L2<L0>** for some new terminal component **L0**) or adding new components (**F2** = **L2<L5<L0>>** for some new nonterminal component **L5**)—will cause the classes of the framework to change.

Similarly, code repetition in multiple framework instances corresponds to the situation where the type equations of framework instances share the same framework subexpression and some (not necessarily all) remaining components. Consider equations **Fig1 = L4<L3<F>>** and **Q = L4<L5<F>>**. Both framework instances are distinct, but share the same framework subexpression (**F**) and component **L4**.

A way to avoid code replication problems in frameworks is to decompose the domain that a framework and its instances represent into a library of refinements/components. These are the components that should be given to application developers. They will choose which type equation (component composition) that they need to define their framework and/or framework instance. The boundary where an abstract class ends and concrete classes begin is left up to the developer. We would expect that the library of components that is distributed to be incomplete. That is, we would not expect the library to have enough components to construct a target framework instance (because we anticipate novel capabilities to be added by the instance). However, if there are sufficient components (or even just a few that can be reused), application developers will be further along their software development than they would otherwise.

## 5 AN EXAMPLE PRODUCT-LINE

To illustrate the concepts of the previous sections, we present a GenVoca product-line model of graph traversal applications (Smaragdakis and Batory, 1998). This application domain is interesting because of the interchangeability of its components. More complex examples are discussed in Section 6 (Related Work).

**The Model**. A graph traversal application is a program that implements traversals on graphs. The library of refinements that we consider focuses on two traversals: vertex numbering and cycle checking. (The library can be expanded with other traversals—see (Smaragdakis and Batory, 1998)). The membership of our traversal library is:

```
undirected          // undirected graphs
directed            // directed graphs
dft<G x>            // depth-first traversal
bft<G x>            // breadth-first traversal
number<G x>         // vertex numbering
cycle<G x>          // cycle checking
```

where all members implement the (empty) interface **`G`**.

The mixin-layers **`undirected`** and **`directed`** implement undirected graphs and directed graphs, respectively. Both encapsulate a pair of classes **`Vertex`** and **`Graph`**. The methods of these classes support vertex addition and removal from graphs, but not traversals. Both mixin-layers are designed to implement the same interface, so that they are plug-compatible and interchangeable.

The mixin-layers **`dft<G x>`** and **`bft<G x>`** implement depth-first and breadth-first traversals, respectively. Both encapsulate refinements of the **`Vertex`** and **`Graph`** classes and add new abstract class **`WorkSpace`**. (Thus the **`dft`** and **`bft`** mixin-layers have three inner classes, two of which are mixins). The **`Vertex`** and **`Graph`** classes are refined with the addition of traversal methods: **`GraphSearch`** is added to **`Graph`** and **`VertexSearch`** is added to **`Vertex`**. Both methods take a **`WorkSpace`** object as a parameter. At various times during a graph or vertex search, e.g., prior to visiting a node and after visiting a node, a dispatch is made to the **`WorkSpace`** object for graph-traversal-specific actions. Each **`WorkSpace`** object supports three abstract methods: **`init_vertex`** (to initialize a vertex for a particular traversal), **`preVisitAction`**, and **`postVisitAction`**.

The mixin-layers **number<G x>** and **cycle<G x>** implement vertex numbering and cycle checking, respectively. Both encapsulate refinements of the **Vertex** and **Graph** classes, in addition to adding a subclass to **WorkSpace**. (**number** adds the subclass **WorkSpaceNumber**; **cycle** adds the subclass **WorkSpaceCycle**). **number** refines **Graph** by adding the **VertexNumber** method (which is called by application users to invoke vertex numbering); **number** refines **Vertex** by adding a public integer called **vertexCount** (which holds the assigned number of a vertex). The **WorkSpaceNumber** class supplies methods for initializing a vertex (i.e., setting **vertexCount** to zero), doing nothing for a **preVisitAction**, and assigning a number to a vertex for the **postVisitAction**. The **cycle** mixin-layer encapsulates similar capabilities and refinements for cycle-checking.

**The Product-Line**. Consider the following type equations:

```
frame1 = dft<undirected>
inst11 = number<frame1>
inst12 = cycle<frame1>
inst13 = number<cycle<frame1>>
```

A framework is defined by **frame1**: it is a set of classes that encapsulate a depth-first traversal on undirected graphs. These classes are incomplete in that there is no traversal application; an application must be supplied by extending these classes in a framework instance. Distinct framework instances are defined by **inst11**—**inst13**. A vertex numbering application is defined by **inst11**; a cycle checking application is defined by **inst12**; **inst13** defines an application that supports both vertex numbering and cycle checking. Note that one of the limitations of frameworks is exposed by this example: two different framework instances share common code (e.g., **inst11** and **inst13** share the **number** component; **inst12** and **inst13** share the **cycle** component). By encapsulating domain features as mixin-layers, we minimize code replication through component reuse.

Now consider the equations:

```
frame2 = dft<directed>
inst21 = number<frame2>
inst22 = cycle<frame2>
inst23 = number<cycle<frame2>>
```

A framework is defined by **frame2**: it is a set of classes that encapsulates a depth-first traversal on directed graphs. The framework instances **inst21**—**inst23** respectively define applications for vertex numbering, cycle checking, and both numbering and cycle checking on directed graphs. Note that the other limitation of frameworks is exposed by this example: the

variation of classes in a framework (e.g., **`frame1`** and **`frame2`** are distinct frameworks that share the **`dft`** component). Other variations can be created by swapping **`dft`** with **`bft`**. This would yield vertex numbering and cycle checking applications on directed graphs using breadth-first search algorithms.[6]

Note that our frameworks above encapsulated a pair of features: a graph encoding and a traversal; a framework instance added the traversal application(s). We could have "raised the delineation line" so that our framework merely encoded a graph; instances of this framework would have to provide a traversal method and application:

```
frame0 = undirected;
inst01 = number<dft<frame0>>   // equivalent to inst11
inst02 = cycle<bft<frame0>>
```

The advantages of this decision is that the framework is more general, but writing instances is more work. Alternatively, we could have "lowered the delineation line" to enrich the capabilities of our framework:

```
frame00 = number<dft<undirected>>// same as inst11
inst01 = cycle<frame00>          // same as inst13
```

The advantages of this decision is that less code needs to be written in framework instances at an expense that the framework may be too specialized to use for a particular application. Mixin-layers, however, eliminates the annoying problem of deciding where to draw the framework-instance "line"; application designers are free to define the contents of frameworks as they see fit. This provides designers more flexibility in customizing their applications than using frameworks whose designs are inflexible to such customizations.

## 6 RELATED WORK

**Use of Mixin-Layers**. Product-lines using mixin-layer components have been created for extensible compilers (Batory, Lofaso, Smaragdakis, 1998; Findler and Flatt, 1998) and command-and-control simulators for fire support for the U.S. Army (Batory, Johnson, MacDonald, and von Heeder,

6. Note that there is a definite order in which components can be legally composed: a directed/undirected graph component can be refined by a depth/breadth-first component, which can be refined by one or more traversal applications. Our typing of these components does not encode these constraints. See (Smaragdakis and Batory, 1998; Batory and Geraci, 1997) for constraint enforcement.

2000). Non-OO implementations of early versions of mixin-layers can be found in product-lines for databases, file systems, and network protocols (Batory and O'Malley, 1992; Heidemann and Popek, 1994).

**Library Scalability**. The drawbacks that we have noted with frameworks are classical examples of the *library scalability problem* (Batory, Singhal, Sirkin, and Thomas, 1993; Biggerstaff, 1994). The idea is simple: in a domain where there are *n* optional features, there can be in excess of *n!* different programs, each implementing a unique combination of features. Library components should *not* implement combinations of features, because (obviously) libraries would have exponentially large memberships. A better approach is to populate libraries with building blocks that implement *individual* features, and compose these blocks to synthesize the program with the desired combination of features. Such libraries are *scalable*: they grow at a linear rate, but the number of programs that can be synthesized from component combinations grows at an exponential rate.

We have seen that the classes of a framework may correspond to a composition of primitive refinements (i.e., mixin-layers). The classes of a framework instance may also correspond to a composition of primitive refinements. Since both are treated as encapsulated units, any variation made to either (corresponding to the addition, removal, or replacement of one or more refinements) will theoretically lead to an exponential number of variations to maintain. Our contribution in this paper is to show how to solve the library scalability problem for frameworks and framework instances.

**Parameterized Components**. GenVoca is an example of a programming paradigm called *parameterized programming*—that applications are synthesized by composing components via parameter instantiation (Goguen, 1986). It is interesting to note that programming support for mixin-layers is presently limited in Java. The most well-known versions of Java that offer parameterization (e.g., Pizza and GJ (Odersky and Wadler, 1997; Bracha, Odersky, Stoutamire, and Wadler, 1998)) do not support parameterized inheritance. This led to the development of JTS, a tool suite for creating a product-line of Java dialects, which does support parameterized inheritance (Batory, Lofaso, and Smaragdakis, 1998).

Most work on parameterized programming deals with parametric source code. Industry prefers to distribute binaries rather than source. Our approach presently cannot extend binary components, where the source code of mixin-frameworks is unavailable. This is a temporary problem. The static parameterizations of mixin-layers are simple enough to be expressed

as parameterized binaries (e.g., parameterized Java .class files). That is, parameter instantiation is accomplished at class load time rather than at mixin-layer compile time. Recent work on parameterized Java class binaries suggests this possibility is not far off (Duncan and Hölzle, 1999). Thus, we anticipate in the future that libraries of binaries will be distributed.

**Aspect-Oriented Programming (AOP)**. AOP is intimately related to refinements. An *aspect* is feature of a domain whose implementation "cross-cuts" multiple application classes (Kiczales, et. al., 1997). When an aspect is added to an existing application, multiple classes must updated. Clearly, aspects are refinements. An application-specific AOP implementation can provide custom refinements to any existing application. GenVoca implementations, on the other hand, start by conceptually decomposing legacy applications and resynthesizing them in an extensible way through component composition. Interface conformance plays a prominent role in the software composition process of GenVoca, more so than in AOP.

**Reuse Contracts**. The problems of framework version proliferation and architectural drift may be mitigated through formal annotations on classes (Codenie, De Hondt, Steyaert, and Vercammen, 1997). Reuse contracts record the design intentions of reusable classes and the assumptions made by actual users of those classes. Automated annotation checking can detect if new modifications violate the contract from either the producer or consumer point of view. Codifying the management of framework evolution in this way limits the proliferation of code that violates reuse conventions.

GenVoca is "neutral" on the use of contracts. Contracts can be used in the development and composition of mixin-layers. So the benefits accrued by using contracts are also available to layered designs.

**Framework Coding Techniques**. The use of traditional object-oriented construction techniques and patterns is typically not restricted when building frameworks under GenVoca. For example, inverting control through the use of hook methods (Fayad and Schmidt, 1997) is common in traditionally constructed frameworks and can easily encapsulated as GenVoca components. Many of the design choices that current framework developers face still need to be addressed when using GenVoca. For instance, the choice between black box and white box framework implementations must still be made when using a layered approach.

## 7 CONCLUSIONS

A framework is an object-oriented code-structuring technique that seems ideal for product-lines. The classes of a framework encapsulate the common algorithms that arise in a family of related applications. A particular application of a product-line is created by defining an instance of this framework, i.e., supplying concrete subclasses of framework classes to provide the necessary customizations. While frameworks are indeed useful, we and others have noticed that frameworks fail miserably in the very common case of optional features. Framework classes can vary (which leads to framework proliferation); classes of different framework instances can have much in common (which leads to code replication).

The core of these problems is that frameworks exhibit a rather inflexible design: the delineation between the content of framework classes and classes of framework instances is fixed. Application features are either hard-coded into framework classes or hard-coded into instance classes. In this paper, we have outlined a different approach for product-line implementation. We have presented a component-based model that reveals the building blocks of frameworks and framework instances. Our components allow application designers to define the set of features that they want both in their framework classes and instance classes. If features need to be changed, our model supports this by adding, swapping, or removing components from previously defined compositions. In general, the application product-lines that we can express with our model is much more varied with far less code replication than that which can be expressed by frameworks.

The essence of our approach is understanding software in terms of object-oriented collaborations or refinements, and creating a parametric model of product-lines that is based on refinement/collaboration composition. Many product-lines have been built using this approach before; the contribution of this paper is demonstrating that this approach offers significant advantages over frameworks in building application product-lines.

*Acknowledgments.* We thank Mike Kistler for his comments on earlier drafts of this paper.

## 8 REFERENCES

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

D. Batory, V. Singhal, M. Sirkin, and J. Thomas, “Scalable Software Libraries”, *ACM SIGSOFT 1993*.

D. Batory and B.J. Geraci, “Component Validation and Subjectivity in GenVoca Generators”, *IEEE Trans. Software Engineering*, February 1997.

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

D. Batory, “Product-Line Architectures”, Invited presentation, Smalltalk und Java in Industrie and Ausbildung, Erfurt, Germany, October 1998.

D. Batory, C. Johnson, R. MacDonald, and D. von Heeder, “Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study”, *International Conference on Software Reuse*, Vienna, Austria, June 2000.

T. Biggerstaff, “The Library Scaling Problem and the Limits of Concrete Component Reuse”, *International Conference on Software Reuse*, Rio de Janeiro, November 1-4, 1994, 102-110.

J. Bosch, “Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study”, *Software Architecture*, Kluwer Academic Publishers, 1999.

G. Bracha and W. Cook, “Mixin-Based Inheritance”, *ECOOP/OOPSLA 90*, 303-311.

G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler, “Making the future safe for the past: Adding Genericity to the Java Programming Language”, *OOPSLA 98*, Vancouver, October 1998.

W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen, “From Custom Applications to Domain-Specific Frameworks”, *Communications of the ACM*, 40(10), October 1997.

S. Cohen and L. Northrop, “Object-Oriented Technology and Domain Analysis”, *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

K. Czarnecki and U.W. Eisenecker, “Components and Generative Programming”, *SIGSOFT 1999*, LNCS 1687, Springer-Verlag, 1999.

J-M. DeBaud and K. Schmid, “A Systematic Approach to Derive the Scope of Software Product Lines”, *ICSE 99*.

A. Duncan and U. Hölzle, “Load-Time Adaptation: Efficient and Non-Intrusive Language Extension for Virtual Machines”, Technical Report TRCS99-09, University of California, Santa Barbara, 1999.

M. Fayad and D. Schmidt, “Object-Oriented Application Frameworks”, *Communications of the ACM*, 40(10), October 1997.

R.B. Findler and M. Flatt, “Modular Object-Oriented Programming with Units and Mixins”, *ICFP 98*.

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

J.A. Goguen, “Reusing and Interconnecting Software Components”, *IEEE Computer*, February 1986.

H. Gomaa et. al., “A Prototype Domain Modeling Environment for Reusable Software Architectures”, *3rd International Conference on Software Reuse*, Rio de Janeiro, November 1-4, 1994, 74-83.

A.N. Habermann, L. Flon, and L. Cooprider, “Modularization and Hierarchy in a Family of Operating Systems”, *CACM*, May 1976.

M.G. Hayden, “The Ensemble System”, Ph.D. dissertation, Dept. Computer Science, Cornell, January 1998.

J.S. Heidemann and G.J. Popek, “File-System Development with Stackable Layers”, *ACM Transactions on Computer Systems*, 12(1), 58-89, 1994.

R. Johnson and B. Foote, “Designing Reusable Classes”, *Journal of Object-Oriented Programming*, 1(2): June/July 1988, 22-35.

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

D. McIlroy, “Mass Produced Software Components”, *Software Engineering: Report on a Conference by the Nato Science Committee*, October 1968, P. Naur and B. Randell, eds. 138-150.

M. Odersky and P. Wadler, “Pizza into Java: Translating Theory into Practice”, *ACM Symposium on Principles of Programming Languages 1997*, 146-159.

D.L. Parnas, “On the Design and Development of Program Families”, *IEEE Transactions on Software Engineering*, March 1976.

H. Partsch and R. Steinbruggen, “Program Transformation Systems”, *Computing Surveys*, March 1983.

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.

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

M. VanHilst and D. Notkin, “Using C++ Templates to Implement Role-Based Designs”, *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.

D.M. Weiss and C.T.R. Lai, *Software Product-Line Engineering*, Addison-Wesley, 1999.