Building Product-Lines with Mixin-Layers[1]

Don Batory and Yannis Smaragdakis
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

A *mixin-layer* is a building block for assembling applications of a product-line. We explain mixin-layers, their relationship to collaboration-based designs, layered designs, and GenVoca. We also summarize some of the product-lines that we have built using mixin-layers.

1 Introduction

A *product-line architecture (PLA)* is a design for a family of related applications. Our interest in PLAs stems from our earlier work in scalable libraries and software generators [Bat93-94]. A *generator* converts a high-level, declarative specification into a high-performance application. It is a tool—actually a compiler or configuration manager—that can produce a family or *product-line* of related applications. A *scalable library* is a small set of components that can be composed in exponential numbers of ways, where a product-line application is a particular composition of these components. Generators that rely on scalable libraries to produce application product-lines are called *GenVoca* generators.

At its core, GenVoca is a design methodology for building architecturally-extensible software—i.e., software that is extensible via component addition and removal. To the contrary of its historical development, GenVoca can be understood as a scalable outgrowth of an old, largely unscalable, and mostly 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, thousands, or millions of small refinements) [Bat92, Boe96].

When GenVoca refinements are composed statically, their implementation can be expressed as an object-oriented (OO) building block which we call a *mixin-layer*. Among the benefits of expressing GenVoca refinements as mixin-layers is that (1) it presents a very different way in which OO designs can expressed, (2) it presents a clean way in which to create OO PLAs from mixin-layers, and (3) it reveals a fundamental connection with an important area of OO design called collaboration-based designs. In the following sections, we sketch the relationship of collaboration-based designs with GenVoca refinements, introduce the idea of mixin-layers for creating mix-and-match application architectures, and review PLAs that we have built using mixin-layers.

2 Refinements and Collaboration-Based Designs

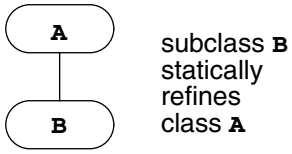
In an object-oriented design, objects are encapsulated entities that are rarely self-sufficient. Although an object is fully responsible for maintaining the data it encapsulates, it needs to cooperate with other objects

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.

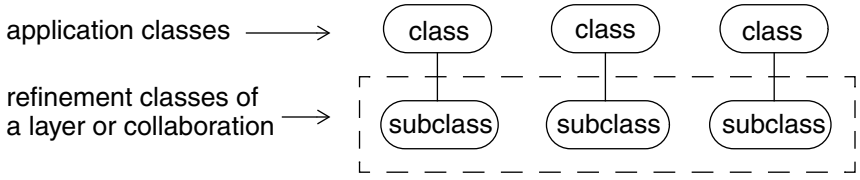
to complete a task. An interesting way to encode object interdependencies is through collaborations. A *collaboration* is a set of objects and a protocol (i.e., a set of allowed behaviors) that determines how these objects interact. The part of an object that enforces the protocol that a collaboration prescribes is called the object's *role* in the collaboration.

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.

Static Refinements. GenVoca refinements and collaboration-based designs have much in common: Object classes are of secondary importance and components (collaborations) interrelate many classes. To build even one class of an application, several components (collaborations) must be combined. The central question is how are collaborations expressed in OO languages? Consider how static refinements are expressed in OO. 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 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).

> To give this intuitive meaning, 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 a primitive application building block. This idea is called an *aspect* in *Aspect-Oriented Programming (AOP)*, although GenVoca [Bat92] predates AOP [Kic97].

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 shortly 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 instantiation 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 phenomenon. 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.

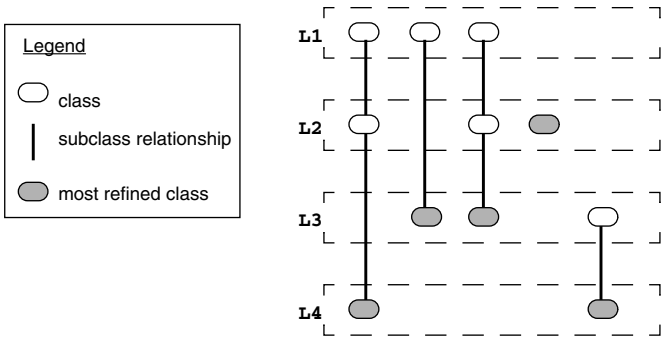


Figure 1 Creating Inheritance Hierarchies by Composing Layers

Each inheritance "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, each of which 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. The non-terminal (non-shaded) subclasses represent intermediate derivations of the application classes. Thus, the composition of layers **`L1`** through **`L4`** yields five classes (i.e., those that are shaded); the unshaded classes represent the "intermediate" derivations of these shaded classes. Even though the resulting class hierarchies can be complex, 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. Layers are basic building blocks for large families of applications, where an *application* is a composition of layers. In general, *n* layers can be composed in excess of *n*! ways, because the order of composition matters and layer replication is possible. This is the central idea behind GenVoca.

(So it is not uncommon that rather different applications of a product-line can be assembled by composing exactly the same layers in different orders [Bat92, Hay98]. This can be seen in Figure 1: the order of **`L2`**-**`L4`** could be permuted, provided that **`L4`** is "below" **`L3`**.)

3 Mixin Layers

We now consider how collaboration-based designs can be implemented. We use *Jak*, a superset of Java that adds parameterization, to convey the basic implementation technique, called *mixin-layers*. C++ expressions of mixin-layers are presented elsewhere [Sma98a-b].

Mixins. A *mixin* is a class whose superclass is specified by a parameter. Mixins can be expressed as templates. Below we define a mixin **`M`** whose superclass is defined by parameter **`S`**:[2,3]

2. **`S`** can be any class since all classes implement interface **`AnyClass`**.

3. We use Bracha's general definition of "mixin", which is a class whose superclass is left unspecified [Bra90]. C++ has evolved a different meaning of "mixin" that is *not* equivalent to our use.

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

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

Nested Classes. In both Java and *Jak*, class declarations can be nested inside other class declarations. Nested classes behave in most respects (e.g., access control, scoping) just like regular members of a class. Interestingly enough, nested classes can also be inherited. Consider the following example:

```
class OuterParent { class Inner { ... } }
class OuterChild extends OuterParent { }
```

In this case, **OuterChild** is a subclass of **OuterParent** in an inheritance hierarchy. Although no **OuterChild.Inner** class is explicitly defined, such a class does, in fact, exist as it is inherited from **OuterParent**. 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 as an *Jak* template that has *n+1* parameters: one parameter **S** that defines the superclass of **M**, plus *n* additional parameters that define the specific classes the collaboration's role classes are to refine.

```
class M <AnyClass S, AnyClass r1, AnyClass r2, ... AnyClass rn>
extends S {
   class role1 extends r1 { ... }
   class role2 extends r2 { ... }
   ...
   class rolen extends rn { ... }
}
```

Experience has shown that different collaborations often 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 common and compact form 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 role2 extends S.role2 { ... }
   ...
   class rolen extends S.rolen { ... }
}
```

Note the three mechanisms that we exploit: mixins (i.e., parameterized inheritance), nested classes (for encapsulating multiple classes within a single unit), and name standardization (for consistent role names across multiple collaborations).

Compositions. Collaborations 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 **T** with the general form:

```
class T extends
     Collab1 < Collab2 < Collab3 < ... < FinalCollab > ... >
```

That is, **Collab1**, **Collab2**, …, **FinalCollab** are mixin-layers, "**<...>**" is the *Jak* operator for template instantiation, and **T** is the name given to the class that is produced by this composition. The classes of **T** are referenced in the usual way, namely $\mathbf{T.Role}_1$ defines the application class $\mathbf{Role}_1$, etc.

4 Applications

The *Jakarta Tool Suite (JTS)* is a set of Java-based compiler tools for building domain-specific languages and embedding domain-specific languages into Java. JTS has been used to create a product-line of Java dialects, of which Jak (mentioned in previous sections) is an example. JTS and Jak have been built from mixin-layers. Jak itself is a composition of 10 mixin-layers that encapsulate over 500 classes [Bat98].

We are now re-engineering the *Fire Support Automated Test System (FSATS),* a command-and-control simulator for the U.S. Army, as a GenVoca product-line where primitive building blocks are mixin-layers. Preliminary results show that using mixin-layers provides a substantial simplification to FSATS design and makes it possible to create variations of FSATS (e.g., for use by other armed-forces) that would otherwise be infeasible.

The techniques and results discussed in this paper are not isolated; GenVoca itself has a long history of successes, including avionics [Bat95], data structures [Bat93], and network protocols [Hay98]. Among the technical issues previously addressed are the verification of layer composition, automatic optimization of compositions, and meta-program implementation of refinements. Although mixin-layers were conceived only recently, many earlier GenVoca PLAs could have been equivalently (and more cleanly) expressed in terms of these ideas.

5 References

[Bat92] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components". *ACM Transactions on Software Engineering and Methodology,* 1(4):355-398, October 1992.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[Bat94] D. Batory, J. Thomas, and M. Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler", *ACM SIGSOFT 1994*.

[Bat95] D. Batory, L. Coglianese, M. Goodwill, and S. Shafer. "Creating Reference Architectures: An Example from Avionics". *Symposium on Software Reusability,* Seattle Washington, April 1995.

[Bat98] D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: Tools for Implementing Domain-Specific Languages". *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.

[Boe96] E. Boerger and I. Durdanovic, "Correctness of Compiling Occam to Transputer Code", *The Computer Journal*, Vol. 39, No. 1.

[Bra90] G. Bracha and W. Cook, "Mixin-Based Inheritance", *ECOOP/OOPSLA 90*, 303-311.

[Hay98] M.G. Hayden, "The Ensemble System", Ph.D. dissertation, Dept. Computer Science, Cornell, January 1998.

[Kic97b] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin, "Aspect-Oriented Programming", *ECOOP 97*, 220-242.

[Sma98b] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers". *12th European Conference on Object-Oriented Programming, (ECOOP '98)*, July 1998.

[Sma98a] Y. Smaragdakis and D. Batory, "Implementing Reusable Object-Oriented Components." *5th International Conference on Software Reuse*, Victoria, Canada, June 1998.