Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs

YANNIS SMARAGDAKIS
Georgia Institute of Technology
and
DON BATORY
The University of Texas at Austin

A "refinement" is a functionality addition to a software project that can affect multiple dispersed implementation entities (functions, classes, etc.). In this paper, we examine large-scale refinements in terms of a fundamental object-oriented technique called collaboration-based design. We explain how collaborations can be expressed in existing programming languages or can be supported with new language constructs (which we have implemented as extensions to the Java language). We present a specific expression of large-scale refinements called *mixin layers*, and demonstrate how it overcomes the scalability difficulties that plagued prior work. We also show how we used mixin layers as the primary implementation technique for building an extensible Java compiler, JTS.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures; D.2.13 [**Software Engineering**]: Reusable Software; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Design, Languages

Additional Key Words and Phrases: Collaboration-based design, component-based software, product-line architectures

1. INTRODUCTION

The history of software design and programming languages intimately evolves around the concept of modularity. Modules encapsulate primitive functionality or services that, ideally, can be reused in the construction of many applications. The granularity of modules has evolved from small to medium scale, and now to large-scale—from functions, to abstract data types or classes (i.e., suites of interrelated functions), and now, more commonly, to components or packages (i.e.,

The authors gratefully acknowledge funding by 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.
Authors' addresses: Yannis Smaragdakis, College of Computing, Georgia Tech, Atlanta, GA 30332; Don Batory, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2002 ACM 1049-331X/02/0400–0215 $5.00

suites of interrelated classes). The benefit of increased module scale is that of economics—applications are easier to build from fewer and larger parts—and design simplicity—applications are easier to comprehend when modules encapsulate, and thus hide, irrelevant implementation details.

The benefits of scaled modularity, however, are driven by reuse. The more a module is reused, the more valuable it becomes. But there is an ironic twist: the larger the module, the more specific its use and functionality, and this, in turn, reduces the likelihood that other applications will need its exact capabilities. In other words, it seems that reuse opportunities become fewer as a module becomes larger: scaling modularity seems to defeat the purpose of reuse, and this is exactly the opposite of what we want [Biggerstaff 1994].

The solution to this problem lies in a very different concept of modularity, where neither entire functions, entire classes, nor entire packages are the answer. Instead, the unit of modularity that we seek, encapsulates *fragments* of multiple classes, which in turn encapsulate *fragments* of multiple functions. An extensive body of research has shown that such units are indeed the reusable building blocks of large-scale modules; composing sets of class fragments yields a package of fully-formed classes. This recognition has become particularly clear in the area of software product-lines, where the goal is to construct large families of related applications from primitive and reusable components. The components that made this possible encapsulated fragments of classes.

We use the term *refinement* (also in [Batory and Geraci 1997]) for any such unit of functionality in a software system. A refinement is a functionality addition to a program, which introduces a conceptually new service, capability, or feature, and may affect multiple implementation entities. Various researchers have offered different descriptions, implementations, and names to fairly analogous concepts over the years: layers [Batory et al. 1988], collaborations [Reenskaug et al. 1992], subjects [Ossher and Harrison 1992; Tarr et al. 1999] and aspects [Kiczales et al. 1997]. Parnas's [1979] classic work has offered much of the software engineering context for these approaches.[1]

We believe that scalability is a prominent characteristic of successful refinement technologies. Implementing microscopic refinements (i.e., refinements that dealt with code fragments at the expression level) has not produced great software engineering advances in the past, and is unlikely to do so in the future. The novelty of current research strikes at the core problem—that of scaling the unit of refinement from a microscopic to a large scale where a single

[1]The definition of "refinement" that seems closest to our intended meaning is "the act of making improvement by introducing subtleties or distinctions" (Merriam-Webster's Dictionary). Formal approaches to programming use the term "refinement" to denote the elaboration of a program by adding more implementation detail until a fully concrete implementation is reached. The set of behaviors (i.e., the legal variable assignments) of a "refined" program is a subset of the behaviors of the original "unrefined" program. This appears to be different from our use of the term. Our "refinements" follow the dictionary definition by adding "subtleties or distinctions" at the *design* level. At the implementation level, however, a refinement can yield dramatic changes: both the exported functionality (semantics of operations) and the exported interface (signatures of operations) may change. Thus, unlike the use of "refinement" in formal approaches to programming, the set of allowed behaviors of our "refined" program might not be a subset of the behaviors of the "unrefined" program.

refinement alters multiple classes of an application. A large-scale refinement exhibits "cross-cutting"—multiple classes must be updated simultaneously and consistently. Thus, composing a few large scale refinements yields an entire application. This means that the inverse relationship between module size, and reusability, which has crippled conventional concepts, no longer applies, and a fresh look at software modularity has become a topic of wide-spread interest.

This paper is about modular implementations of large-scale refinements, and the development of families of related applications through refinement. In particular, we show that a fundamental object-oriented concept, called *collaboration-based* designs, is in fact how large-scale refinements are expressed in object-oriented models. We begin by explaining the core ideas of collaboration-based design, and how they are related to large-scale refinements. We then show how these ideas can be expressed in existing programming languages, or supported with new language constructs (which we have implemented as extensions of the Java language). We introduce a specific expression of large-scale refinements called *mixin layers*, and demonstrate how it extends and overcomes problems of prior work on the refinement-based designs of VanHilst and Notkin [1996a, 1996b, 1996c, 1997] and application frameworks [Johnson and Foote 1988]. Mixin layers implementations are discussed, but we intend to convince the reader that one *should* implement programs using mixin layers, not that one *is merely able to* do so. Although better implementations than the ones we propose may be possible, or languages other than the ones we examine may offer more complete support for mixin layers, this would not alter our main argument, which is that application development through mixin layers is desirable. As a practical validation, we show how we used mixin layers as the primary implementation technique in a medium-size project: the JTS tool suite for implementing domain-specific languages. Our experience shows that the mechanism is versatile and can handle refinements of substantial size.

2. BACKGROUND: COLLABORATION BASED DESIGNS

Collaboration-based, or *role-based*, designs have been the subject of many papers [Cunningham and Beck 1989; Helm et al. 1990; Holland 1992; Reenskaug et al. 1992; VanHilst and Notkin 1996b]. The concept may have originated with Reenskaug, et al. [1992], but the ideas have been used in various forms, often without being named (e.g., [Batory et al. 1988]). A good introduction to collaboration-based design can be found in the presentation of the OORAM approach [Reenskaug et al. 1992] A detailed treatment of collaboration-based designs, together with a discussion of how to derive them from use-case scenarios [Rumbaugh 1994] can be found in VanHilst's [1997] Ph.D. dissertation.

2.1 Collaborations and Roles

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

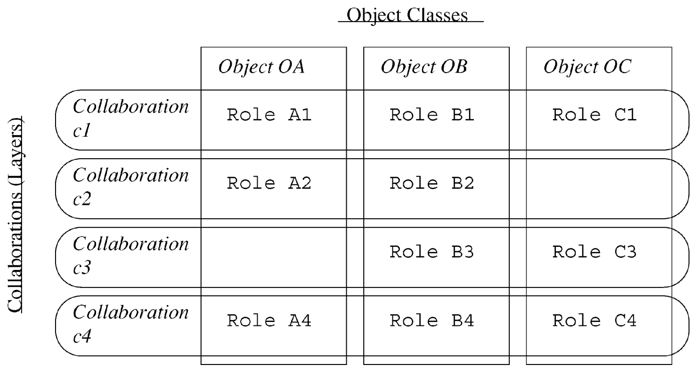

Fig. 1. Example collaboration decomposition. Ovals represent collaborations; rectangles represent objects; their intersections represent roles.

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. Objects of an application generally participate in multiple collaborations simultaneously, and thus may encode several distinct roles. Each collaboration, in turn, is a collection of roles, and represents relationships across corresponding objects. A role isolates the part of an object that is relevant to a collaboration, from the rest of the object. Different objects can participate in a collaboration, as long as they support the required roles.

In collaboration-based design, the objective is to express an application as a composition of largely independently-definable collaborations. *Viewed in terms of design modularity, collaboration-based design acknowledges that a unit of functionality (module), is neither a whole object nor a part of it, but can cross-cut several different objects.* If a collaboration is reasonably independent of other collaborations (i.e., a good approximation of an ideal module), the benefits are great. First, the collaboration can be reused in a variety of circumstances where the same functionality is needed, by just mapping its roles to the right objects. Second, any changes in the encapsulated functionality will only affect the collaboration, and will not propagate throughout the whole application.

In abstract terms, a collaboration is a view of an object-oriented design from the perspective of a single concern, service, or feature. For instance, a collaboration can be used to express a producer-consumer relationship between two communicating objects. Clearly, this collaboration prescribes roles for (at least) two objects and there is a well-defined "protocol" for their interactions. Interestingly, the same collaboration could be instantiated more than once in a single object-oriented design, with the same objects playing different roles in every instantiation. In the example of the producer-consumer collaboration, a single object could be both a producer (from the perspective of one collaboration) and a consumer (from the perspective of another).

Figure 1 depicts the overlay of objects and collaborations in an abstract application involving three different objects (*OA*, *OB*, *OC*), each supporting multiple roles. Object *OB*, for example, encapsulates four distinct roles: `B1`, `B2`, `B3`, and `B4`. Four different collaborations (*c1*, *c2*, *c3*, *c4*) capture distinct aspects of the application's functionality. Each collaboration prescribes roles to certain objects. For example, collaboration *c2* contains two distinct roles, `A2` and `B2`, which are assumed by distinct objects (namely *OA* and *OB*). An object does not need to play a role in every collaboration—for instance, *c2* does not affect object *OC*.

Collaborations can be composed dynamically at application run-time, or statically at application compile-time. In this paper, we examine the static composition of collaborations, where roles that are played by an object are uniquely determined by its class. For instance, in Figure 1, all three objects must belong to different classes (since they all support different sets of roles). The work described in this paper can be generalized to dynamic compositions of collaborations.

From a broader perspective, a collaboration is a large-scale refinement. Again, a refinement elaborates a program to extend its functionality or to add implementation details. A refinement is large scale if it modifies multiple classes of an application. For example, when collaboration *c4* is (statically) added to the program of Figure 1, the classes for objects *OA*, *OB*, and *OC* are updated consistently and simultaneously, so that the "feature" or "service" defined by *c4* is appropriately implemented. Thus, composing collaborations is an example of refinement, where a simple program is progressively elaborated into a more complex one. Collaborations are large-scale and reusable refinements—they can be used in the construction of many programs.

2.2 An Example

As a running example that illustrates important points of our discussion, we consider a graph traversal application that was initially examined by Holland [1992], and subsequently by VanHilst and Notkin [1996b]. This affords a historical perspective on the development of collaboration-based designs, and a perspective on the contribution of this work. The application defines three different operations (algorithms) on an undirected graph, all based on depth-first traversal: *Vertex Numbering* numbers all nodes in the graph in depth-first order, *Cycle Checking* examines whether the graph is cyclic, and *Connected Regions* classifies graph nodes into connected graph regions. A client of this application can instantiate a graph and separately invoke algorithms that perform vertex numbering, cycle checking, and/or find connected regions on a graph. The application itself has three distinct classes: *Graph*, *Vertex*, and *Workspace*. The *Graph* class describes a container of nodes with the usual graph properties. Each node is an instance of the *Vertex* class. Finally, the *Workspace* class includes the application part that is specific to each graph operation. For example, the *Workspace* object for a *Vertex Numbering* operation holds the value of the last number assigned to a vertex as well as the methods used to update this number.

In decomposing this application into collaborations, we need to capture distinct aspects as separate collaborations. A decomposition of this kind is straightforward and results in five distinct collaborations.

One collaboration (*Undirected Graph*) encapsulates properties of an undirected graph. This is clearly an independent aspect of the application—the problem could very well be defined for directed graphs, for trees, and so on.

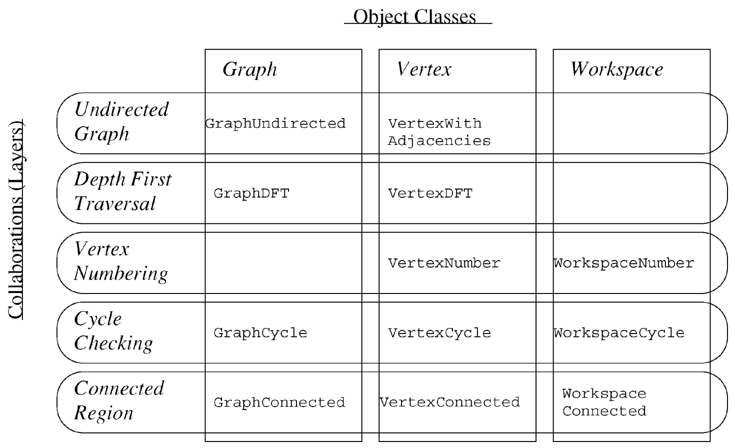

Fig. 2. Collaboration decomposition of the example application domain: A depth-first traversal of an undirected graph is specialized to yield three different graph operations. Ovals represent collaborations, rectangles represent classes.

Another collaboration (*Depth First Traversal*) encapsulates the specifics of depth-first traversals and provides a clean interface for extending traversals. That is, at appropriate moments during a traversal (the first time a node is visited, when an edge is followed, and when a subtree rooted at a node is completely processed) control is transferred to specialization methods that can obtain information from the traversal collaboration, and supply information to it. Consider the *Vertex Numbering* operation as a refinement of a depth-first traversal. Numbering is realized by specializing the action when visiting a node for the first time during a traversal. The action assigns a number to the node and increases the count of visited nodes.

Using this approach, each of the three graph operations can be seen as a refinement of a depth-first traversal, and each can be expressed by a single collaboration. Figure 2 is reproduced from VanHilst and Notkin [1996b] and presents the collaborations and classes of our example application domain. The intersection of a class and a collaboration in Figure 2 represents the role prescribed for that class by the collaboration. A role encodes the part of an object that is specific to a collaboration. For instance, the role of a *Graph* object in the "*Undirected Graph*" collaboration supports storing and retrieving a set of vertices. The role of the same object in the "*Depth First Traversal*" collaboration implements a part of the depth-first traversal algorithm. (In particular, it contains a method that initially marks all vertices of a graph *not-visited* and then calls the method for depth-first traversal on each graph vertex object.)

The goal of a collaboration-based design is to encapsulate within a collaboration, all dependencies between classes that are specific to a particular service or feature. In this way, collaborations themselves have no outside dependencies and can be reused in a variety of circumstances. The "*Undirected Graph*" collaboration, for instance, encodes the properties of an undirected graph (pertaining to the *Graph* and *Vertex* classes, as well as the interactions between objects of the two). Thus, it can be reused in any application that deals with undirected graphs. Ideally, if we could define an "interface" to a collaboration, we should also be able to easily replace one collaboration with another that exports the same interface. For instance, it would be straightforward to replace the "*Undirected Graph*" collaboration with one representing a directed graph, assuming that both collaborations exported the same interface.

Of course, simple interface conformance does not guarantee composition correctness—the application writer must ensure that the algorithms used (for example, the depth-first traversal) are still applicable after the change. The algorithms presented by Holland [1992] for this example are general enough to be applicable to a directed graph. If, however, a more efficient, specialized-for-undirected-graphs algorithm were used (as is, for instance, possible for the *Cycle Checking* operation) the change would yield incorrect results. Smaragdakis [1999], Smaragdakis and Batory [1998], and Batory and Geraci [1997] discuss, in detail, the issue of ensuring that collaborations are actually interchangeable.

Although we have focussed on a single application that supports all three graph operations, it is easy to see how variants of this application could be created (e.g., by omitting or adding operations), where each variant would be described by the use of different collaborations. This very fact makes collaboration-based designs ideal for describing product-line architectures, that is, designs for families of related applications. As we will see, collaborations define the building blocks for application families; compositions of these building blocks yield different product-line members.

3. IMPLEMENTING COLLABORATION-BASED DESIGNS WITH MIXIN LAYERS

3.1 Mixin Classes and Mixin Layers

A refinement of an object-oriented class is encapsulated by a subclass—a subclass can add new methods and data members, as well as override existing methods of its superclass. Thus, inheritance is a built-in mechanism for statically refining classes in object-oriented languages. The challenge is to scale inheritance from refining individual classes to expressing the large-scale refinements of collaboration-based designs.

A solution is to build on an existing object-oriented construct called a *mixin*. Mixins are similar to classes but with some added flexibility. Unfortunately, mixins alone are not sufficient to express large-scale refinements—they suffer from being able to refine only a single class at a time, not a collection of cooperating classes. To address this, we introduce *mixin-layers*: a scaled-up form of mixins that can contain multiple smaller mixins.

3.1.1 *Introduction to Mixins.* The term *mixin class* (or just "mixin") has been overloaded to mean several specific programming techniques and a general mechanism that they all approximate. Mixins were originally explored in the context of the Lisp language with object-systems like Flavors [Moon 1986] and CLOS [Kiczales et al. 1991]. They were defined as classes that allow their superclass to be determined by *linearization* of multiple inheritance. In C++, the term has been used to describe classes in a particular (multiple) inheritance arrangement: as superclasses of a single class that themselves have a common *virtual base class* (see Stroustrup [1997], p. 402). Both of these mechanisms are approximations of a general concept described by Bracha and Cook [1990]. Here we use "mixin" in this general sense.

The main idea of mixins is simple: in object-oriented languages, a superclass can be defined without specifying its subclasses. This property is not, however, symmetric: when a subclass is defined, it must have a specific superclass. Mixins (also commonly known as *abstract subclasses* [Bracha and Cook 1990]) represent a mechanism for specifying classes that eventually inherit from a superclass. This superclass, however, is not specified at the site of the mixin's definition. Thus a single mixin can be instantiated with different superclasses yielding widely varying classes. This property makes them appropriate for defining uniform incremental extensions for a multitude of classes. When a mixin is instantiated with one of these classes as a superclass, it produces a class incremented with the additional behavior.

Mixins can be implemented using parameterized inheritance (a class whose superclass is specified by a parameter). Using C++ syntax we can write a mixin as:

```
template <class Super> class Mixin : public Super {
  ... /* mixin body */
};
```

Mixins are flexible and can be applied in many circumstances without modification. To give an example, consider a mixin implementing *operation counting* for a graph. Operation counting means keeping track of how many nodes and edges have been visited during the execution of a graph algorithm. (This simple example is one of the non-algorithmic refinements to algorithm functionality discussed in Weihe [1997].) This mixin could have the form:[2]

```
template <class Graph> class Counting: public Graph {
  int nodes_visited, edges_visited;
public:
  Counting() : Graph() { nodes_visited = edges_visited = 0; }

  node succ_node (node v) {
    nodes_visited++;
    return Graph::succ_node(v);
  }

  edge succ_edge (edge e) {
    edges_visited++;
    return Graph::succ_edge(e);
  }
```

[2]We use C++ syntax for most of the examples of this section, in the belief that concrete syntax clarifies, rather than obscures, our ideas. To facilitate readers with limited C++ expertise, we avoid several cryptic idioms or shorthands (for instance, constructor initializer lists are replaced by assignments, we do not use the "struct" keyword to declare classes, etc.).

```
  // example method that displays the cost of an algorithm in
  // terms of nodes visited and edges traversed
  void report_cost () {
    cout << "The algorithm visited " << nodes_visited <<
            " nodes and traversed " << edges_visited <<
            " edges\n";
  }
  ... // other methods using this information may exist
};
```

By expressing operation counting as a mixin, we ensure that it is applicable to many classes that have the same interface (i.e., many different kinds of graphs). The implicit assumption is that classes, like `Dgraph` and `Ugraph`, have been designed so that they export similar interfaces. By standardizing certain aspects of the design, like the method interfaces for different kinds of graphs, we gain the ability to create mixin classes that can be reused in different occasions.[3] We can, for instance, use two different compositions:

```
typedef Counting < Ugraph > CountedUgraph;
```

and

```
typedef Counting < Dgraph > CountedDgraph;
```

to define a counted undirected graph type and a counted directed graph type. (We omit parameters to the graph classes for simplicity.) Note that the behavior of the composition is exactly what one would expect: any methods not affecting the counting process are exported (inherited from the graph classes). The methods that do need to increase the counts are "wrapped" in the mixin.

3.1.2 *Mixin Layers.* To implement entire collaborations as components, we need to use mixins that encapsulate other mixins. We call the encapsulated mixin classes *inner mixins*, and the mixin that encapsulates them the *outer mixin*. Inner mixins can be inherited, just as any member variables or methods of a class. An outer mixin is called a *mixin layer* when *the parameter* (*superclass*) *of the outer mixin encapsulates all parameters* (*superclasses*) *of inner mixins*.[4] This is illustrated in Figure 3. `ThisMixinLayer` is a mixin that refines (through inheritance) `SuperMixinLayer`. `SuperMixinLayer` encapsulates three classes: `FirstClass`, `SecondClass`, and `ThirdClass`. `ThisMixinLayer` also encapsulates three inner classes. Two of them are mixins that refine the corresponding classes of `SuperMixinLayer`, while the third is an entirely new class.

[3]Stated another way, a mixin defines a refinement of a class, but this refinement is not meaningful for every possible class. Standardized interfaces is a way to type or restrict the set of classes that a mixin can meaningfully refine. C++ syntax, in this regard, is unsatisfactory because C++ templates have untyped parameters. Languages like Pizza [Odersky and Wadler 1997] or GJ [Bracha et al. 1998] offer a better mechanism, where class parameters are typed by the interfaces that they implement. Unfortunately, Pizza and GJ do not support parameterized inheritance.
[4]Inner mixins can themselves be mixin layers.

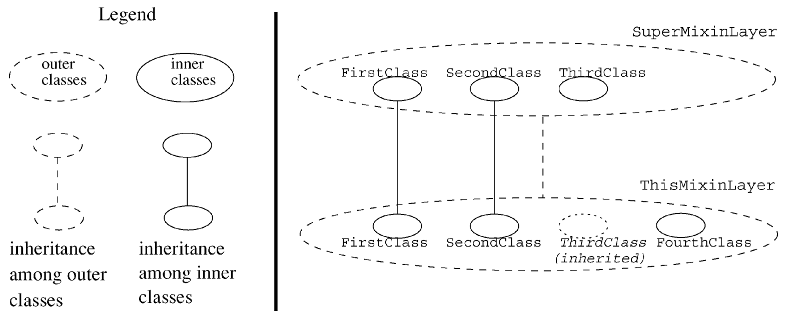

Fig. 3. Mixin layers schematically.

Inheritance works at two different levels. First, a layer can inherit inner classes from the layer above it (for instance, `ThirdClass` in Figure 3). Second, the inner mixins inherit member variables, methods, or other classes from their superclass.

3.1.3 *Mixin Layers in Various OO Languages.* The mixin layer concept is quite general and is not tied to any particular language idiom. Many flavors of the concept, however, can be expressed via specific programming language idioms: as stand-alone language constructs, as a combination of C++ nested classes and parameterized inheritance, as a combination of CLOS class-metaobjects and mixins, and so on. We next examine some of these different realizations. The introduction of technical detail is necessary at this point, as it helps us demonstrate concretely, in Section 3.2 , the advantages of mixin layers for implementing collaboration-based designs.

C++. We would like to support mixin layers in C++, using the same language mechanisms as those used for mixin classes. To do this, we can standardize the names used for inner class implementations (make them the same for all layers). This yields an elegant form of mixin layers that can be expressed using common C++ features. For instance, using C++ parameterized inheritance and nested classes, we can express `ThisMixinLayer` as a mixin layer (see again Figure 3) with two inner mixins (`FirstClass` and `SecondClass`) and one additional class (`FourthClass`):

```
template <class LayerSuper>
class ThisMixinLayer: public LayerSuper {
public:
  class FirstClass  : public LayerSuper::FirstClass  { ... };
  class SecondClass : public LayerSuper::SecondClass { ... };
  class FourthClass                                  { ... };
  ...
};
```

The above code fragment represents the form of mixin layers that we use in the examples of this section. Note that specifying a parameter for the outermost mixin automatically determines the parameters of all inner mixins. Composing mixin layers to form concrete classes is now as simple as composing mixin classes. If we have four mixin layers (`Layer1`, `Layer2`, `Layer3`, `Layer4`), we can compose them as:

```
Layer4 < Layer3 < Layer2 < Layer1 > > >
```

where "<. . .>" is the C++ operator for template instantiation. Note that `Layer1` has to be a concrete class (i.e., not a mixin class). Alternatively we can have a class with empty inner classes that is the root of all compositions. (A third alternative is to use a *fixpoint* construction and instantiate the topmost layer with the result of the entire composition. This pattern has several desirable properties and is analyzed further in Chapter 3 of Smaragdakis [1999].)

In the above code fragment, we mapped the main elements of the mixin layer definition to specific implementation techniques. We used nested classes to implement class encapsulation. We also used parameterized inheritance to implement mixins. However, there are very different ways of encoding the same concept in other languages.

CLOS (and other reflective languages). We can encode mixin layers in CLOS [Kiczales et al. 1991] (and other reflective systems) by simulating their main elements using reflection (classes as first-class entities). Because of lack of space, we elide the implementation specifics. A discussion can be found in Smaragdakis and Batory [1998] and Smaragdakis [1999]. CLOS mixin layers are not semantically equivalent to C++ mixin layers (for instance, there is no default class data hiding: class members are by default accessible from other code in CLOS). Nevertheless, the two versions of mixin layers are just different flavors of the same idea.

Our ideas are applicable to other reflective languages. Smalltalk, in particular, has been a traditional test-bed for mixins, both for researchers (e.g., Bracha and Griswold [1996], Mezini [1997], and Steyaert et al. [1993]) and for practitioners [Montlick 1996]. A straightforward (but awkward) way to implement mixins in Smalltalk is as *class-functors*; that is, mixins can be functions that take a superclass as a parameter and return a new subclass.

Java. The Java language is an obvious next candidate for mixin layers. Java has no support for mixins and it is unlikely that the core language will include mixins in the near future. As will be described in Section 4, we extended the Java language with constructs that capture mixins and mixin layers explicitly. In this effort we used our JTS set of tools [Batory et al. 1998] for creating compilers for domain-specific languages. The system supports mixins and mixin layers through parameterized inheritance and class nesting, in much the same way as in C++.[5] Additionally, the fundamental building blocks of JTS itself were expressed as mixin layers, resulting in an elegant bootstrapped implementation. More on JTS in Section 4.

[5]The Java 1.1 additions to the language [Sun Microsystems 1997] support nested classes and interfaces (actually both "nested" classes as in C++ and *member* classes—where nesting has access control implications). Nested classes can be inherited just as any other members of a class.

Adding mixins to Java is also the topic of other active research [Agesen et al. 1997; Flatt et al. 1998] (although such work is almost certain to remain in the research domain). The work of Flatt et al. [1998] presented a semantics for mixins in Java. This is particularly interesting from a theoretical standpoint, since it addresses issues of mixin integration in a type-safe framework. As we saw, mixins can be expressed in C++ using parameterized inheritance. There have been several recent proposals for adding parameterization/genericity to Java [Agesen et al. 1997; Odersky and Wadler 1997; Bracha et al. 1998; Myers et al. 1997; Thorup 1997], but only Agesen et al. [1997] supports parameterized inheritance and, hence, can express mixin layers.

It is interesting to examine the technical issues involved in supporting mixins in Java genericity mechanisms. Three of these mechanisms [Odersky and Wadler 1997; Bracha et al. 1998; Thorup 1997] are based on a *homogeneous* model of transformation: the same code is used for different instantiations of generics. This is not applicable in the case of parameterized inheritance—different instantiations of mixins are not subclasses of the same class (see [Agesen et al. 1997] for more details). There may also be conceptual difficulties in adding parameterized inheritance capabilities the genericity approach of [Thorup 1997] is based on virtual types. Parameterized inheritance can be approximated with virtual types by employing *virtual superclasses* [Madsen and Møller-Pedersen 1989], but this is not part of the design of Thorup [1997].

The approaches of Myers et al. [1997] and Agesen et al. [1997] are conceptually similar from a language design standpoint. Even though parameterized implementations do not directly correspond to types in the language (in the terminology of Cardelli and Wegner [1985] they correspond to *type operators*), parameters can be explicitly constrained. This approach, combined with a *heterogeneous* model of transformation (i.e., one where different instantiations of generics yield separate entities) is easily amenable to adding parameterized inheritance capabilities, as was demonstrated in Agesen et al. [1997].

3.2 Implementing Collaboration-Based Designs

Given the mixin layer concept, we can now express collaboration-based designs directly at the implementation level. We show how mixin layers can be used to perform the task and examine how it compares to two previous approaches. One is the straightforward implementation technique of application frameworks using just objects and inheritance. The other is the technique of VanHilst and Notkin that employs C++ mixins to express individual roles.

3.2.1 *Using Mixin Layers.* A collaboration can be expressed by a mixin layer. The roles played by different objects are expressed as nested classes inside the mixin layer. The general pattern is:

```
template <class CollabSuper>
class CollabThis : public CollabSuper {
public:
  class FirstRole  : public CollabSuper::FirstRole  { ... };
  class SecondRole : public CollabSuper::SecondRole { ... };
  class ThirdRole  : public CollabSuper::ThirdRole  { ... };
  ...         // more roles
};
```

Again, mixin layers are composed by instantiating one layer with another as its parameter. This produces two classes that are linked as a parent-child pair in the inheritance hierarchy. For four mixin layers, `Collab1`, `Collab2`, `Collab3`, `FinalCollab` of the above form, we can define a class `T` that expresses the final product of the composition as:

```
     typedef Collab1 < Collab2 < Collab3 < FinalCollab > > > T ;
```

or (alternatively):

```
  class T : public Collab1 < Collab2 < Collab3 < FinalCollab > > >
  { /* empty body */ };
```

In this paper, we consider these two forms to be equivalent.[6]

The individual classes that the original design describes are members (nested classes) of the above components. Thus, `T::FirstRole` defines the application class `FirstRole`, etc. Note that classes that do not participate in a certain collaboration can be inherited from collaborations above (we subsequently use the term "collaboration" for the mixin layer representing a collaboration when no confusion can result). Thus, class `T::FirstRole` is defined even if `Collab1` (the bottom-most mixin layer in the inheritance hierarchy) prescribes no role for it.

Example. Consider the graph traversal application of Section 2.2. Each collaboration is represented as a mixin layer. *Vertex Numbering*, for example, prescribes roles for objects of two different classes: *Vertex* and *Workspace*. Its implementation has the form:

```
template <class CollabSuper>
class NUMBER : public CollabSuper {
public:
  class Workspace : public CollabSuper::Workspace {
  ... // Workspace role methods
  };

  class Vertex : public CollabSuper::Vertex {
  ... // Vertex role methods
  };
};
```

Note how the actual application classes are nested inside the mixin layer. For instance, the roles for the *Vertex* and *Workspace* classes of Figure 1 correspond to `NUMBER::Vertex` and `NUMBER::Workspace`, respectively. Since roles are encapsulated, there is no possibility of name conflict. Moreover, we rely on the

[6]There are differences, but these are a consequence of C++ policies and are not important for our discussion (they are discussed together with other C++ specific issues in Smaragdakis [1999], Chapter 3).

standardization of role names. In this example the names `Workspace`, `Vertex`, and `Graph` are used for roles in all collaborations. Note how this is used in the above code fragment: Any class generated by this template defines roles that inherit from classes `Workspace` and `Vertex` in its superclass (`CollabSuper`).

Other collaborations of our Section 2.2 design are similarly represented as mixin layers. Thus, we have a `DFT` and a `UGRAPH` component that capture the *Depth-First Traversal* and *Undirected Graph* collaborations respectively. For instance, methods in the `Vertex` class of the `DFT` mixin layer include `visitDepthFirst` and `isVisited` (with implementations as suggested by their names). Similarly, methods in the `Vertex` class of `UGRAPH` include `addNeighbor`, `firstNeighbor`, and `nextNeighbor`, essentially implementing a graph as an adjacency list.

To implement default work methods for the depth-first traversal, we use an extra mixin layer, called `DEFAULTW`. The `DEFAULTW` mixin layer provides the methods for the `Graph` and `Vertex` classes that can be overridden by any graph algorithm (e.g., *Vertex Numbering*) used in a composition.

```
template <class CollabSuper>
class DEFAULTW : public CollabSuper {
public:
  class Vertex : public CollabSuper::Vertex {
  protected:
    bool workIsDone( CollabSuper::Workspace* )         {return 0;}
    void preWork( CollabSuper::Workspace* )            {}
    void postWork( CollabSuper::Workspace* )           {}
    void edgeWork( Vertex*, CollabSuper::Workspace* )  {}
  };

  class Graph : public CollabSuper::Graph {
  protected:
    void regionWork( Vertex*, CollabSuper::Workspace* ) {}
    void initWork( CollabSuper::Workspace* )           {}
    bool finishWork( CollabSuper::Workspace* )         {return 0;}
  };
};
```

The introduction of `DEFAULTW` (as a component separate from `DFT`) is an implementation detail, borrowed from the VanHilst and Notkin [1996b] implementation of this example. Its purpose is to avoid dynamic binding, and enable multiple algorithms to be composed as separate refinements of more than one `DFT` component. This topic is discussed in detail as part of the comparison of mixin layers and application frameworks (Section 3.2.2).

With the collaboration entities of the original design represented as distinct mixin layers, it is easy to produce an entire application by composing collaborations. In fact, the mixin layers defined, can be used to implement a *product-line*: a family of related applications. Different compositions of layers yield different products (members) of the family. In our example, the building blocks are the *Undirected Graph*, *Depth First traversal*, etc. collaborations. We show the

```
typedef DFT < NUMBER < DEFAULTW < UGRAPH > > > NumberC;
```

Fig. 4. A composition implementing the vertex numbering operation.

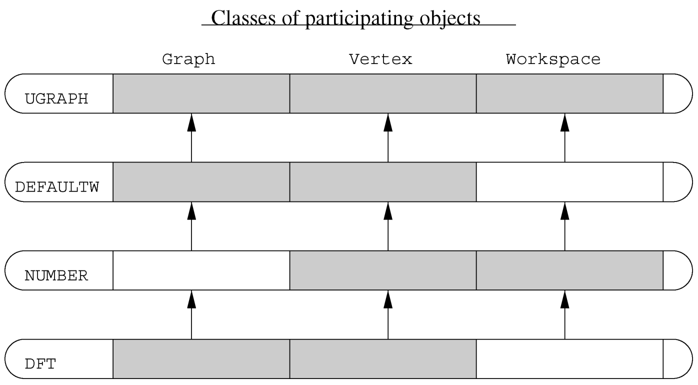

Fig. 5. Mixin-layers (ovals) and role-members (rectangles inside ovals) in the composition. Every component inherits from the one above it. Shaded role-members are those contained in the collaboration, unshaded are inherited. Arrows show inheritance relationships drawn from subclass to superclass.

collaborations that are composed to build the vertex numbering graph application in Figure 4. We will soon explain what this composition means, but first let us see how the different classes are related. The final implementation classes are members of the product of the composition, `NumberC` (e.g., `NumberC::Graph` is the concrete graph class). Figure 5 shows the mixin layers and their member classes, which represent roles, as they are actually composed. Each component inherits from the one above it. That is, `DFT` inherits role-members from `NUMBER`, which inherits from `DEFAULTW`, which inherits from `UGRAPH`. At the same time, `DFT::Graph` inherits methods and variables from `NUMBER::Graph`, which inherits from `DEFAULTW::Graph`, which inherits from `UGRAPH::Graph`. This double level of inheritance is what makes the mixin-layer approach so powerful. For instance, even though `NUMBER` does not specify a `Graph` member, it inherits one from `DEFAULTW`. The simplicity that this design affords becomes apparent in the following sections, when we compare it with alternatives.

The interpretation of the composition in Figure 4 is straightforward. It expresses the development of a vertex numbering application as a series of refinements. One begins with the `UGRAPH` mixin layer that implements an undirected graph. Next, default classes and methods that are common to all graph traversal algorithms are added by the mixin layer `DEFAULTW`. Then the algorithms and data members that are specific for vertex numbering are introduced by the `NUMBER` mixin layer. These algorithms, by themselves, are insufficient for performing vertex numbering because they rely on graph search algorithms which have yet to be added. Finally, the graph search algorithms—in this case, depth first search—are grafted on by the `DFT` mixin layer, thereby completing the specification and implementation of this application.

Thus, every mixin layer except `UGRAPH` is implemented in terms of the ones above it. For instance, `DFT` is implemented in terms of methods supplied by NUMBER, DEFAULTW, and UGRAPH. An actual code fragment from the visitDepthFirst method implementation in DFT::Vertex is the following:

```
for ( v = (Vertex*)firstNeighbor(); v != NULL;
      v = (Vertex*)nextNeighbor() )
{
  edgeWork(v, workspace);
  v->visitDepthFirst(workspace);
}
```

The firstNeighbor, nextNeighbor, and edgeWork methods are not implemented by the DFT component. Instead, they are inherited from components above it in the composition. firstNeighbor and nextNeighbor are implemented in the UGRAPH component (as they encode the iteration over nodes of a graph). edgeWork is a traversal refinement and (in this case) is implemented by the NUMBER component.

We can now see how mixin layers are both reusable and interchangeable. The DFT component of Figures 4 and 5 is oblivious to the *implementations* of methods in components above it. Instead, DFT only knows the *interface* of the methods it expects from its parent. Thus, the code above represents a skeleton expressed in terms of abstract operations firstNeighbor, nextNeighbor, and edgeWork. Changing the implementation of these operations merely requires the swapping of mixin layers. For instance, we can create an application (CycleC) that checks for cycles in a graph by replacing the NUMBER component with CYCLE:

```
typedef DFT < CYCLE < DEFAULTW < UGRAPH > > > CycleC;
```

The results of compositions (CycleC above and NumberC in Figure 4) can be used by a client program as follows: First, an instance of the nested Graph class (NumberC::Graph or CycleC::Graph) needs to be created. Then, Vertex objects are added and connected in the graph (the Graph role in mixin-layer UGRAPH defines methods addVertex and addEdge for this purpose). After the creation of the graph is complete, calling method depthFirst on it, executes the appropriate graph algorithm.

Mixin layers are the building blocks of a graph application product-line. Each mixin layer is a reusable component and different members (i.e., products) of the family can be created by using different compositions of mixin layers. Note that no direct editing of the component is necessary and multiple copies of the same component can co-exist in the same composition. For instance, we could combine two graph algorithms by using two instances of the DFT mixin layer (in the same inheritance hierarchy), refined to perform a different operation each time:

```
class NumberC : public DFT < NUMBER < DEFAULTW < UGRAPH > > > {};
class CycleC  : public DFT < CYCLE < NumberC > > {};
```

Both algorithms can be invoked, depending on whether we access the depth-first traversal through a NumberC or a CycleC reference:

```
CycleC::Graph *graph_c = new CycleC::Graph();
NumberC::Graph *graph_n = graph_c;
```

Now a call to `graph_c->depth_first` invokes the cycle checking algorithm, while a call to `graph_n->depth_first` calls the vertex numbering algorithm. (Alternatively, we can qualify method names directly, e.g.,

```
graph_c->NumberC::Graph::depth_first(...).)
```

As another example, the design may change to accommodate a different under-lying model. For instance, operations could now be performed on directed graphs. The corresponding update (`DGRAPH` replaces `UGRAPH`) to the composition is straight-forward (assuming that the algorithms are still valid for directed graphs as is the case with Holland's [1992] original implementation of this example):

```
typedef DFT < NUMBER < DEFAULTW < DGRAPH > > > NumberC;
```

Again, note that the interchangeability property is a result of the independence of collaborations.[7] A single `UGRAPH` collaboration completely incorporates all parts of an application that relate to maintaining an undirected graph (although these parts span several different classes). The collaboration communicates with the rest of the application through a well-defined and usually narrow interface.

For this and other similar examples, the reusability and interchangeability of mixin layers solves the *library scalability problem* [Batory et al. 1993; Biggerstaff 1994]: there are *n* features and often more than *n*! valid combinations (because composition order matters and feature replication is possible [Batory and O'Malley 1992]). Hard-coding all different combinations leads to libraries of exponential size—the addition of a single feature can double the size of a library. Instead, we would like to have a collection of building blocks, and compose them appropriately to derive the desired combination. In this way, the size of the library grows linearly in the number of features it can express (instead of exponentially, or super-exponentially).

Multiple Collaborations in a Single Design. An interesting question is whether mixin layers can be used to express collaboration-based designs where a single collaboration is instantiated more than once, with the same class playing different roles in each case. The answer is positive, and the desired result can be effected using *adaptor* mixin layers. Adaptor layers add no implementation, but adapt a class so that it can play a pre-defined role. That is, adaptor layers contain classes with empty bodies that are used to "redirect" the inheritance chain so that predefined classes can play the required roles.

Consider the case of a producer-consumer collaboration that was briefly discussed in Section 2.1. Our example is from the domain of compilers. A parser in a compiler can be viewed as a consumer of tokens produced by a lexical analyzer. At the same time, however, a parser is a producer of abstract

[7]By "independence" we mean that collaborations are composable because they conform to a particular design—all collaborations use Graph, Vertex, and Workspace classes with standardized methods. Given this standardization, the interchangeability—or independence—of these collaborations is achieved.

syntax trees (consumed, for instance, by an optimizer). We can reuse the same producer-consumer collaboration to express both of these relationships. The reason for wanting to provide a reusable implementation of the producer-consumer functionality is that it could be quite complex. For instance, the buffer for produced-consumed items may be guarded by a semaphore, multiple consumers could exist, and so on. The mixin layer implementing this collaboration takes `Item` as a parameter, describing the type of elements produced or consumed:

```
template <class CollabSuper, class Item>
class PRODCONS : public CollabSuper {
public:
  class Producer : public CollabSuper::Producer {
    void produce(Item item) { ... }
    // The functionality of producing Items is defined here
    ... // other Producer role methods
  };

  class Consumer : public CollabSuper::Consumer {
    Item consume() { ... }
    // The functionality of consuming Items is defined here
    ... // other Consumer role methods
  };
};
```

That is, `PRODCONS` adds the generic "produce" functionality to the Producer class and adds generic "consumer" functionality to the Consumer class.

Now we can use two simple adaptors to make a single class (`Parser`) be both a producer and a consumer in two different collaborations. The first adaptor (`PRODADAPT`) expresses the facts that a producer is also going to be a consumer (the actual consumer functionality is to be added later) and that the `Optimizer` class inherits the existing consumer functionality. This adaptor is shown below:

```
template <class CollabSuper>
class PRODADAPT : public CollabSuper {
public:
  class Consumer  : public CollabSuper::Producer {};
  class Optimizer : public CollabSuper::Consumer {};
  class Producer                                 {};
};
```

The second adaptor (`CONSADAPT`) is similar:

```
template <class CollabSuper>
class CONSADAPT : public CollabSuper {
public:
  class Lexer  : public CollabSuper::Producer    {};
  class Parser : public CollabSuper::Consumer    {};
};
```

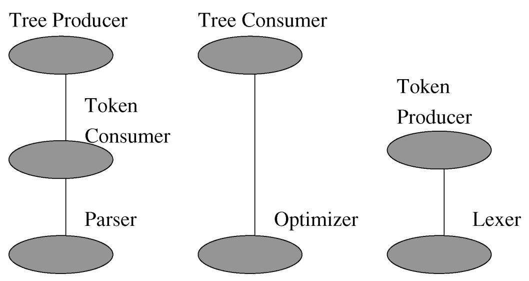

Fig. 6. The desired inheritance hierarchy has a `Parser` inheriting functionality both from a consumer class (a `Parser` is a consumer of tokens) and a producer class (a `Parser` is a producer of trees).

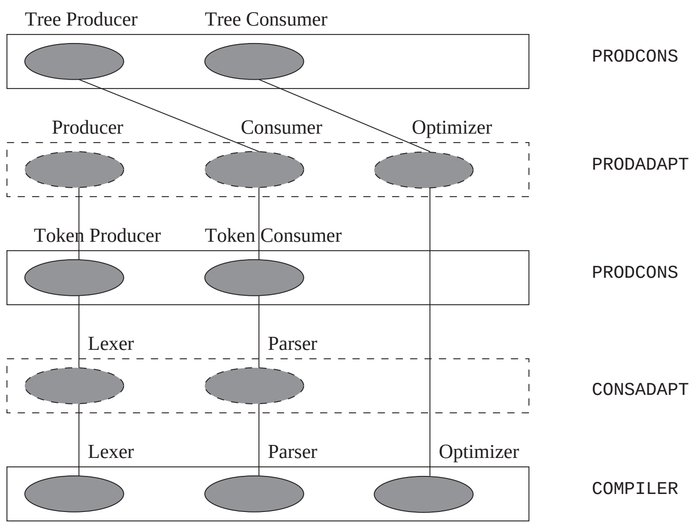

Fig. 7. By using adaptor layers (dotted rectangles), one can emulate the inheritance hierarchy of Figure 6, using only pre-defined mixin layers (solid rectangles). Since a single mixin layer (`PRODCONS`) is instantiated twice, adaptors help determine which class will play which role every time.

Now a single composition can contain two copies of the `PRODCONS` mixin layer, appropriately adapted. For instance:

```
typedef COMPILER < CONSADAPT < PRODCONS <
                   PRODADAPT < PRODCONS < ..., Tree> >, Token > > >
  CompilerApp;
```

In the above, the `COMPILER` mixin layer is assumed to contain the functionality of a compiler that defines three classes, `Lexer`, `Parser`, and `Optimizer`. These classes use the functionality supplied by the producer-consumer mixin layer. For instance, there may be a `parse` method in `COMPILER::Parser` that repeatedly calls the `consume` and `produce` methods. To better illustrate the role of adaptors, Figures 6 and 7 show the desired inheritance hierarchy for this example, as well as the way that adaptors are used to enable emulating this hierarchy using only predefined mixin layers. Note that each of the layers participating in the above composition appears as a rectangle in Figure 7.

3.2.2 *Comparison to Application Frameworks.* In object-oriented programming, an *abstract* class cannot be instantiated (i.e., it cannot be used to create objects), but is only used to capture the commonalities of other classes. These classes inherit the common interface and functionality of the abstract class. An *object-oriented application framework* (or just *framework*) consists of a suite of interrelated abstract classes that embodies an abstract design for software in a family of related systems [Johnson and Foote 1988]. Each major component of the system is represented by an abstract class. These classes contain dynamically bound methods (`virtual` in C++), so that the framework user can add functionality by creating subclasses and overriding the appropriate methods. Thus, frameworks have the advantage of allowing reuse at a granularity larger than a single abstract class. But frameworks have the disadvantage that using them means manually making the client classes inherit from framework classes. Thus, the framework classes cannot easily be interchanged (with a different, similar framework) and the client classes cannot be reused in a different context—they are hard-wired to the framework.

In a *white-box framework*, users specify system-specific functionality by adding *methods* to the framework's classes. Each method must adhere to the *internal* conventions of the classes. Thus, using white-box frameworks is difficult, because it requires knowledge of their implementation details. In a *black-box framework*, the system-specific functionality is provided by a set of classes. These classes must adhere only to the proper *external* interface. Thus, using black-box frameworks is easier, because it does not require knowledge of their implementation details. Using black-box frameworks is further simplified when they include a library of pre-written functionality that can be used as-is with the framework.

Frameworks can be used to implement collaboration-based designs, but the amount of flexibility and modularity they can afford is far from optimal. The reason is that frameworks allow the reuse of abstract classes but have no way of specifying collections of concrete classes that can be used at will (i.e., either included or not and in any order) to build an application (Batory et al. [2000a]). Intuitively, frameworks allow reusing the skeleton of an implementation but not the individual pieces that are built from the skeleton. This can be seen through a simple combinatorics argument. Consider a set of four features, A, B, C, and D that can be combined arbitrarily to yield complete applications. For simplicity, assume that feature A is always first, and that no feature repetition is allowed. Then a framework may encode feature combination AB, thus allowing the user to program combinations $ABCD$ and $ABDC$. Nevertheless, these combinations must be coded separately (i.e., they cannot use any common code other than their common prefix, AB). The reason is that each instantiation of the framework creates a separate inheritance hierarchy, and reusing a combination is possible only if one can inherit from one of its (intermediate or final) classes—only common prefixes are reusable. In our four-feature example, combinations that have no common prefix with the framework (for instance, *ACD*) simply cannot take advantage of it and have to be coded separately. This amounts to exponential redundancy for complex domains.

In the general case, assume a simple cost model that assigns one cost unit to each reimplementation of a feature. If feature order matters, but no repetitions are possible, the cost of implementing all possible combinations using frameworks is equal to the number of combinations (each combination of length k differs by one feature from its prefix of length $k-1$). Thus, for n features, the total cost for implementing all combinations using frameworks is $\sum_{k=1}^{n} \frac{n!}{(n-k)!}$. (This number is derived by considering the sum of the feature combinations of length k, for each k from 1 to n.) In contrast, the cost of using mixin layers for the same implementation is equal to n—each component is implemented once and can be combined in arbitrarily many ways. With mixin layers, even compositions with no common prefixes share component implementations.

Even though our combinatorics argument represents an extreme case, it is reflective of the inflexibility of frameworks. For instance, optional features are common in practice and frameworks cannot accommodate them, unless all combinations are explicitly coded by the user. This is true even for domains where feature composition order does not matter, or features have a specific order in which they must be used.

Another disadvantage of using frameworks to implement collaboration-based designs, comes from the use of dynamically bound methods in frameworks. Even though the dynamic dispatch cost is sometimes negligible, or can be optimized away, it often imposes a run-time overhead, especially for fine-grained classes and methods. With mixin layers, this overhead is avoided, as there is little need for dynamic dispatch. The reason is that mixin layers can be ordered in a composition, so that most of the method calls are to their parent layers.

This reveals a general and important difference between mixin-based programming and standard object-oriented programming. When a code fragment in a conventional OO class needs to be generic, it is implemented in terms of dynamically bound methods. These methods are later overridden in a subclass of the original class, thus refining it for a specific purpose. With mixin classes, the situation is different. A method in a mixin class can define generic functionality by calling methods in the class's (yet undefined) *superclass*. That is, generic calls for mixins can be both up-calls and down-calls in the inheritance hierarchy. Generic up-calls are specialized statically, when the mixin class's superclass is set. Generic down-calls provide the standard OO run-time binding capabilities. Their use can be limited to cases where the exact version of the method to be called, is truly not known until run-time. In contrast, in application frameworks, dynamic binding is often used just for modularity reasons (calling functionality without yet having defined it) even if the target ends up being known statically. This can be eliminated in a mixin-based approach because we are allowed to add functionality to a mixin class's superclass. Refinement of existing functionality is not just a top-down process but involves composing mixins arbitrarily, often with many different orders being meaningful.

Example. We illustrate the above points with the graph algorithm example of Section 2.2. The original implementation of this application [Holland 1992] used a black-box application framework on which the three graph algorithms were implemented. The framework consists of the implementations of the `Graph`, `Vertex`, and `Workspace` classes for the *Undirected Graph* and *Depth First Traversal* collaborations. The classes implementing the depth-first traversal have methods like `preWork`, `postWork`, `edgeWork`, etc., *which are declared to be dynamically bound* (`virtual` in C++). In this way, any classes inheriting from the framework classes can refine the traversal functionality by redefining the operation to be performed the first time a node is visited, when an edge is traversed, and so forth.

VanHilst and Notkin [1996b] discussed the framework implementation of this example in detail. Our presentation here merely adapts their observations to our discussion of using frameworks to implement collaboration-based designs. A first observation is that, in the framework implementation, the base classes are fixed, and changing them requires hand-editing (usually copying and editing, which results in redundant code). For instance, consider applying the same algorithms to a directed, as opposed to an undirected graph. If both combinations need to be used in the same application, code replication is necessary. The reason is that the classes implementing the graph algorithms (e.g., *Vertex Numbering*) must have a fixed superclass. Hence, two different sets of classes must be introduced, both implementing the same graph algorithm functionality but having different superclasses.

A second important observation pertains to our earlier discussion of optional features in an application. In particular, a framework implementation does not allow more than one refinement to exist in the same inheritance hierarchy. Thus, with frameworks, unlike the mixin layer version of the code in Section 3.2.1, we cannot have a single graph that implements both the *Vertex Numbering* and the *Cycle Checking* operations. The reason is that the dynamic binding of methods in the classes implementing the depth-first traversal causes the most refined version of a method to be executed on every invocation. Thus, multiple refinements cannot coexist in the same inheritance hierarchy since the bottom-most one in the inheritance chain always supersedes any others. In contrast, the flexibility of mixin layers allows us to break the depth-first traversal interface in two (the `DEFAULTW` and the `DFT` component, discussed earlier), so that `DFT` calls the refined methods *in its superclass* (i.e., without needing dynamic binding). In this way, multiple copies of the `DFT` component can coexist and be refined separately. At the same time, obviating dynamic binding results in a more efficient implementation—dynamic dispatch incurs higher overhead than calling methods of known classes (although sometimes it can be optimized by an aggressive compiler).

3.2.3 *Comparison to the VanHilst and Notkin Method.* The VanHilst and Notkin [1996a, 1996b, 1996c, 1997] approach is another technique that can be used to map collaboration-based designs into programs. The method employs C++ mixin classes, which offer the same flexibility advantages over a framework implementation as the mixin layers approach. Nevertheless, the

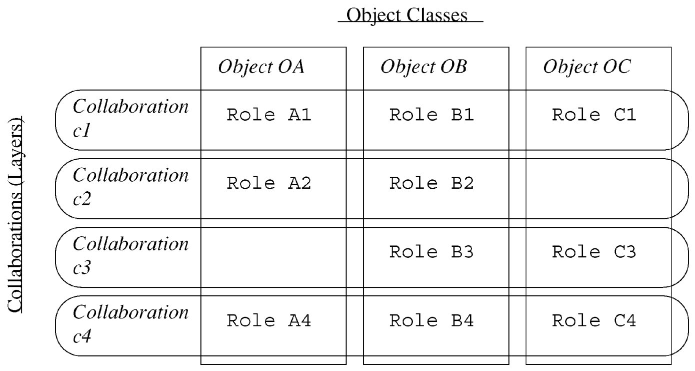

Fig. 8. Example collaboration decomposition. Ovals represent collaborations, rectangles represent objects, their intersections represent roles.

components represented by VanHilst and Notkin are small-scale, resulting in complicated specifications of their interdependencies.

VanHilst and Notkin use mixins in C++ to represent roles. More specifically, each individual role is mapped to a different mixin and is also parameterized by any other classes that interact with the given role in its collaboration. For an example, consider role `B4` in Figure 8 (which replicates Figure 1 for easy reference). This role participates in a collaboration together with two other roles, `A4` and `C4`. Hence, it needs to be aware of the classes playing the two roles (so that, for instance, it can call appropriate methods). With the VanHilst and Notkin technique, the role implementation would be a mixin that is parameterized by these two extra classes:

```
template <class RoleSuper, class OA, class OC>
class B4 : public RoleSuper {
   ... /* role implementation, using OA, OC */
};
```

Consider that the actual values for parameters `OA`, `OC` would themselves be the result of template instantiations, and their parameters also, and so on (up to a depth equal to the number of collaborations). This makes the VanHilst and Notkin method complicated even for relatively small examples. In the case of a composition of n collaborations, each with m roles, the VanHilst and Notkin method can yield a parameterization expression of length m^n. Additionally, the programmer has to explicitly keep track of the mapping between roles and classes, as well as the collaborations in which a class participates. For instance, the mixin for role `A4` in Figure 1 has to be parameterized with the mixin for role `A2`—the programmer cannot ignore the fact that collaboration *c3* does not specify a role for object *OA*. From a software evolution standpoint, local design changes cannot be easily isolated, since collaborations are not explicitly represented as components. These limitations make the approach unscalable: various metrics of programmer effort (e.g., length of composition expressions, parameter bindings that need to be maintained, etc.) grow exponentially in the

```
class NumberC: public DFT <NUMBER <DEFAULTW <UGRAPH> > > {};
class CycleC : public DFT < CYCLE < NumberC > >         {};
```

Fig. 9. Our mixin layer implementation of a multiple-collaboration composition. The individual classes are members of `NumberC`, `CycleC` (e.g., `NumberC::Vertex`, `CycleC::Graph`, etc.).

number of features supported. (This is the same notion of scalability as in our earlier discussion of the library scalability problem.)

Conceptually, the scalability problems of the VanHilst and Notkin approach are due to the small granularity of the entities they represent: each mixin class represents a single role. Roles, however, have many external dependencies (for instance, they often depend on many other roles in the same collaboration). To avoid hard-coding such dependencies, we have to express them as extra parameters to the mixin class, as in the preceding code fragment. Reusable components should have few external dependencies, as made possible by using mixin layers to model collaborations.

Example. Consider a composition implementing both the *Cycle Checking* and the *Vertex Numbering* operation on the same graph. Recall that the ability to compose more than one refinement (or multiple copies of the same refinement) is an advantage of the mixin-based approach (both ours and the VanHilst and Notkin method) over frameworks implementations.

The components (mixins) used by VanHilst and Notkin are similar to the inner classes in our mixin layers, with extra parameters needed to express their dependencies with other roles in the same collaboration. Our specification is shown in Figure 9 (reproducing a previously presented code fragment). A compact representation of a VanHilst and Notkin specification is shown in Figure 10. (A more readable version of the same code included in VanHilst and Notkin [1996b] is even lengthier.)[8]

Figure 10 makes the complications of the VanHilst and Notkin approach apparent. Each mixin representing a role can have an arbitrary number of parameters and can instantiate a parameter of other mixins. In this way, parameterization expressions of exponential (to the number of collaborations) length can result. To alleviate this problem, the programmer has to introduce explicitly intermediate types that encode common sub-expressions. For instance, `V` is an intermediate type in Figure 10. Its only purpose is to avoid introducing the sub-expression `VertexDFT<WS,VNumber>` three different times (wherever `V` is used). Of course, `VNumber` itself is also just a shorthand for `VertexNumber<WS,VWork>`. `VWork`, in turn, stands for `VertexDefaultWork-<WS,VGraph>`, and so on.[9] Additional complications arise when specifying a composition: users must know the number and position of each parameter of a role-component. Both of the above requirements significantly complicate the implementation and make it error-prone.

[8]The object code of both is, as expected, of almost identical size.

[9]Some compilers (e.g., MS VC++, g++) internally expand template expressions, even though the user has explicitly introduced intermediate types. This caused page-long error messages for incorrect compositions when we experimented with the VanHilst and Notkin method, rendering debugging impossible.

```
class Empty {};
class WS         : public WorkspaceNumber                {};
class WS2        : public WorkspaceCycle                 {};
class VGraph     : public VertexAdj<Empty>               {};
class VWork      : public VertexDefaultWork<WS,VGraph>   {};
class VNumber    : public VertexNumber<WS,VWork>         {};
class V          : public VertexDFT<WS,VNumber>          {};
class VWork2     : public VertexDefaultWork<WS2,V>       {};
class VCycle     : public VertexCycle<WS2,VWork2>        {};
class V2         : public VertexDFT<WS2,VCycle>          {};
class GGraph     : public GraphUndirected<V2>            {};
class GWork      : public GraphDefaultWork<V,WS,GGraph>  {};
class Graph      : public GraphDFT<V,WS,GWork>           {};
class GWork2     : public GraphDefaultWork<V2,WS2,Graph> {};
class GCycle     : public GraphCycle<WS2,GWork2>         {};
class Graph2     : public GraphDFT<V2,WS2,GCycle>        {};
```

Fig. 10. Same implementation using the VanHilst/Notkin approach. `V` corresponds to our `NumberC::Vertex`, `Graph` to `NumberC::Graph`, `WS` to `NumberC::Workspace`, etc.

Using mixin layers, the exponential blowup of parameterization expressions is avoided. Every mixin layer has only a single parameter (the layer above it). By parameterizing a mixin layer A by B, A becomes implicitly parameterized by all the roles of B. Furthermore, if B does not contain a role for an object that A expects, it will inherit one from above it. This is the benefit of expressing the collaborations themselves as classes: they can extend their interface using inheritance.

Another practical advantage of mixin layers is that it encourages consistent naming for roles. Hence, instead of explicitly giving unique names to role-members, we have standard names and distinguish instances only by their enclosing mixin layer. In this way, `VertexDFT`, `GraphDFT`, and `VertexNumber` become `DFT::Vertex`, `DFT::Graph` and `NUMBER::Vertex`, respectively.

VanHilst and Notkin [1996b] questioned the scalability of their method. One of their concerns was that the composition of large numbers of roles "can be confusing even in small examples..." The observations above (length of parameterization expressions, number of components, consistent naming) show that mixin layers address this problem, and do scale gracefully, without losing the advantages of the VanHilst and Notkin implementation.

3.3 Mixin Layers Considerations

We have argued that mixin layers are better for implementing collaboration-based designs than other alternatives. Nevertheless, mixin layers are certainly not a "silver bullet." They are good for in-house development of product-line architectures for mature domains, and require programming language and tool support for specification and debugging. These points are analyzed below in more detail, but we note that they are by no means specific to mixin layers; other competitive techniques (e.g., application frameworks, or the VanHilst and Notkin method) have similar restrictions.

—*Appropriate Domains for Mixin Layers*: Mixin layers are not appropriate for every domain. In general, the most suitable domains are mature, well-understood, amenable to detailed decompositions, and elaborations of collaboration-based designs. The domain should be decomposable into largely independent refinements. Composing such refinements need not result in an increase in the level of abstraction. Instead, refinements can represent different concerns at the same conceptual level. (E.g., the addition of more operations on graphs does not alter the abstraction that we are still dealing with graphs, rather, adding more operations merely enriches the graph abstraction.) A well-known observation is that, even in strictly layered domains, like operating systems, the notion of "information module" does not necessarily coincide with the notion of "layer of abstraction." Modules may encompass different parts of several layers [Habermann et al. 1976]. Mixin layers are a kind of "information module" and similar observations apply. Mixin layers lead to physically layered implementations, which may or may not have a negative impact on application performance. Mixin layers are implementations of a standard design imposed on a domain. In-house environments of individual companies are best to maintain this standard; open collaborative communities might make such standards difficult to follow. No precise quantification of these properties can be given, but a designer can usually assess the appropriateness of our techniques.

—*Difficulties in Using Mixin Layers*: Good OO designs limit the depth of inheritance hierarchies to a small number (e.g., 3). In contrast, compositions of mixin layers often lead to long inheritance chains. This can become a problem for debugging (chasing method calls up an inheritance hierarchy) and for understanding where the functionality of a class is located on an inheritance chain. Another difficulty can be learning the order in which mixin layers can be composed. While this can be ameliorated by good tool support [Batory and Geraci 1997], it is something more that needs to be learned; and composition rules need to be precisely stated.

—*Implementation Requirements for Mixin Layers and Interaction with Language Features*: Mixin layers are only as good as the technology to support them. Some of the proposed implementation techniques have specific technical disadvantages, especially in conjunction with particular compiler technology. For instance, our C++ template implementation of mixin layers may result in (binary) code duplication, if the same layer is used multiple times in a composition. Nevertheless, no fundamental implementation drawbacks exist in relation to mixin layers. Implementation considerations for the C++ version of mixin layers are described in Smaragdakis and Batory [2000].

Several general programming language issues arise in connection with mixin layers and their compositions. Most of these issues pertain to the interactions of mixin layers with type systems. Type information can be used to detect errors in a composition of mixin layers. At the same time, layers are defined in isolation, and the problem of propagating type information between layers is especially interesting. Since the focus of this paper is not on concrete language solutions, we point the reader to Smaragdakis [1999, Chapter 3] where such issues are analyzed in detail.

4. AN APPLICATION: THE JAKARTA TOOL SUITE

In this section, we discuss an application of mixin layers to a medium-size software project (about 30 K lines of code). The project is the *Jakarta Tool Suite* (*JTS*) [Batory et al. 1998]—a set of language extensibility tools, aimed mainly at the Java language. We use mixin layers as the building blocks that form different versions of the *Jak* tool of JTS. Jak is the modular compiler in JTS. Different versions of Jak can be created using different combinations of layers. Layers may be responsible for type-checking, compiling, and/or creating code for a different set of language constructs. Additionally, layers may be used to add new functionality across a large group of existing classes. In this way, the user can design a language by putting together conceptual language "modules" (i.e., consistent sets of language constructs) and implement a compiler for this language as a version of Jak composed of the mixin layers corresponding to each language module. Currently available layers support the base Java language, meta-programming extensions, general purpose extensions (e.g., syntax macros for Java), a domain-specific language for data structure programming (P3), and so forth.

The choice of the compiler domain as a large-scale test case for mixin layers is not arbitrary. Compilers are well-understood, with modern compiler construction benefiting from years of formal development and stylized design patterns. The domain of compilers has been used several times in the past in order to demonstrate modularization mechanisms. Selectively, we mention the *visitor* design pattern [Gamma et al. 1995], which is commonly described using the example of a compiler with a class corresponding to each syntactic type that its parser can recognize (e.g., there is a class for if-statements, a class for declarations, etc.). In this case, the visitor pattern can be used to add new functionality to all classes, without distributing this functionality across the classes. Our application of mixin layers to the compilers domain has very much the same modularization flavor. We use mixin layers to isolate aspects of the compiler implementation, which can be added and removed at will. Compared to the visitor pattern, mixin layers offer greater capabilities—for instance, allowing the addition of state (i.e., member variables) to existing classes.

The outcome of applying mixin layers to JTS was very successful. The flexibility afforded by a layered design is essential in forming compilers for different language dialects. Additionally, mixin layers helped with the internal organization of the code, so that changes were easily localized. Additions that could be conceptually grouped together (like those reflecting the language changes from Java 1.0 to Java 1.1) were introduced as new mixin layers, without disrupting the existing design. JTS was thus easier to implement and has become easier to maintain.

We next discuss JTS and the use of mixin layers in its implementation. Section 4.1 offers some essential background in JTS by describing the way parsers are generated and initial class hierarchies are established based on language syntax. Section 4.2 discusses the actual application of mixin layers in JTS.

4.1 JTS Background: Bali as a Parser Generator

Bali is the JTS tool responsible for putting together compilers. Although Bali is a component-based tool, in this section we limit our attention to the more conventional grammar-specification aspects of Bali.

The syntax of a language is specified as a Bali grammar, which is an annotated BNF grammar extended with regular-expression repetitions. Bali transforms a Bali grammar into a lexical analyzer and parser. For example, two Bali productions are shown below: one defines `StatementList` as a sequence of one or more `Statements`, and the other defines `ArgumentList` as a sequence of one or more `Arguments` separated by commas.

```
StatementList : ( Statement )+ ;
ArgumentList : Argument ( ‘,’ Argument )*;
```

Repetitions have been used before in the literature [Wirth 1977; Wile 1993; Reasoning Systems 1990]. They simplify grammar specifications and allow an efficient internal representation as a list of trees.

Bali productions are annotated by the class of objects that is to be instantiated when the production is recognized. For example, consider the Bali specification of the Jak `SelectStmt` rule:

```
SelectStmt
  : IF ‘(’ Expression ‘)’ Statement    ::IfStm
  | SWITCH ‘(’ Expression ‘)’ Block    ::SwStm
  ;
```

When a parser recognizes an “if” statement (i.e., an `IF` token, followed by ‘(‘, `Expression`, ‘)’, and `Statement`), an object of class `IfStm` is created. Similarly, when the pattern defining a “switch” statement (a `SWITCH` token followed by ‘(‘, `Expression`, ‘)’, and `Block`) is recognized, an object of class `SwStm` is created. As a program is parsed, the parser instantiates the classes that annotate productions, and links these objects together to produce the syntax tree of that program.

A Bali grammar specification is a streamlined document. It is a list of the lexical patterns that define the tokens of the grammar followed by a list of annotated productions that define the grammar itself. A Bali grammar for an elementary integer calculator is shown in Figure 11. From this grammar specification, Bali generates a lexical analyzer and a parser (we use the `JavaCC` lexer/parser generator as a backend).

Associating grammar rules with classes allows Bali to do more than generate a parser. In particular, Bali can deduce an inheritance hierarchy of classes representing different pieces of syntax. Consider Figure 12, which shows rules `Rule1` and `Rule2`. When an instance of `Rule1` is parsed, it may be an instance of `pattern1` (an object of class `C1`), or an instance of `Rule2` (an object of class `Rule2`). Similarly, an instance of `Rule2` is either an instance of `pattern2` (an object of `C2`) or an instance of `pattern3` (an object of `C3`). The inheritance hierarchy of Figure 12 is constructed from this information: classes `C1` and `Rule2` are subclasses of `Rule1`, and `C2` and `C3` are subclasses of `Rule2`.

```
       // Lexeme definitions
"print"   PRINT
"+"       PLUS
"-"       MINUS
"("       LPAREN
")"       RPAREN
"[0-9]*"  INTEGER

%%        // production definitions
          // start rule is Action

Action  : PRINT Expr                       :: Print
        ;
Expr    : Expr PLUS Expr                   :: Plus
        | Expr MINUS Expr                  :: Minus
        | MINUS Expr                       :: UnaryMinus
        | LPAREN Expr RPAREN               :: Paren
        | INTEGER                          :: Integer
        ;
```

Fig. 11. A Bali grammar for an integer calculator.

```
Rule1 : pattern1   :: C1
      | Rule2
      ;

Rule2 : pattern2   :: C2
      | pattern3   :: C3
      ;
```

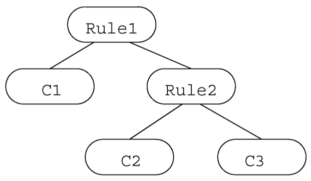

Fig. 12. Inferring inheritance hierarchies from grammar rules.

Additionally, for each production Bali infers the constructors for syntax tree node classes. Each parameter of a constructor corresponds to a token or non-terminal of a pattern.[10] For example, the constructor of the `IfStm` class has the following signature:

```
IfStm(Token iftk, Token lp, Expression exp, Token rp, Statement st)
```

Methods for editing and unparsing nodes are additionally generated.

Although Bali automatically generates an inheritance hierarchy and some methods for the produced Jak compiler, there are obviously many methods that cannot be generated automatically. These include type checking, reduction, and optimization methods. Such methods are syntax-type-specific; we hand-code these methods and encapsulate them as a mixin layer that contains subclasses of Bali-generated classes.

In essence, Bali takes the grammar specification and uses it to produce a skeleton for the compiler of the language. The skeleton has the form of a set of classes organized in an inheritance hierarchy, together with the methods that

[10]The tokens need not be saved. However, Bali-produced precompilers presently save all white space—including comments—with tokens. In this way, JTS-produced tools that transform domain-specific programs retain embedded comments. This is useful when debugging programs with a mixture of generated and hand-written code, and is a necessary feature if transformed programs are subsequently maintained by hand.

can be automatically produced (that is, constructors, editing, and unparsing methods). In other words, Bali produces an *application framework* for a compiler. The framework is encapsulated in a mixin layer that occupies the root of all mixin layer compositions implementing different versions of Jak.

4.2 Bali Components and Mixin Layers in JTS

Apart from its parser generator aspect, Bali is also a tool that synthesizes language implementations from components. Bali can create compilers for a family of languages, depending on the selection of components used as its input. This is essentially a product-line of language translators, with their common functionality factored out in reusable components. We use the name *Jak* for any Bali-generated compiler. Currently available Bali components support the base Java language, meta-programming extensions (e.g., code template operators), general purpose extensions (e.g., syntax macros for Java), a domain-specific language for state machines [Batory et al. 2000b], and more. Compositions of these components define different variants of Jak (i.e., different members of a product-line of Java dialects): with and without meta-programming constructs, with and without state machine extensions, with and without data structure extensions, and so on. This is another instance of the library scalability problem [Batory et al. 1993; Biggerstaff 1994]. We want to compose the different variants of Jak from components encapsulating orthogonal units of functionality.

A *Bali component* has two parts: The first is a Bali grammar file, which contains the lexical tokens and grammar rules that define the syntax of the host language or language extension—for extensions that only change the semantics, but not the syntax, this file is absent. The second is a mixin layer encapsulating a collection of multiple hand-coded classes that contain the reduction, type-checking, and so on, methods for each syntax type defined in that grammar file.

To illustrate how classes are defined and refined in Bali, consider four concrete Bali components: `Java` is a component implementing the base Java language, `SST` implements code template operators like tree constructors and explicit escapes,[11] `GScope` supplies scoping support for program generation, and `P3` implements a language for data structures. The Jak language and compiler can be defined by a composition of these components. We use the `[...]` operator to designate component composition—for instance, `P3[GScope[SST[Java]]]`.

The syntax of a composed language is defined by taking the union of the sets of production rules in each Bali component grammar. The semantics of a composition is defined by composing the corresponding mixin layers. Figure 13 depicts the class hierarchy of the Jak compiler. `AstNode` belongs to the JTS kernel, and is the root of all inheritance hierarchies that Bali generates. Using the composition grammar file (the union of the grammar files for the `Java`, `SST`, `GScope`, and `P3` components), Bali generates a mixin layer that encapsulates

[11]Our code template operators are analogous to the backquote/unquote pair of Lisp operators. Unlike Lisp, however, multiple operators exist in JTS—one for each syntactic type (e.g., declaration, expression, etc.). Multiple constructors in syntactically rich languages are common (e.g., Weise and Crew [1993]; Chiba [1996]). The main reason has to do with the ease of parsing code fragments.

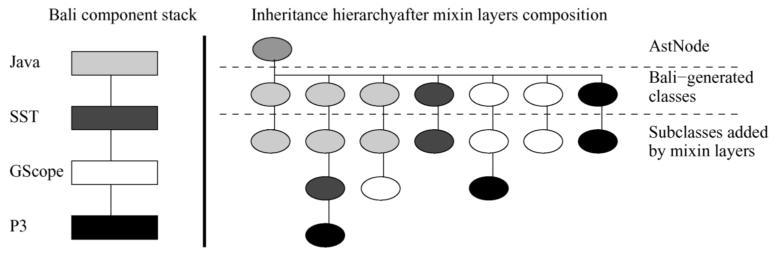
Bali component stack
Inheritance hierarchyafter mixin layers composition
Java
SST
GScope
P3
AstNode
Bali–generated classes
Subclasses added by mixin layers

Fig. 13. The Jak inheritance hierarchy.

the hierarchy of classes that contain tree node constructors, unparsing, and editing methods. Each remaining mixin layer then grafts its hand-coded classes onto this hierarchy. These define the reduction, optimization, and type-checking methods of tree nodes by refining existing classes. *The terminal classes of this hierarchy are those that are instantiated by the generated compiler.*

It is worth noting that Figure 13 is not drawn to scale. Jak consists of over 500 classes. The number of classes that a mixin layer adds to an existing hierarchy ranges from 5 to 40. Nevertheless, the simplicity and economy of specifying Jak using component compositions is enormous—to build the Jak compiler, all that users have to provide to Bali is the equation `Jak = P3[GScope[SST[Java]]]`, and Bali does the rest. To compose all these classes by hand (as would be required by Java) would be very slow, extremely tedious, and error prone. Additionally, the scalability advantages of mixin layers can easily be demonstrated: when new extension mechanisms or new base languages are specified as components, a subset of them can be selected and Bali automatically composes a compiler for the desired language variant.

4.3 Java Mixin Layers for JTS

In Section 3.1.3, we discussed the applicability of mixin layers in various programming languages. There, we explained that Java already supports nested classes, but the language currently specifies no parameterization mechanism. Furthermore, some of the proposed parameterization mechanisms for Java (e.g., Pizza [Odersky and Wadler 1997] or Thorup's [1997] virtual types) do not support parameterized inheritance. In order to support mixin layers for Bali components in JTS, we implemented our own Java language extensions for parameterization. This section gives a brief overview of the main language construct.

Our parameterization extensions to Java are geared towards mixin layer development (as opposed to general-purpose genericity). Our approach in designing and implementing these language constructs was motivated by pragmatic, not conceptual, considerations: we needed a layer mechanism to facilitate our own development efforts—not to supply the best-designed and robust parameterization mechanism for Java. Therefore, our implementation was straightforward, adopting a heterogeneous model of transformation: for each instantiation of a mixin layer, a new Java class is created at the source code level. Thus, our approach resembles C++ template instantiation and does not take advantage of the facilities for load-time class adaptation offered by the Java Virtual Machine (see, e.g., the approach of Agesen et al. [1997], and the work on binary component adaptation [Keller and Hölzle 1998]). Nevertheless, in this context, our approach is not necessarily at a disadvantage. Mixin layers in Bali component compositions are never reused in the same application (i.e., a single Jak compiler uses at most one instance of a mixin layer). Therefore, code bloat (redundancy in generated classes) is not a problem. At the same time, our straightforward approach made for an easier implementation, which contributed to the faster development of JTS.

The implementation of our Java extensions for mixin layer support occurred concurrently with the development of JTS. In fact, an early version of JTS was used to implement the first version of our Java mixin layers. The Java mixin layers were, in turn, used to evolve and further develop JTS, resulting in a bootstrapped implementation. (Actually, this is not the only reason that JTS is based on a bootstrapped implementation. Another reason is that the meta-programming capabilities added to Java have been used in the code that implements JTS itself. The entire JTS system is compiled using a basic version of the Jak compiler, composed of only a few layers that specify the basic Java language, code template operators, syntax macros, etc.)

The syntax of mixin layers is straightforward and resembles their C++ counterparts. Two new keywords are introduced: `layer` and `realm`. The `layer` keyword is analogous to `class` but defines a mixin layer (i.e., an outer class that may be parameterized with respect to its superclass). The `realm` keyword is used to specify interface conformance for mixin layers, in analogy to the Java `implements` keyword. Finally, the `[...]` operator is used to specify layer composition. The (slightly simplified) general form of a layer definition is shown below, with the terminal symbols appearing in bold for clarity:

`layer_definition :`

layer `layer_name` **(**`param_list`**)** **realm** `realm_name [super]`

{ `declaration_list` **}**

The syntax for non-terminals in the above definition is straightforward. `param_list` is a list of type parameters for the mixin layer. If the parameter list contains layers, the parameterization can be constrained by specifying the expected realm of these layers. The optional `super` construct designates an `extends` clause (in much the same way as for regular Java classes). The contents of a mixin layer can only be Java type declarations.

The actual details of our implementation are not important; we consider the general approach that this implementation represents to be of much greater importance. What we did in JTS is an example of a *domain-specific languages* approach to software construction. In the course of creating a medium-size software project, we recognized that mixin layers would significantly facilitate our task. That is, we saw an opportunity for improving our implementation through extra language support. It then proved cost-effective to add the extra linguistic constructs that were needed (i.e., mixin layers), in the course of implementing the original project (i.e., JTS).

It is our belief that the domain-specific language approach to software construction is a promising way to build better software. The designer of a software application can (and should) be thinking about language constructs that can have a significant impact in the application's efficiency, maintainability, and reusability. Often, such constructs can be readily identified, but they are not available in the implementation language of choice. With the advent of language extensibility tools, as well as extensible/reflective programming languages, supplying special-purpose (or *domain-specific*) language support may be the right approach in fighting software complexity. JTS is a tool aimed at facilitating the implementation of domain-specific languages and language extensions. The use of mixin layers in the implementation of JTS is a vivid demonstration of the same paradigm that JTS promotes.

5. RELATED WORK

There is an enormous wealth of research in the area of component-based software construction and code modularization. Here, we selectively discuss some approaches that are related to our work but have not been previously described in this paper.

5.1 GenVoca

GenVoca is a layered design and implementation methodology, mainly applied to application generators (i.e., compilers for domain-specific programming languages). GenVoca advocates that a domain be decomposed in terms of largely-orthogonal features which are implemented as layers. Applications in the domain can be synthesized by composing layers; layer composition is performed by a generator. The name "GenVoca" was derived from the first two generators that exhibited these principles: Genesis (extensible database systems) [Batory 1987; Batory et al. 1988] and Avoca (network protocols) [O'Malley and Peterson 1992]. GenVoca generators for other domains include: data manipulation languages [Villarreal 1994], distributed file systems [Heidemann and Popak 1994], host-at-sea buoy systems [Weiss 1990], and real-time avionics software [Coglianese and Szymanski 1993]. Mixin layers were originally inspired by the GenVoca model and are now an essential part of its arsenal of implementation techniques. Although we have not attempted full implementations, our experience suggests that mixin layers can be used to obtain many of the same benefits as full GenVoca generators, for the above domains. That is, much of the benefit of GenVoca generators is due to the layering technology and not to the use of compiler techniques.

5.2 Modules in High-Level Languages

High-level languages often provide *modules* (a.k.a. *packages* or *namespaces*) as fundamental abstractions. Representative approaches include Ada *packages* [International Organization for Standardization 1995]—which is a prototypical modularization scheme for block structured languages, ML

[Milner et al. 1990]—which provides a very powerful module system based on polymorphic types, Java *packages*, and C++ namespaces [Stroustrup 1997].[12]

Mixin layers are expressible in the latest incarnations of Ada (Ada95 [International Organization for Standardization 1995]). Standard ML still lacks support for extensible records (i.e., a counterpart of inheritance). Nevertheless, there is nothing fundamental that prevents integrating mixin layers. Recent research has brought some of the mixin layers ideas in a modular language framework. Findler and Flatt's [1998] work introduces constructs remarkably similar to mixin layers, in an experimental, module-based object system.

The most interesting lesson, however, is that modules—unlike classes—are often not well integrated in programming languages. For example, a C++ namespace cannot be parameterized, while a class can. This prevents us from using mixin-like patterns with C++ namespaces. With class nesting and parameterized inheritance, mixin layers are a kind of module with some desirable characteristics from a software engineering standpoint.

5.3 Meta-Object Protocols

Meta-Object Protocols (e.g., Forman et al. [1994]; Kiczales et al. [1991]) are reflective facilities for modifying the behavior of an object system while the system is being used. Classical modifications include executing arbitrary code around method invocations (method *wrapping*), and changing the semantics of inheritance. Specific examples of method wrapping include function tracing, invariant checking, and object locking [Forman et al. 1994].

Meta-object protocols solve a different problem than mixin layers. Mixin layers address the issue of grouping class refinements together so they can be treated as a unit. In contrast, meta-object protocols can express modifications to fundamental operations of an object system. Meta-object protocols can be used for desirable functionality additions that are not convenient with mixin layers—for example, the application of a single wrapper to all methods of a class at once. Of course, a meta-object protocol is a mechanism, not a design guideline. An appropriately designed meta-object protocol, allowing the encapsulation of many metaclasses in parameterized modules, could certainly be used to implement mixin layers. Unfortunately, to our knowledge, none of the standard meta-object protocols offer such encapsulation capabilities.

5.4 Aspect-Oriented Programming

Aspect-oriented programming (*AOP*) advocates decomposing application domains into orthogonal *aspects* [Kiczales et al. 1997]. Aspects are distinct implementation entities that encapsulate code that would otherwise be intertwined

[12]It is perhaps debatable, whether C++ namespaces and Java packages are modules, because they can later be re-opened and have more definitions added to them. Nevertheless, we choose to include these mechanisms here. In practice, they are often used, under certain assumptions, in the same way as modules in other languages. For instance, several Java tools perform whole-package static analysis, although a change in any file of the package may invalidate the results of the entire analysis.

throughout an application. In this respect, aspect-oriented programming seems strikingly similar to GenVoca. Indeed, early AOP manifestos [Kiczales et al. 1997] are very similar to the work describing GenVoca generators: the software engineering arguments are identical and the implementation techniques used are very similar. Many of the AOP example applications in Kiczales et al. [1997] are layered generators for domain-specific languages (an image processing language, a language for specifying data transfer on remote procedure calls, etc.). Domain-specific languages (or language extensions) are called *aspect languages* in AOP terminology, and generators are called *aspect weavers*.

An aspect, just like a collaboration, expresses a refinement that affects multiple classes of an application. In this sense, mixin layers can be regarded as an aspect-oriented implementation technique. Nevertheless, it is perhaps hard to find cross-cutting software implementation techniques that would *not* qualify as "aspect-oriented." The term has nowadays acquired broad meaning and encompasses many different techniques. We view using "aspect-oriented" terminology as purely a matter of taste. Certainly, the cross-cutting software development ideas pre-date the introduction of "aspect-orientation."

5.5 Adaptive OO Components

Another approach to modular OO software development is Lieberherr's *Demeter* method and adaptive components [Lieberherr 1996; Lieberherr and Patt-Shamir 1997; Mezini and Lieberherr 1998]. Adaptive components specify functionality additions based on an abstract pattern of participating classes. The pattern can later be applied to actual classes of an application to extend their capabilities. This technique is analogous to identifying collaborations in an object-oriented design, only now collaborations are implementation-level entities. Note that mixin layers offer the same flexibility, through the concept of adaptor layers discussed in Section 3.2.1. An important difference is that adaptor layers are themselves mixin layers. That is, with mixin layers, both the representation of a collaboration, and the representation of a collaboration application are the same (namely, mixin layers).

Nevertheless, the work on adaptive components reveals an interesting direction of research, with no counterpart in our work. Adaptive components can be declared by a *strategy*. That is, a strategy is a way to declaratively specify a path through the *class graph* (the graph induced on classes by inheritance and containment relationships among them). Along each node in the strategy, extra methods can be added. In this way, strategies are compact ways of expressing functionality additions to many classes. For example, one can easily specify new methods to be added to a class *and all its superclasses*. Similarly, assume that class `A` has a member variable that can hold an instance of class `B`, which, in turn, may hold an instance of class `C`. Using strategies, a programmer can describe the path from `A` to `C` in the class graph. (Class `B` does not need to be specified explicitly.) An adaptive component employing this strategy can then define a new method to be added to all three classes. Thus, strategies are a higher-level way of specifying collaborations (refinements); mixin layers could be used to implement strategies.

5.6 Design Patterns for Modularization

The *visitor* design pattern [Gamma et al. 1995] serves modularization purposes similar to mixin layers. Visitor is a pattern allowing a *functional* style of programming in object-oriented languages: multiple definitions of the same operation (applicable to objects of several different classes) can be grouped together in a visitor class, instead of these methods being distributed over individual classes. Visitor is a fundamental modularization mechanism and has been used to implement more sophisticated techniques (e.g., Mezini and Lieberherr [1998]).

Visitors are different from mixin layers in two ways. First, visitors are dynamic in nature, whereas mixin layers are static. This means that mixin layers can be used to add state (i.e., member variables) to the classes they refine. (For instance, imagine a class describing a graph node. If one wants to maintain the information "`is_marked`" for all nodes, this is easier to do with mixin layers: an `is_marked` field can be added in a mixin and carried in every single refined node object. With a visitor-based approach, this information must be maintained in a table on the side.) Additionally, visitors, unlike mixin layers, impose a run-time overhead. Second, visitors are not allowed to access the internals of the classes they extend. In contrast, mixin layers define subclasses of the refined classes. Hence, mixin layers are often able to access more implementation details than visitors. For instance, a C++ class may export a fairly extensive interface to its subclasses (using the `protected` keyword), without making the same interface public so that visitors can use it. This issue commonly arises when other design patterns (e.g., *singleton*) are used in conjunction with the visitor pattern.

Visitors, like many other design patterns, express refinements of objects or classes. Although not a design pattern, a mixin layer can be viewed as an elegant way of expressing a collaboration pattern among classes so that it is clear at the language level. Mixin layers can be expressed with the aid of a type system, rather than bypassing it, so that more compile-time checking and optimization is possible.

5.7 Subjectivity

Objects written for one application may not be reusable in another because their interfaces are different, even though both applications may deal with what is fundamentally the same object. The principle of *subjectivity* asserts that no single interface can adequately describe any object; objects are described by a family of related interfaces [Harrison and Ossher 1993; Ossher and Harrison 1992; Ossher et al. 1995]. The appropriate interface for an object is application-dependent (or *subjective*).

Subjectivity arose from the need for simplifying programming abstractions—for example, defining views that emphasize relevant aspects of objects and that hide irrelevant details. Ossher and Harrison took an important step further by recognizing that application-specific views of inheritance hierarchies can be automatically produced by composing different "subjects" [Harrison and

Ossher 1993]. Subjects encapsulate a primitive aspect or “view” of a hierarchy, whose implementation requires a set of additions (e.g., new data and method members) to one or more classes of the hierarchy.

Collaboration-based designs and mixin layers are analogous to subjectivity and subjects. Nevertheless, even though the goals are common, different parts of the problem are emphasized in the two approaches. The biggest difference between subject-oriented programming and our approach is that a subject-oriented approach aspires to combine programs that are developed completely independently. Mixin layers focus on a different problem: the consistent refinement of groups of classes, in order to raise the level of programming from single-class to multiple-class components. Mixin layers need to be developed with interoperability in mind. This makes mixin layers a more general technique, but with a lower degree of automation and little applicability to prewritten software—manual adaptation is required.

6. CONCLUSIONS

Improved modularizations are the key to improved component-based software development. We and others have observed that traditional notions of modularization—method, class, package—are inadequate for this purpose. Many different results in modularization point to large-scale refinements—the ability to encapsulate and modularize fragments of classes and methods—as the basis for next-generation modularizations. The core is the idea of refinement as the centerpiece for component-based software development. Our refinements are large-scale: a single refinement can update multiple classes of an application, and a composition of a few refinements specifies a complete implementation of an application.

The fragments of classes and methods that need to be encapsulated are *not arbitrary*. Rather, fragments are encapsulated together when they all define how a particular service or feature, which can be shared by many applications of a domain, is implemented. That is, these fragments must have meaningful expressions in software designs. We have shown that the object-oriented concept of collaboration-based designs captures this idea. A collaboration is an abstract design that specifies roles for different classes of objects, and defines protocols by which objects of these classes interact to realize a particular service or feature. Collaborations are the way large-scale (i.e., multi-class) refinements are expressed in object-oriented models. Applications are typically defined by compositions of a small number of reusable collaborations.

We have shown how collaborations can be defined and composed statically using existing programming language constructs, and how they can be supported by new language constructs. We presented a particular way of expressing large-scale refinements as *mixin layers*, a name chosen to emphasize its connection to the common *mixin* concept in object-oriented languages. We showed how mixin layers overcame the scalability difficulties that plagued prior work. They rely on a novel combination of parameterized inheritance, and class nesting, in effect generalizing the concept of a package (set of classes), so that parameterized packages could participate in inheritance lattices. As an example, we showed how mixin layers were used as the primary implementation technique for building an extensible compiler for the Java language.

REFERENCES

AGESEN, O., FREUND, S. N., AND MITCHELL, J. C. 1997. Adding type parameterization to the java language. In *Conference Proceedings of OOPSLA '97, Atlanta*. ACM SIGPLAN Notices *32*, 10. ACM, 49–65.

BATORY, D., CARDONE, R., AND SMARAGDAKIS, Y. 2000a. Object-oriented frameworks and product lines. In *Proceedings of the First Software Product Line Conference*, P. Donohoe, Ed. 227–247.

BATORY, D. AND GERACI, B. 1997. Composition validation and subjectivity in GenVoca generators. *IEEE Trans. Softw. Eng. 23*, 2 (Feb.), 67–82.

BATORY, D., JOHNSON, C., MACDONALD, B., AND VON HEEDER, D. 2000b. Achieving extensibility through product-lines and domain-specific languages: A case study. In *Proceedings of the Sixth International Conference on Software Reuse*, W. B. Frakes, Ed. 117–136.

BATORY, D., LOFASO, B., AND SMARAGDAKIS, Y. 1998. JTS: Tools for implementing domain-specific languages. In *Proceedings: Fifth International Conference on Software Reuse*, P. Devanbu and J. Poulin, Eds. IEEE Computer Society Press, 143–153.

BATORY, D. AND O'MALLEY, S. 1992. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol. 1*, 4 (Oct.), 355–398.

BATORY, D., SINGHAL, V., SIRKIN, M., AND THOMAS, J. 1993. Scalable Software Libraries. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering.* 191–199.

BATORY, D. S. 1987. Concepts for a database system synthesizer. In *Symposium on Principles of Database Systems (PODS '88)*. ACM Press, New York, 184–192.

BATORY, D. S., BARNETT, J. R., GARZA, J. F., SMITH, K. P., TSUKUDA, K., TWICHELL, B. C., AND WISE, T. E. 1988. GENESIS: An extensible database management system. *Software Engineering 14*, 11, 1711–1730.

BIGGERSTAFF, T. J. 1994. The library scaling problem and the limits of concrete component reuse. In *Proceedings: 3rd International Conference on Software Reuse*, W. Frakes, Ed. IEEE Computer Society Press, 102–109.

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *OOPSLA/ECOOP '90 Proceedings*, N. Meyrowitz, Ed. ACM SIGPLAN, 303–311.

BRACHA, G. AND GRISWOLD, D. 1996. Extending smalltalk with mixins. Workshop on Extending Smalltalk at OOPSLA 1996. See `http://java.sun.com/people/gbracha/mwp.html`.

BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, C. Chambers, Ed. ACM SIGPLAN Notices *33*, 10. Vancouver, BC, 183–200.

CARDELLI, L. AND WEGNER, P. 1985. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec.), 471–522.

CHIBA, S. 1996. Open C++ programmer's guide for version 2. Tech. Rep. SPL-96-024, Xerox PARC.

COGLIANESE, L. AND SZYMANSKI, R. 1993. DSSA-ADAGE: An environment for architecture-based avionics development. Proceedings of the NATO AGARD Conference.

CUNNINGHAM, W. AND BECK, K. 1989. Constructing Abstractions for Object-Oriented Applications. *J. Obj. Orient. Program.*, July 1989.

FINDLER, R. B. AND FLATT, M. 1998. Modular object-oriented programming with units and mixins. In *International Conference on Functional Programming (ICFP '98)*. 94–104.

FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. 1998. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*. ACM, New York, NY, 171–183.

FORMAN, I. R., DANFORTH, S., AND MADDURI, H. 1994. Composition of before/after metaclasses in SOM. In *Proceedings of OOPSLA '94.* ACM Sigplan Notices, vol. 29. Portland, 427–439.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns.* Addison Wesley, Reading, MA.

HABERMANN, A. N., FLON, L., AND COOPRIDER, L. W. 1976. Modularization and hierarchy in a family of operating systems. *Commun. ACM 19*, 5 (May), 266–272.

HARRISON, W. AND OSSHER, H. 1993. Subject-oriented programming (A critique of pure objects). In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications.* ACM Press, Los Alamitos, CA, USA, 411–28.

HEIDEMANN, J. S. AND POPEK, G. J. 1994. File-system development with stackable layers. *ACM Trans. Comput. Syst. 12*, 1 (Feb.), 58–89.

HELM, R., HOLLAND, I. M., AND GANGOPADHYAY, D. 1990. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*, 169–180. Published as ACM SIGPLAN Notices, volume 25, number 10.

HOLLAND, I. M. 1992. Specifying Reusable Components Using Contracts. In *Proceedings of the ECOOP '92 European Conference on Object-oriented Programming*, O. L. Madsen, Ed. LNCS 615. Springer-Verlag, Utrecht, The Netherlands, 287–308.

International Organization for Standardization 1995. *Ada 95 Reference Manual. The Language. The Standard Libraries.* International Organization for Standardization. ANSI/ISO/IEC-8652:1995.

JOHNSON, R. E. AND FOOTE, B. 1988. Designing reusable classes. *J. Obj. Orient. Program. 1*, 2, 22–35.

KELLER, R. AND HÖLZLE, U. 1998. Binary component adaptation. In *ECOOP '98—Object-Oriented Programming*, E. Jul, Ed. Lecture Notes in Computer Science, vol. 1445. Springer, 307–329.

KICZALES, G., DES RIVIERES, J., AND BOBROW, D. G. 1991. *The Art of the Meta-Object Protocol.* MIT Press, Cambridge, MA, USA.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *ECOOP '97—Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 1241. Springer, 220–242.

LIEBERHERR, K. AND PATT-SHAMIR, B. 1997. Traversals of object structures: Specification and efficient implementation. Tech. Rep. NU-CCS-97-15, College of Computer Science, Northeastern University.

LIEBERHERR, K. J. 1996. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company.

MADSEN, O. L. AND MØLLER-PEDERSEN, B. 1989. Virtual classes: A powerful mechanism in object-oriented programming. In *OOPSLA '89 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, N. Meyrowitz, Ed. ACM Press, 397–406.

MEZINI, M. 1997. Dynamic object evolution without name collisions. In *ECOOP'97—Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 1241. Springer, 190–219.

MEZINI, M. AND LIEBERHERR, K. 1998. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98).* ACM SIGPLAN Notices, vol. 33, 10. ACM Press, New York, 97–116.

MILNER, R., TOFTE, M., AND HARPER, R. W. 1990. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts.

MONTLICK, T. 1996. Implementing mixins in smalltalk. The Smalltalk Report (July).

MOON, D. A. 1986. Object-oriented programming with *flavors.* In *OOPSLA '86 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, N. Meyrowitz, Ed. ACM SIGPLAN, ACM Press, 1–8.

MYERS, A. C., BANK, J. A., AND LISKOV, B. 1997. Parameterized types for Java. In *Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM SIGACT and SIGPLAN, ACM Press, 132–145.

Odersky, M. and Wadler, P. 1997. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* Paris, France, 146–159.

O'Malley, S. W. and Peterson, L. L. 1992. A dynamic network architecture. *ACM Trans. Comput. Syst. 10*, 2 (May), 110–143.

Ossher, H. and Harrison, W. 1992. Combination of Inheritance Hierarchies. In *Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications.* 25–40. Published as ACM SIGPLAN Notices, volume 27, number 10.

Ossher, H., Kaplan, M., Harrison, W., Katz, A., and Kruskal, V. 1995. Subject-oriented composition rules. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 235–250.

Parnas, D. L. 1979. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng. SE-5*, 2 (Mar.), 128–138.

Reasoning Systems 1990. *Dialect User's Guide.* Reasoning Systems.

Reenskaug, T., Anderson, E., Berre, A., Hurlen, A., Landmark, A., Lehne, O., Nord-hagen, E., Ness-Ulseth, E., Oftedal, G., Skaar, A., and Stenslet, P. 1992. OORASS: Seamless support for the creation and maintenance of object-oriented systems. *Journal of Object-Oriented Programming 5*, 6 (Oct.), 27–41.

Rumbaugh, J. 1994. Getting started: Using use cases to capture requirements. *J. Obj. Orient. Program. 7*, 5 (Sept.), 8–23.

Smaragdakis, Y. 1999. Implementing large-scale object-oriented components. Ph.D. dissertation, University of Texas at Austin.

Smaragdakis, Y. and Batory, D. 1998. Implementing layered designs with mixin layers. In *Proceedings ECOOP'98*, E. Jul, Ed. LNCS 1445. Brussels, Belgium, 550–570.

Smaragdakis, Y. and Batory, D. 2000. Mixin-based programming in C++. In *GCSE'00—Generative and Component-Based Software Engineering Symposium*. Lecture Notes in Computer Science, vol. 2177. Springer, 163–177.

Steyaert, P., Codenie, W., D'Hondt, T., Hondt, K. D., Lucas, C., and Limberghen, M. V. 1993. Nested Mixin-Methods in Agora. In *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, O. Nierstrasz, Ed. LNCS 707. Springer-Verlag, Kaiserslautern, Germany, 197–219.

Stroustrup, B. 1997. *The C++ Programming Language*, 3 ed. Addison-Wesley, Reading, Mass.

Sun Microsystems 1997. *Java Inner Classes Specification*. Sun Microsystems. In `http://java.sun.com/products/jdk/1.1/docs/`.

Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr, S. M. 1999. N Degrees of Separation: Multi-dimensional Separation of Concerns. In *Proceedings of ICSE'99*. Los Angeles CA, USA, 107–119.

Thorup, K. K. 1997. Genericity in Java with virtual types. In *ECOOP'97—Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 1241. Springer, 444–471.

VanHilst, M. 1997. Role-oriented programming for software evolution. Ph.D. dissertation, University of Washington, Seattle, Washington.

VanHilst, M. and Notkin, D. 1996a. Decoupling change from design. In *SIGSOFT'96: Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, D. Garlan, Ed. ACM Press, 58–69.

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

VanHilst, M. and Notkin, D. 1996c. Using role components to implement collaboration-based designs. In *OOPSLA '96 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, 359–369.

Villarreal, E. 1994. Automated compiler generation for extensible data languages. Ph.D. dissertation, University of Texas at Austin.

Weihe, K. 1997. A software engineering perspective on algorithmics. In `http://www.informatik.uni-konstanz.de/Preprints/`.

WEISE, D. AND CREW, R. 1993. Programmable syntax macros. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*. 156–165.

WEISS, D. M. 1990. Synthesis operational scenarios. Tech. Rep. 90038-N, Version 1.00.01, Software Productivity Consortium, Herndon, Virginia.

WILE, D. 1993. Popart: Producer of parsers and related tools. Tech. rep., USC/Information Sciences Institute.

WIRTH, N. 1977. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM 20*, 11 (Nov.), 822–823.

Received January 2001; revised September 2001; accepted October 2001