Implementing Layered Designs with Mixin Layers[1]

Yannis Smaragdakis and Don Batory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{smaragd, dsb}@cs.utexas.edu

Abstract. *Mixin layers* are a technique for implementing layered object-oriented designs (e.g., collaboration-based designs). Mixin layers are similar to abstract subclasses (mixin classes) but scaled to a multiple-class granularity. We describe mixin layers from a programming language viewpoint, discuss checking the consistency of a mixin layer composition, and analyze the language support issues involved.

1 Introduction

The complexity of software has driven both researchers and practitioners toward design methodologies that decompose design problems into intellectually manageable pieces and that assemble partial products into complete software artifacts. The principle of separating logically distinct and (largely independent) facets of an application is behind many good software design practices. A key objective in designing reusable software modules is to encapsulate within each module a single (and largely orthogonal) aspect of application design. Many design methods in the object-oriented world build on this principle of design modularity (e.g., *design patterns* [12] and *collaboration-based* designs [7][14][15][28][37]). The central issue is to provide implementation (i.e., programming language) support for expressing modular designs concisely.

Our work addresses this problem in the context of collaboration-based (or *role-based*) designs. Such designs decompose an object-oriented application into a set of classes *and* a set of *collaborations*. Each application class encapsulates several *roles*, where each role embodies a separate aspect of the class's behavior. A cooperating suite of roles is called a *collaboration*. In collaboration-based designs, collaborations express distinct (and largely independent) aspects of an application. This property makes collaborations an interesting way to express software designs in a modular way. Collaboration-based design is an example of a *layered* design methodology: collaborations are components and layered compositions of these components define an application.

While collaboration-based designs cleanly capture different aspects of application behavior, their implementations often do not preserve this modularity. *Application frameworks* [17] are a standard implementation technique. As shown in [37], frameworks not only do not preserve the design structure but also may result in inefficient implementations, requiring excessive use of dynamic binding. VanHilst and Notkin

[1] We gratefully acknowledge the sponsorship of Microsoft Research, the Defense Advanced Research Projects Agency (Cooperative Agreement F30602-96-2-0226), and the University of Texas at Austin Applied Research Laboratories.

proposed an alternative technique [37][38][39] using *mixin classes* [8] in C++. Their approach mapped design-level entities (roles) directly into implementation components (mixin classes). It suffered, however, from highly complex parameterizations in the presence of multiple classes, and the inability to contain intra-collaboration design changes. This caused them to question its scalability [37], and seek a way to explicitly capture collaborations as distinct implementation entities.

Our work shows how to remove the difficulties of the VanHilst and Notkin method by scaling the concept of a mixin to multiple classes. We call these scaled entities *mixin layers*. A mixin layer can be viewed as a mixin class encapsulating other mixins with the restriction that the parameter (superclass) of an outer mixin must encapsulate all parameters of inner mixins. We will use C++ templatized nested classes as our primary means of expressing mixin layers, but the ideas are not specific to C++. We will discuss in detail the language support issues involved, mainly relative to C++, CLOS, and Java. There are several examples of designs that can be expressed using mixin layers (e.g., see the enumeration in [2], as well as [4], [15], [31]). We will illustrate a simple example in this paper.

The primary emphasis of this paper is on mixin layers from a programming language standpoint. In a previous paper [33] we studied the applications of a particular implementation of mixin layers. Here, we will consider how the mechanism depends on specific language features and how it addresses fundamental problems associated with layered object-oriented implementations. Such problems include verifying the consistency of a composition of layers and handling the propagation of type information from a subclass to a superclass.

2 Background

2.1 Layered Designs

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) that determines how these objects interact. The part of an object enforcing 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 the corresponding objects. Essentially, 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, we try 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

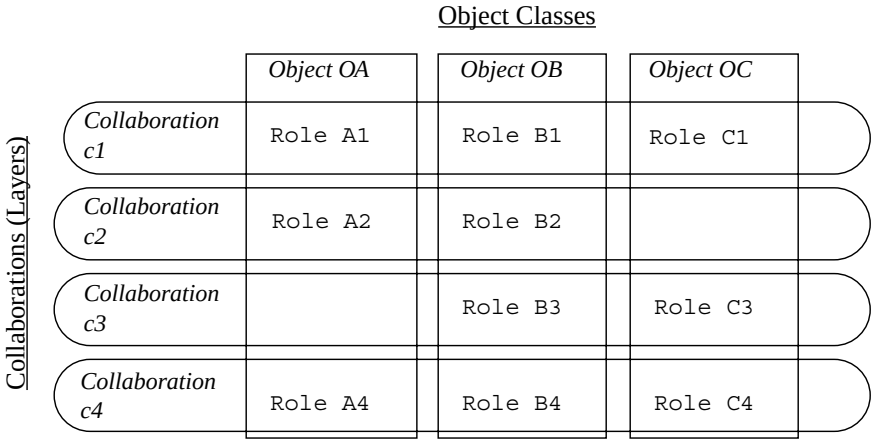

Figure 1: Example collaboration decomposition. Ovals represent collaborations, rectangles represent objects, their intersections represent roles.

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.

Figure 1 depicts the overlay of objects and collaborations in a design. The figure contains 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. To do this, collaborations have to prescribe certain roles for 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 — *c2* does not affect object *OC*.

Collaboration-based designs are an example of layered designs. In this paper we will concentrate on collaboration-based designs, but a more general classification of layered designs can be found in [2] and [33]. Additionally, the designs we will examine (as well as those examined by VanHilst and Notkin) are *static*: the roles played by an object are uniquely determined by its class. For instance, in Figure 1, all three objects must belong in different classes (since they all support different sets of roles).

2.2 Mixin Classes

The term "mixin class" (or just "mixin") has been overloaded to mean several specific programming techniques and a general mechanism that they all approximate. Here we will use "mixin" in the general sense of [8]. Common alternative meanings include CLOS classes whose superclasses are determined by linearization of multiple inheritance, as well as C++ classes used in a specific multiple inheritance pattern (as superclasses of a single class that themselves have a common "virtual base class").

The main idea implemented by mixins is quite simple: in object-oriented lan-

guages, a superclass can be defined without specifying its subclasses. This is not, however, symmetric: when a subclass is defined, it must have a specific superclass. Mixins (also commonly known as *abstract subclasses* [8]) represent a mechanism for specifying classes that will 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 of mixins makes them appropriate for defining uniform incremental extensions for a multitude of classes. When the mixin is instantiated with one of these classes as a superclass, it produces a class incremented with the additional behavior.

Mixins can easily be implemented using parameterized inheritance. In this case, a mixin is a parameterized class with the parameter becoming its superclass. Using C++ syntax we can write a mixin as:

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

This was the primary implementation technique used by VanHilst and Notkin in their approach to mapping collaboration-based designs into programs. *Their mixin classes represented roles* and were also parameterized by any other classes that interacted with the given role in its collaboration. For instance, role `B4` in Figure 1 would be expressed as:

```
template <class RoleSuper, class OA, class OC>
class B4 : public RoleSuper {
   ... /* role implementation, using OA, OC */
};                                                              (1)
```

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. The programmer has to explicitly keep track of the mapping between roles and classes. Additionally, the programmer has to make explicit 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*. These limitations make the approach unscalable. As we show in [33], the length of parameterization expressions increases exponentially with the number of different roles for a class. This is illustrated in Appendix A by showing the parameterization code from an example in [37].

Conceptually, the scalability problems of the VanHilst and Notkin approach are due to the small granularity of the entities they represent. In their methodology, each mixin class represents a 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 (1). VanHilst and Notkin acknowledged this limitation [37]. They suggested trying to map entire collaborations to implementation entities as future work. This is accomplished with the concept of a mixin layer.

3 Mixin Layers

We solve the scalability problems identified by VanHilst and Notkin by implementing collaborations as *mixins that encapsulate other mixins*. We will call the encapsulated mixin classes *inner mixins*, and the mixin that encapsulates them the *outer mixin*. Inner mixins can be inherited, just like 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*[2]. This is illustrated in Figure 2. `ThisMixinLayer` is a mixin that refines (through inheritance) `SuperMixinLayer`. `SuperMixinLayer` encapsulates three classes: `FirstClass`, `SecondClass`, and `ThirdClass`. `ThisMixinLayer` also encapsulates three inner classes that are themselves mixins and refine the corresponding classes of `SuperMixinLayer`.

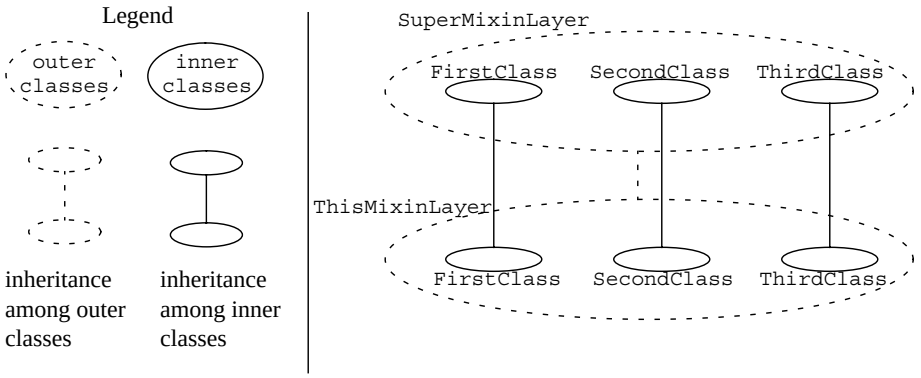

Figure 2: Mixin layers schematically.

We can conceptually see how some of the problems of the VanHilst and Notkin method are addressed if collaborations are expressed as mixin layers: classes that do not participate in a certain collaboration are inherited from collaborations above (we will subsequently use the term "collaboration" for the mixin layer representing a collaboration when no confusion can result). This way every collaboration can be expressed in terms of its superclass collaboration only (since the superclass defines roles for all objects, either explicitly or by inheritance). Parameters like `OA` and `OC` in example (1) are not needed — a role can directly refer to other roles in the same collaboration.

We would like to support mixin layers using the same language mechanisms as those used for mixin classes. To do this, we can standardize the names used for role implementations (make them the same as the name of the class that plays them). This yields an elegant form of mixin layers that can be expressed using common programming language features. For instance, using C++ parameterized inheritance and nested classes, we can express a mixin layer (see again Figure 2) implementing a collaboration as:

---

[2] Inner mixins can actually themselves be mixin layers.

```
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
};                                                              (2)
```

The code fragment in (2) represents the form of mixin layers that we will use in this paper. Note that specifying a parameter for the outermost mixin automatically determines the parameters of all inner mixins.

In (2) 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. *None of these implementation choices is part of the mixin layer definition*. There are very different ways of encoding the same design. For example, we can encode mixin layers in CLOS with different implementation dependencies. Class encapsulation is implemented by defining member methods that return CLOS class-metaobjects. No lexical nesting of any kind (as in (2)) is necessary. This combines nicely with the method-based character of CLOS mixins and the reflective capabilities of the language. (To keep the discussion short, we put the CLOS counterpart of code fragment (2) in Appendix B). Note the importance of the above discussion: mixin layers are not a linguistic idiom. Many flavors of the mixin layer 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, etc. We will use a C++ idiom in our examples, in the belief that concrete syntax will clarify, rather than obscure, our ideas.

Back to (2), composing mixin layers to form concrete classes is now as simple as composing mixin classes. If, like in Figure 1, we have four mixin layers (`Collab1`, `Collab2`, `Collab3`, `Collab4`) implementing four different collaborations, we can compose them as `Collab4 < Collab3 < Collab2 < Collab1 > > >`[3]. Note that even though some collaborations (like *c2*, *c3* in Figure 1) do not specify roles for all classes, they inherit such roles from their superclasses. This results in linear length expressions for collaboration-based designs and, hence, solves the scalability problems of the VanHilst and Notkin approach (see Appendix A).

4 A Concrete Example

In this section, we will show the benefits of mixin layers through an example. More complex examples are presented in [6] and [33].

4.1 A Data Structure

We examine a data structure design that was used in both the P2 lightweight

[3] `Collab1` will then have to be a concrete class (i.e., not a mixin). Alternatively we can have a collaboration of empty roles that we use as the root of all compositions.

DBMS generator [3][4], as well as in the DiSTiL library for data structures [32]. In this example we add functionality to a data structure by assigning more roles to the classes that participate in the design. There are two such classes: a *node* class, of which all data nodes are instances, and a *container* class, which has one instance per data structure. A third class for data structure cursors (iterators) is generally needed but to keep the example simple we will equate cursors with pointers to node objects. This model for data structure construction is, in fact, quite general. Composite data structures, run-time bound checks, garbage collection, a lock and transaction manager, etc., can all be specified as new roles for the node and container classes (see [4]). This can be achieved using mixin layers, as we will show with extensions to a binary tree data structure.

Our target data structure consists of four different collaborations: *bintree*, *alloc*, *timestamp*, and *sizeof*. *Bintree* captures the functionality of a binary tree. *Alloc* captures the functionality of memory allocation. *Timestamp* is responsible for maintaining timestamps for data structure and element updates. *Sizeof* simply keeps track of the data structure size. The design is simple and we will not concern ourselves with its schematic representation (in the form of Figure 1) or the way we obtained it. For a good reference on how to obtain collaboration-based designs from use-case scenarios [29] see VanHilst's Ph.D. dissertation [40].

A mixin layer implementing a binary tree collaboration has the form[4]:

```
template <class Super> class BINTREE : public Super {
public:
   class Node : public Super::Node {
      Node* parent_link,
         left_link, right_link ;   // Node data members
   public:
      ...                          // Node interface
   };

   class Container : public Super::Container {
      Node* header;                // Container data members
   public:
      void insert ( EleType el ) { ... }
                          // Definition of EleType inherited
      void erase ( Node* node ) { ... }
      bool find ( EleType* el ) { ... }
      ...                          // Other methods
   };
};
```

Note that the `Container` class is aware of the `Node` class (e.g., it declares a member variable of type `Node*`). The two classes must be designed together and, hence, it makes sense to encapsulate both in a single unit.

---

[4] We will present simplified code fragments, ignoring implementation details that are not directly relevant to our discussion. We will highlight class definitions for readability and use ellipses (. . .) for omitted code.

Now consider the implementation of the *timestamp* collaboration: the data structure maintains the time of its last update, as well as the creation and update time of each node. The set of exported operations on the data structure can be enriched (e.g., by defining an operation that returns the data structure update time, as well as a variant of `find`: `find_newer`). This enrichment can be viewed as a collaboration prescribing roles for both the `Node` and the `Container` class. Its implementation using mixin layers has the form:

```
template <class Super> class TIMESTAMP : public Super {
public:
   class Node : public Super::Node {
      time_t creation_time, update_time; // Node data members
   public:
      bool more_recent (time_t t) { ... }
      ...                          // Other time-related methods
   };

   class Container : public Super::Container {
      time_t update_time;          // Container data members
   public:
      bool find_newer ( EleType* el, time_t t ) { ... }
      void insert ( EleType el ) { ... }
      ...                          // Other time-related methods
   };
};
```

Not all collaborations need to specify roles for all classes in a design. The *sizeof* collaboration, for instance, only needs to maintain a counter of elements associated with a container and only prescribes a role for the `Container` class. It can be implemented as a mixin layer that is a trivial wrapper around a mixin class:

```
template <class Super> class SIZEOF : public Super {
public:
   class Container : public Super::Container {
      int count;                   // Container data members
   public:
      Container() : count(0), Super::Container() {};
                                   // Constructor
      void insert ( EleType el ) {
         Super::Container::insert(el); count++; }
      void erase ( Node* node ) {
         Super::Container::erase(el); count--; }
      int size () { return count; }
   };
};
```

Again, classes generated by instantiating the `SIZEOF` mixin layer do have a `Node` nested class — this class is inherited from mixin layers above `SIZEOF` in the inheritance chain.

To put everything together we need a concrete (i.e., non-mixin) class to be the root of our inheritance hierarchy. This could be a “dummy” class, containing only empty

roles. In most applications, however, it is easy to identify a collaboration, which has to be the basis upon all other functionality is built. In this particular example, the *alloc* collaboration serves this purpose. *Alloc* is responsible for the actual memory allocation for the data structure. Note that the implementation of this collaboration (as well as any of the other mixin layers) can have parameters other than the one we used to designate the superclass. These extra parameters can be used to specify polymorphic behavior. In our example, it makes sense to parameterize the layer representing *alloc* by the type of the elements stored in the data structure. Then we have:

```
template <class EleType> class ALLOC {
public:
   class Node {
      EleType element;  // The actual stored data
   public:
      ...               // Any methods pertaining to stored data
   };

   class Container {
   protected:
      typedef Super::EleType EleType;
                        // The actual type of stored data
      void* node_alloc();
      ...               // Other allocation methods
   };
};
```

We form our target data structure by composing mixin layers. A binary tree storing integers and maintaining time information and size is defined as:

```
typedef SIZEOF < TIMESTAMP < BINTREE < ALLOC < int > > > > >
 Tree1;                                                          (3)
```

The `Node` and `Container` classes are accessible[5] as `Tree1::Node` and `Tree1::Container`. An outline of the composition of (3) is shown in Figure 3. We have annotated the design with some of the inherited member variables and methods. Note how both the `SIZEOF` and the `TIMESTAMP` mixin layers depend on layers above them to insert and erase elements from the data structure. We will return to this later.

4.2 Discussion

Our example illustrates the benefits of mixin layers:

- *Mixin layers preserve the structure of the design.* This enhances the maintainability of an application. If changes are introduced in the design (e.g., in our data structure example, if we want to use a different form of a binary tree, if we want to maintain time information for retrievals as well, etc.) it is easy to isolate them. A single mixin layer encapsulates changes to multiple classes. Additionally, the specification of the inheritance hierarchy is separate from the definition of class

[5] There is no reason why the `Node` class should be user accessible. What really needs to be user accessible is an iterator class, which for this example is the same as a pointer to a `Node` object.

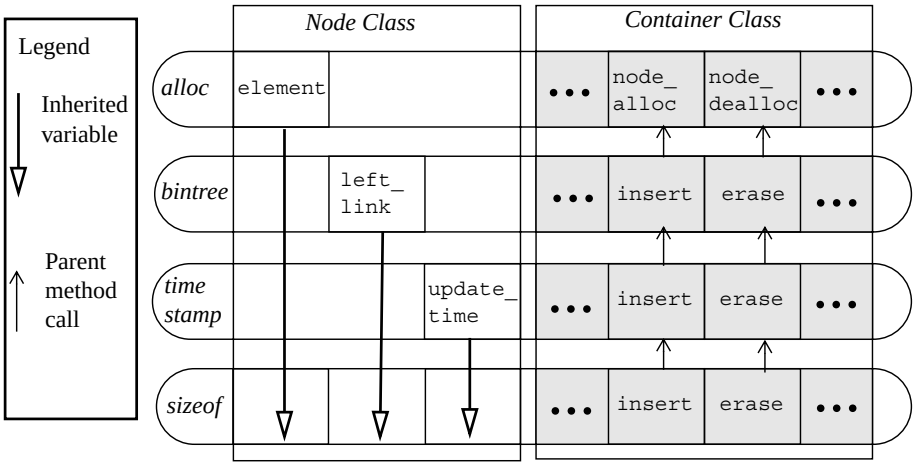

Figure 3: A composite data structure. Shaded areas represent roles.

functionality. Hence, changing the inheritance chain is as simple as editing a composition like (3), above.

- *Mixin layers are reusable and interchangeable*. A single layer can be used in several different compositions and is, to an extent, isolated from other layers. In our data structure example, the `SIZEOF`, `ALLOC`, and `TIMESTAMP` layers are not specific to binary trees. They could just as well be used with a doubly-linked list or many other data structures. The change is minimal: we only have to swap the `BINTREE` layer with a different layer.
- *Mixin layer compositions are scalable*. In our example, if we have more than one variation of a data structure in the same application, very little hand editing is involved. For instance, we could have a second binary tree that maintains no time information in the same program. The definition would be:

  ```
  typedef SIZEOF < BINTREE < ALLOC < int > > >  Tree2;
  ```

  Consider what would happen if application frameworks were used to implement our design. (Frameworks specify superclasses and implementations are defined via subclassing). To express the second data structure, we would have to explicitly subclass from the binary tree abstract class and reintroduce by hand the changes dictated by the `SIZEOF` class. In general, application frameworks cannot express more than one feature variation without code replication. Mixin layers, on the other hand, can be composed in an exponential number of ways to express a large variety of implementations (see also [3]).

Such benefits have usually been claimed for techniques that group many objects into large-scale components (e.g., [2], [13], [19], [31]). In these approaches, grouping objects into components was not done with existing object-oriented mechanisms. Mixin layers can be used to express similar functionality using a novel combination of object-oriented constructs.

5 Discussion and Related Work

We glossed over several important issues in our example of Section 4. In this section, we consider the issues of compositional consistency, virtual types, and language support for mixin layers, with references to related work.

5.1 Composition Consistency

The most important issue arising in mixin layer composition is ensuring composition correctness. Some mixin layers depend on the existence or the right ordering of others. Many problems can be detected immediately. As shown in Figure 3, the `BINTREE` layer calls the allocator directly for every element insertion (i.e., it does not propagate the `insert` and `erase` operations). Omitting the `BINTREE` layer in (3) will cause a compilation error in C++: operations like `insert` that are propagated by `SIZEOF` and `TIMESTAMP` will be undefined. Other problems, however, are more subtle. Consider reordering the `BINTREE` and `SIZEOF` layers in a composition:

```
typedef BINTREE < SIZEOF < ALLOC < int > > >  Tree3;
```

This will cause the `insert` and `erase` methods of `SIZEOF` to be shadowed (overridden) by those of `BINTREE`. Hence, the implementation is wrong: the count of elements in the data structure will never be updated (since this is only done in the `insert` and `erase` methods of `SIZEOF` and these methods are not called by `BINTREE`). The `size` operation will be visible, however, and will always return 0, although the data structure may contain elements.

Such mistakes may not actually cause a compilation error. This can be true even for statically typed languages — in C++, for instance, no error will be signalled even though `SIZEOF::Container` has an explicit call to the `insert` method of its superclass and no such method is defined. This has to do with the treatment of methods in parameterized classes as function templates, as we will discuss in Section 5.3. In essence, the `insert` method for `SIZEOF::Container` is never compiled since it is not needed, thus the error is never discovered.

In general, mixin layers may have subtle semantic dependencies that are not reflected in their interfaces. In large libraries there may be a variety of layers supporting identical interfaces but implementing different semantics. Many combinations of layers may be illegal but there may not be a way to detect this from the interfaces alone.

This problem has been studied in the context of layered systems. The *design rule checking* approach of [5] offers a solution using propositional properties and requirements that are propagated both up and down a layer hierarchy. The *nested mixin-methods* of [34] resulted in a powerful constraint system. Nesting of mixins was used as a way to restrict their scope. A mixin class of [34] can define other mixins that can be composed with it, inherit some mixins when composed, and cancel inherited mixins. The *feature-oriented* programming approach of [27] uses the `assumes` keyword to express the property that the correctness of one feature (layered component) assumes the existence of another.

Interestingly enough there is a simple way to express basic dependencies within the mixin layers framework. Every mixin layer can export propositional properties

describing its behavior (essentially encoding semantic knowledge in its interface). Recall that when mixin layers are composed, they are linked in an inheritance chain. Properties are propagated in the same direction as inherited methods and variables: from superclasses to subclasses. Layers can explicitly make inherited properties unavailable to their subclasses. Finally, a layer can check (require) whether it has inherited a property or not. A composition is correct if none of these requirements fails. This technique is similar to the `assumes` functionality of [27] and the design rule checking of [5]. Consider the example of Section 4. There are four requirements that we need to express:

- A `BINTREE` mixin layer cannot have a `SIZEOF` layer as an ancestor in its inheritance chain (because otherwise the `insert` method of `SIZEOF` will be shadowed).
- A `BINTREE` mixin layer cannot have a `TIMESTAMP` layer as an ancestor (same reason as above).
- A `SIZEOF` mixin layer needs to ensure that some sort of a data structure is present in the composition. In our example the only data structure is a binary tree but we can easily imagine the same mixin layer being composed, for instance, with a doubly linked list layer.
- A `TIMESTAMP` mixin layer also needs to ensure that a data structure is present.

These can be specified as requirements on the existence of three properties (inherited from ancestors in the inheritance chain):

- No `SIZEOF` layer is present (call this property `P_NoSizeof`).
- No `TIMESTAMP` layer is present (call this property `P_NoTimestamp`).
- A data structure layer is present (call this property `P_DataStructure`).

The implementation is simple. All properties can be expressed as empty classes encapsulated in a mixin layer. Properties are inherited but can be negated by using access control (that is, "hiding" of class members — e.g., by making them "private" members in C++). If the class representing the property is made visible to subclasses (either by declaration or by inheritance without "hiding"), then the property is asserted. Otherwise the property is negated. The requirement that a certain property be satisfied is then enforced by declaring an instance of this class.

In our example, `BINTREE` exports property `P_DataStructure` and requires properties `P_NoSizeof` and `P_NoTimestamp`.

```
template <class Super> class BINTREE : public Super {
protected:
   class P_DataStructure { };
         // Assert this property for subclasses
private:
   P_NoSizeof dummy1;
   P_NoTimestamp dummy2;
         // Require P_NoSizeof and P_NoTimestamp from ancestors
public:
   ...   // nested mixins (same as before)
};
```

The other three mixin layers are modified accordingly:

```
template <class Super> class SIZEOF : public Super {
private:
```

```
   class P_NoSizeof { };      // Negate property for subclasses
   P_DataStructure dummy1;    // Require P_DataStructure
public:
   ...                        // nested mixins (same as before)
};

template <class Super> class TIMESTAMP : public Super {
private:
   class P_NoTimestamp { };   // Negate property for subclasses
   P_DataStructure dummy1;    // Require P_DataStructure
public:
   ...                        // nested mixins (same as before)
};

template <class EleType> class ALLOC {
protected:
   class P_NoSizeof { };      // Assert property for subclasses
   class P_NoTimestamp { };   // Assert property for subclasses
public:
   ...                        // nested classes (same as before)
};
```

Note how the constraint is enforced: the ALLOC mixin layer asserts properties P_NoSizeof and P_NoTimestamp. The BINTREE layer requires that they not be negated by some layer between BINTREE and ALLOC in the inheritance hierarchy. SIZEOF and TIMESTAMP negate P_NoSizeof and P_NoTimestamp, respectively. Also they require that they have some ancestor asserting property P_Datastructure. This accurately describes the constraints we want to impose on the compositions of these four mixin layers: a BINTREE has to be present and if a TIMESTAMP or SIZEOF are present they must be descendants of BINTREE in the inheritance chain.

The method described above only makes use of access control (such as commonly found in C++ or Java and easily emulated in CLOS) and the same general language mechanisms used for mixin layers. The method's clarity could be improved using some form of syntactic sugar. In the absence of static typing (e.g., if we were to implement this technique in CLOS) the checking would have to be performed at run-time by calling an appropriate method. We have developed other constraint techniques for C++ but they are language-specific (or even compiler-specific as is the case with many compile-time techniques that rely on constant-folding).

There is a more important restriction of the technique we presented, however. Even though an erroneous composition will be detected, the error message will be far from informative. In essence, we express relatively deep errors (e.g., semantic incompatibilities among large scale components) as the absence of an inherited class. The compiler will still complain about an undefined type, but the cause of the error (not to mention a possible fix) is not immediately apparent. The problem is intensified in the case of mixin layers developed and used independently by different programmers. A casual user will expect much more expressive error reporting from a black-box component than our technique can offer. Reference [5] presents a general technique for automatically detecting (and suggesting repairs to) errors in layered implementations.

5.2 Virtual Types

An interesting issue arises in various layered implementations that use inheritance together with static typing (not necessarily in *fully* statically typed languages). This is essentially a symmetric problem to the one that originally motivated mixins. Recall that mixins were introduced to remove the restriction that the definition of a subclass in an inheritance relation needs to reference its superclass. This restriction, however, means that superclasses are generally known when a subclass is defined (and references to them may exist in subclass code) while the converse is not true. This is not a problem when a superclass only needs to transfer control to a subclass (i.e., when a superclass needs to call a subclass method). The usual dynamic binding (or *late binding*) of methods (the hallmark of object-oriented programming) deals with exactly this. When, however, superclass code depends on type information that is specific to the current subclass, the problem is harder — type sub-languages usually do not have late binding capabilities.

Recall the `ALLOC` layer from our data structure example. `ALLOC` is the root of the inheritance hierarchy for all compositions of mixin layers in Section 4. One of the compositions we examined is replicated here:

```
typedef SIZEOF < TIMESTAMP < BINTREE < ALLOC < int > > > >
 Tree1;                                                          (4)
```

The `node_alloc` method in the `Container` nested class of `ALLOC` is responsible for allocating storage for a data structure element. One would think that the implementation of this method would be as simple as:

```
{ return new Node; }
```

Unfortunately, this is not true. The actual allocated object should not be of type `Node`, as defined in the `ALLOC` layer (that is, `ALLOC<int>::Node` in (4)). Instead it should be of class `Node` as defined in the most refined layer (i.e., the final subclass in the hierarchy — `Tree1::Node` in (4)). In this way, the allocated node will have enough room for the stored data as well as fields added by every one of the mixin layers of composition `Tree1` (e.g., the `parent_link`, `left_link`, and `right_link` pointers added by `BINTREE`). We can circumvent this problem by weakening our type constraints and obtaining the necessary information at run-time through dynamic binding. In this particular example we need to set the return value of the `node_alloc` method to a universal pointer type (`void*`) and get the size of the allocated node through a C++ `virtual` call (not shown). This solution is general but inconvenient, error-prone (type information is lost), and possibly inefficient (depending on the overhead of dynamic binding). Although there appear to be no generally available alternatives, the problem has been studied extensively and it is interesting to cite some language mechanisms that address it.

A complete and elegant solution to the problem is offered by *virtual types* language mechanisms. Virtual types can be refined by subclasses in an inheritance chain and the most refined version is the one used by superclass code. In our data structure example, by declaring `Node` as a virtual type we express precisely our intention. Any references to `Node` (for instance, in "`new Node`") are taken relative to the most refined class in the inheritance chain (`Tree1::Node` in (4)).

Virtual types first appeared as *virtual class patterns* in the Beta programming language (see [21], ch.9). Recently they have been employed in a variety of programming language mechanisms implementing parameterization and layered frameworks similar to mixin layers. The work of [36], proposes an approach for genericity in Java using virtual types. We recognize the "`assumes inner`" primitive of feature-oriented programming [27] as a virtual type declaration specifier. The `forward` construct in the P++ language [31] serves exactly the same purpose, declaring that a certain type will be refined by subsequent layers in a composition. Our language extensions to Java that add support for mixin layers (currently under implementation — see Section 5.3) include virtual types.

5.3 Language Support

Mixin layers are mixins that encapsulate other mixins. Therefore, *the actual semantics of mixin layers depends directly on the class manipulation and inheritance semantics of the host language*. Mixin layers in CLOS or Smalltalk are semantically different than mixin layers in C++, but they can all be viewed as different implementation flavors of the same concept. For example, CLOS classes (and, therefore, mixins) have no default *class encapsulation* (class encapsulation can be emulated by defining an appropriate metaclass, however). This means that class *slots* (i.e., member variables) are not proprietary to the class that defines them. Thus, in an inheritance hierarchy, slots with the same name are merged. This prevents reusing a mixin in a single composition.

Keeping such differences in mind, we would like to examine the support for mixin layers provided by different languages. The language techniques used for encapsulation of mixins vary from reflection (i.e., methods that return class meta-objects) to lexical nesting of classes. In all the examined languages, supporting mixin layers seems to be as simple as supporting mixin classes.

CLOS. The original use of mixins was a CLOS idiom so it makes sense to ask how well our ideas are supported in CLOS. As we show in Appendix B, encapsulation of mixins can be expressed using methods that return mixins. Combined with the CLOS mixin functionality, this provides a flavor of mixin layers, adapted to the CLOS inheritance and class manipulation capabilities. In all, CLOS offers a very powerful extensibility platform for object systems, so it is no surprise that mixin layers are expressible in this context. We are, however, satisfied that the CLOS mixin layer idiom described in Appendix B is a natural one and directly relates the concept to CLOS mixins. The syntactic transformation machinery of Common Lisp (macros) can be used to add syntactic sugar to this implementation.

Smalltalk. Although we have not experimented with the Smalltalk language, we expect that the ideas explored in CLOS will be largely applicable. Smalltalk has been a traditional test bed for mixins, both for researchers (e.g., [9], [22], [34]) and for practitioners [24]. Like CLOS, the language has powerful reflective capabilities. These can be used to emulate encapsulated classes by methods that return classes. We believe (but have yet to verify) that this technique can be used in conjunction with existing mixin mechanisms to implement mixin layers.

C++. As we have seen, C++ offers direct support for most of the mixin layers

ideas. Nevertheless, there are interesting issues that arise in statically typed languages (like C++ and Java). Programming with C++ inheritance and templates can be cumbersome due to the lack of type-checking for templates. C++ templates are not types in the language (in the terminology of [10], they are *type operators*). Hence, their consistency is not checked until composition time. Furthermore, methods of templatized classes are themselves considered function templates. This means that, even after mixin layers are composed, not all of their methods will be type-checked. Only the methods actually referenced in the object code will be instantiated and, hence, type-checked (see [35] p.330-331). The result is an "interpretive" behavior of template programming. Type errors (including type mismatches and references to undeclared methods) can only be detected with the right template instantiations and method calls. This makes it hard to develop C++ mixin layers independently of the application that will use them.

Java. The Java language is an obvious next candidate for mixin layers. Java has no support for mixins, but this is the topic of active research [1][11]. The work of [11] presented a semantics for mixins in Java. This is particularly interesting from a theoretical standpoint as it addresses issues of mixin integration in a type-safe framework. Also, the latest additions to the language [16] 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 like any other members of a class. Thus, mixin layers will be straightforwardly supported by any extension adding mixin functionality to Java.

As we saw, mixins can be expressed in C++ using parameterized inheritance. There have been several recent proposals for adding parameterization/genericity to Java [1][25][26][36]. All of them have relatively clean semantics and address most of the problems we identified with C++ templates. Only the first [1] supports parameterized inheritance and, hence, can express mixin layers. Additionally, we are already working on our own Java language extensions to support mixins and mixin layers. In this effort we are using our JTS set of tools [6] for creating pre-compilers for domain-specific languages. The system currently supports a form of parameterized inheritance (and, therefore, mixin layers, when combined with nested classes). We are in the process of implementing language extensions that capture mixins and mixin layers explicitly. The extensions include a form of virtual types to address the problems identified in Section 5.2. Additionally, the fundamental building blocks of the JTS system itself were expressed as mixin layers, resulting in an elegant bootstrapped implementation.

It is interesting to examine the technical issues involved in supporting mixins in Java parameterization mechanisms. Two of these mechanisms [26][36] are based on a *homogeneous* model of parameterization: the same code is used for different instantiations of generics. This is not applicable in the case of parameterized inheritance — the superclass needs to actually change (see [1] for more details). Additionally, there may be conceptual difficulties in adding parameterized inheritance capabilities: The parameterization approach of [36] is based on virtual types. Parameterized inheritance can be approximated with virtual types by employing *virtual superclasses* [20], but this is not part of the design of [36]. The conceptual problems in the case of Pizza [26] are different. Pizza employs *type inference* (a characteristic of Hindley/Milner [23] type sys-

tems): it infers the type of parameters directly from the code (instead of requiring that the programmer declare the type explicitly and then checking that the type is indeed valid). There seem to be difficulties in combining parameterized inheritance with type inference. The difficulties are not insurmountable but an implementation may be quite complex and may require significant changes to the existing Pizza semantics.

The approaches of Myers et al. [25] and Agesen et al. [1] are conceptually similar from a language design standpoint. Even though parameterized implementations are not types in the language, their parameters can be explicitly constrained. Instantiation is explicit with constraint checking but no type inference involved. This makes these techniques easily amenable to adding parameterized inheritance capabilities, as was demonstrated in [1].

5.4 Other Related Work

There is a wealth of related research in the area of adding new responsibilities to objects — we will selectively mention a few examples. Subject-oriented programming [13] supports defining new roles for multiple objects dynamically. Aspect-oriented programming [19] also emphasizes changing the semantics of multiple objects, although in a more abstract way (aspects may encapsulate changes in the semantics of components or in implementation policies). Mezini's approach [22] emphasizes dynamic (single-)object evolution with mixin components. The context relations of [30] concentrate on dynamic modifications to a group of classes. Feature-oriented programming [27] focuses on role (or *feature*) interaction, through explicit entities (called *lifters*) that determine how two features interact. The nested mixin-methods of Agora [34] offer a powerful mechanism to control the addition of new roles.

None of the above has been associated with object-oriented design techniques and all of them require special-purpose programming language support. Nevertheless, it will be interesting to examine the main elements of these approaches (e.g., dynamic extensions, support for role interaction, and extension control through member mixins) in the context of mixin layers. We expect this to be part of our future work.

6 Conclusions

Software design methodologies that identify reusable building blocks of application construction are important. Collaboration-based design is one such methodology. It asserts that an application building block, called a *collaboration*, is neither a whole object nor a part of it, but rather cooperating parts of many different objects (called *roles*). A collaboration encapsulates one aspect of an application that is largely independent of other aspects. When this modularity is preserved in an implementation, we end up with components suitable for application synthesis through modular composition. There is a substantial track record in building applications through modular composition in this manner. This approach corresponds to layered implementation paradigms (see [2], [3], [31]). Nevertheless, the connection between this work and object-oriented design and implementation techniques has not been recognized in the above cited references.

The contribution of our work is in bridging the gap between layered design ideas and their implementations in object-oriented languages. The main point of this paper is

that encapsulation of mixins within mixins is a central concept in scalable implementations of layered designs. We named this concept “mixin layers” and showed how it can be expressed in a variety of ways using object-oriented language constructs. We believe this work is important: it directly maps a design methodology to an implementation methodology. It emphasizes object-oriented programming at the level of multiple-object components in a novel way. It shows a direction of programming language research that holds promise for realistic component-based application development.

Our work has immediate consequences: a form of mixin layers is already well supported by widely-used programming languages and can concisely express collaboration-based designs. In particular, using mixin layers in C++, we can eliminate the scalability problems of the VanHilst and Notkin implementation method. The result is a practical layered implementation approach in C++.

There remain several issues to be explored. One of the primary concerns of layered designs is that of compositional correctness: given a composition we need to ensure that it is consistent. We indicated how compositional constraints could be captured within the mixin layer framework. We also indicated the limitations of such an approach (w.r.t. unintelligible error messages). Work on composition validation raises interesting questions on how type systems can accommodate such checking. We expect this to remain a fruitful area of research for some time to come.

Appendix A — Problems with the VanHilst/Notkin Model

The length of a parameterization expression in the VanHilst/Notkin method depends on the actual roles composed and their interdependencies. In the worst case, as we show in [33], it is equal to m^n for n collaborations, each with m roles. Usually, however, not all roles need to be parameterized by all classes. Instead of devising an example, we show the one presented by VanHilst and Notkin in Figure 6 of [37]. The parameterization expression for a design with seven collaborations and three classes is:

```
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>        {};
```

Note the introduction of many intermediate types that encode common sub-expressions (so as to avoid making the expression even lengthier). Considering that we have

encountered designs with several collaborations and more than 30 roles in some of them, this approach is clearly unrealistic.

In contrast, the same composition using mixin layers was as simple as:

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

The actual application classes are nested inside `NumberC` and `CycleC`.

Appendix B — Mixin Layers in CLOS

Mixin layers can be easily expressed in CLOS by combining CLOS mixins with its powerful reflective capabilities. Keep in mind that the semantics of every incarnation of mixin layers depends on the semantics of the host language. Thus, although CLOS mixin layers are not semantically equivalent to C++ mixin layers (e.g., there is no default class encapsulation), they are just a different flavor of the same idea.

The main mixin layer template, illustrated in code fragment (2), is written as:

```
(defclass first-dummy() (...))   ; Definition of 1st inner mixin
(defclass second-dummy () (...)) ; Definition of 2nd inner mixin
(defclass third-dummy () (...))  ; Definition of 3rd inner mixin
...
(defclass collab-this () ())
(defmethod first-role ((self collab-this))
  (cons (find-class 'first-dummy) (call-next-method)))
                                 ; Encapsulate class as method
(defmethod second-role ((self collab-this))
  (cons (find-class 'second-dummy) (call-next-method)))
(defmethod third-role ((self collab-this))
  (cons (find-class 'third-dummy) (call-next-method)))        (5)
```

Note that, just like in the C++ example, the root of the outer inheritance hierarchy must be concrete (i.e., not parameterized). In the above, this means that its role-defining methods should not use `call-next-method`. Composition of mixin layers is a simple matter of using CLOS multiple inheritance (same as with regular mixins). For instance, if we have mixin layers `first-collab`, `second-collab`, `third-collab`, their composition is defined as:

```
(defclass composition
  (first-collab second-collab third-collab) (...))
(setq composite-obj (make-instance 'composition))
```

Note that role-defining methods (like `first-role`, `second-role`, etc. in (5)) return a list of all inner mixins. Constructing the inner classes is then a simple matter of creating classes programmatically using this list. This is a standard CLOS technique (e.g., see function `find-programmatic-class` in [18], p.68). For instance, creating the first of the inner classes could be expressed as:

```
(setq first-inner-class (find-programmatic-class
                          (first-role composite-obj)))
```

The above idiom should be taken as a proof-of-concept, rather than an optimal implementation of mixin layers in CLOS.

References

[1] O. Agesen, S. Freund, and J. Mitchell, “Adding Type Parameterization to the Java Language”, *OOPSLA 1997*, 49-65.

[2] D. Batory and S. O’Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, *ACM TOSEM*, October 1992.

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

[4] D. Batory and J. Thomas, “P2: A Lightweight DBMS Generator”, *Journal of Intelligent Information Systems*, 9, 107-123 (1997).

[5] D. Batory and B.J. Geraci, “Component Validation and Subjectivity in GenVoca Generators”, *IEEE Transactions on Software Engineering*, February 1997, 67-82.

[6] D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages”, to appear at the *5th International Conference on Software Reuse (ICSR ‘98)*. See `ftp://ftp.cs.utexas.edu/pub/predator/jts.ps`.

[7] K. Beck and W. Cunningham, “A Laboratory for Teaching Object-Oriented Thinking”, *OOPSLA 1989*, 1-6.

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

[9] G. Bracha and D. Griswold, “Extending Smalltalk with Mixins”, *Workshop on Extending Smalltalk* at OOPSLA 96. See `http://java.sun.com/people/gbracha/mwp.html`.

[10] L. Cardelli and P.Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys*, 17(4): Dec 1985, 471-522.

[11] M. Flatt, S. Krishnamurthi, M. Felleisen, “Classes and Mixins”. ACM *Symposium on Principles of Programming Languages*, 1998 (PoPL 98).

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

[13] W. Harrison and H. Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)”. *OOPSLA 1993*, 411-428.

[14] R. Helm, I. Holland, and D. Gangopadhyay, “Contracts: Specifying Behavioral Compositions in Object-Oriented Systems”. *OOPSLA 1990*, 169-180.

[15] I. Holland, “Specifying Reusable Components Using Contracts”, *ECOOP 92*, 287-308.

[16] JavaSoft, “Inner Classes Specification”, from `http://java.sun.com/products/jdk/1.1/docs/`.

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

[18] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*. MIT Press, 1991.

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

[20] O. L. Madsen and B. Møller-Pedersen, “Virtual classes: A powerful mechanism in object-oriented programming”, *OOPSLA 1989*, 397-406.

[21] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard, *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[22] M. Mezini, “Dynamic Object Evolution without Name Collisions”, *ECOOP 97*, 190-219.

[23] R. Milner, “A Theory of Type Polymorphism in Programming”, *Journal of Computer and System Sciences*, 17:Dec 1978, 348-375.

[24] T. Montlick, “Implementing Mixins in Smalltalk”, *The Smalltalk Report*, July 1996.

[25] A. Myers, J. Bank and B. Liskov, “Parameterized Types for Java”, ACM *Symposium on Principles of Programming Languages*, 1997 (PoPL 97).

[26] M. Odersky and P. Wadler, “Pizza into Java: Translating theory into practice”, ACM *Symposium on Principles of Programming Languages*, 1997 (PoPL 97).

[27] C. Prehofer, “Feature-Oriented Programming: A Fresh Look at Objects”, *ECOOP* 97, 419-443.

[28] T. Reenskaug, E. Anderson, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet, “OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems”, *Journal of Object-Oriented Programming*, 5(6): October 1992, 27-41.

[29] J. Rumbaugh, “Getting Started: Using use cases to capture requirements”, *Journal of Object-Oriented Programming*, 7(5): Sep 1994, 8-23.

[30] L. Seiter, J. Palsberg, and K. Lieberherr, “Evolution of Object Behavior using Context Relations”, *ACM SIGSOFT* 1996.

[31] V. Singhal, “A Programming Language for Writing Domain-Specific Software System Generators”, Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, August 1996.

[32] Y. Smaragdakis and D. Batory, “DiSTiL: a Transformation Library for Data Structures”, *USENIX Conference on Domain-Specific Languages (DSL 97)*.

[33] Y. Smaragdakis and D. Batory, “Implementing Reusable Object-Oriented Components”, to appear at the *5th International Conference on Software Reuse (ICSR ‘98)*. See `ftp://ftp.cs.utexas.edu/pub/predator/icsrtemp.ps`.

[34] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, and M. Van Limberghen, “Nested Mixin-Methods in Agora”, *ECOOP 93*, 197-219.

[35] B. Stroustrup, *The C++ Programming Language, 3rd Ed.*, Addison-Wesley, 1997.

[36] K. Thorup, “Genericity in Java with Virtual Types”, *ECOOP 97*, 444-471.

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

[38] M. VanHilst and D. Notkin, “Using Role Components to Implement Collaboration-Based Designs”. *OOPSLA 1996*.

[39] M. VanHilst and D. Notkin, “Decoupling Change From Design”, *ACM SIGSOFT* 1996.

[40] M. VanHilst, “Role-Oriented Programming for Software Evolution”, Ph.D. Dissertation, University of Washington, Computer Science and Engineering, 1997.