Mixin-Based Programming in C++[1]

Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
yannis@cc.gatech.edu

Don Batory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
batory@cs.utexas.edu

Abstract. Combinations of C++ features, like inheritance, templates, and class nesting, allow for the expression of powerful component patterns. In particular, research has demonstrated that, using C++ *mixin classes*, one can express layered component-based designs concisely with efficient implementations. In this paper, we discuss pragmatic issues related to component-based programming using C++ mixins. We explain surprising interactions of C++ features and policies that sometimes complicate mixin implementations, while other times enable additional functionality without extra effort.

1 Introduction

Large software artifacts are arguably among the most complex products of human intellect. The complexity of software has led to implementation methodologies that divide a problem into manageable parts and compose the parts to form the final product. Several research efforts have argued that C++ templates (a powerful parameterization mechanism) can be used to perform this division elegantly.

In particular, the work of VanHilst and Notkin [29][30][31] showed how one can implement *collaboration-based* (or *role-based*) designs using a certain templatized class pattern, known as a *mixin class* (or just *mixin*). Compared to other techniques (e.g., a straightforward use of *application frameworks* [17]) the VanHilst and Notkin method yields less redundancy and reusable components that reflect the structure of the design. At the same time, unnecessary dynamic binding can be eliminated, resulting into more efficient implementations. Unfortunately, this method resulted in very complex parameterizations, causing its inventors to question its scalability.

The *mixin layers* technique was invented to address these concerns. Mixin layers are mixin classes nested in a pattern such that the parameter (superclass) of the outer mixin determines the parameters (superclasses) of inner mixins. In previous work [4][24][25], we showed how mixin layers solve the scalability problems of the VanHilst and Notkin method and result into elegant implementations of collaboration-based designs.

---

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.

This paper discusses practical issues related to mixin-based programming. We adopt a viewpoint oriented towards C++ implementations, but our discussion is not geared towards the C++ expert. Instead, we aim to document common problems and solutions in C++ mixin writing for the casual programmer. Additionally, we highlight issues that pertain to language design in general (e.g., to Java parameterization or to the design of future languages). Most of the issues clearly arise from the interaction of C++ features with the constructs under study. The discussion mainly stems from actual experience with C++ mixin-based implementations but a few points are a result of close examination of the C++ standard, since they refer to features that no compiler we have encountered implements. Even though we present an introduction to mixins, mixin layers, and their uses, the primary purpose of this paper is *not* to convince readers of the value of these constructs. (The reader should consult [4], [24], [25], [26], or [29] for that.)

We believe that the information presented here represents a valuable step towards moving some powerful programming techniques into the mainstream. We found that the mixin programming style is quite practical, as long as one is aware of the possible interactions with C++ idiosyncrasies. As C++ compilers move closer to sophisticated template support (e.g., some compilers already support separate template compilation) the utility of such techniques will increase rapidly.

2 Background (Mixins and Mixin Layers)

The term *mixin class* (or just *mixin*) has been overloaded in several occasions. Mixins were originally explored in the Lisp language with object systems like Flavors [20] and CLOS [18]. In these systems, mixins are an idiom for specifying a class and allowing its 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 [28], p.402). (This is ***not*** the meaning that we will use in this paper.) Both of these mechanisms are approximations of a general concept described by Bracha and Cook [6]. The idea is simple: we would like to specify an extension without pre-determining what exactly it can extend. This is equivalent to specifying a subclass while leaving its superclass as a parameter to be determined later. The benefit is that a single class can be used to express an incremental extension, valid for a variety of classes.

Mixins can be implemented using parameterized inheritance. The superclass of a class is left as a parameter to be specified at instantiation time. In C++ we can write this as:

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

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 [33]). The mixin could have the form:

```
template <class Graph>
class Counting: public Graph {
  int nodes_visited, edges_visited;
public:
  Counting() : nodes_visited(0), edges_visited(0), Graph() { }
  node succ_node (node v) {
    nodes_visited++;
    return Graph::succ_node(v);
  }
  edge succ_edge (edge e) {
    edges_visited++;
    return Graph::succ_edge(e);
  }
  ...
};
```

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). We can have, for instance, two different compositions:

```
  Counting< Ugraph > counted_ugraph;
```

and

```
  Counting< Dgraph > counted_dgraph;
```

for undirected and directed graphs. (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.

VanHilst and Notkin demonstrated that mixins are beneficial for a general class of object-oriented designs [29]. They used a mixin-based approach to implement *collaboration-based* (a.k.a. *role-based*) designs [5][15][16][21][29]. These designs are based on the view that objects are composed of different roles that they play in their interaction with other objects. The fundamental unit of functionality is a protocol for this interaction, called a *collaboration*. The mixin-based approach of VanHilst and Notkin results in efficient implementations of role-based designs with no redundancy. Sometimes, however, the resulting parameterization code is quite complicated—many mixins need to be composed with others in a complex fashion. This introduces scalability problems (namely, extensions that instantiate template parameters can be of length exponential to the number of mixins composed—see [24]). To make the approach more practical by reducing its complexity, *mixin layers* were introduced. Because mixin layers are an incremental improvement of the VanHilst and Notkin method, we only discuss implementing collaboration-based designs using mixin layers.

Mixin layers [24][25][26] are a particular form of mixins. They are designed with the purpose of encapsulating refinements for multiple classes. Mixin layers are nested

mixins such that the parameter of an outer mixin determines the parameters of inner mixins. The general form of a mixin layer in C++ is:

```
template <class NextLayer>
class ThisLayer : public NextLayer {
public:
   class Mixin1 : public NextLayer::Mixin1 { ... };
   class Mixin2 : public NextLayer::Mixin2 { ... };
   ...
};
```

Mixin layers are a result of the observation that a conceptual unit of functionality is usually neither one object nor parts of an object—a unit of functionality may span several different objects and specify refinements (extensions) to all of them. All such refinements can be encapsulated in a single mixin layer and the standard inheritance mechanism can be used for composing extensions.

This property of mixin layers makes them particularly attractive for implementing collaboration-based designs. Each layer captures a single collaboration. Roles for all classes participating in a collaboration are represented by inner classes of the layer. Inheritance works at two different levels. First, a layer can inherit entire classes from its superclass (i.e., the parameter of the layer). Second, inner classes inherit members (variables, methods, or even other classes) from the corresponding inner classes in the superclass layer. This dual application of inheritance simplifies the implementation of collaboration-based designs, while preserving the benefits of the VanHilst and Notkin method. An important source of simplifications is that inner classes of a mixin layer can refer unambiguously to other inner classes—the layer acts as a namespace.

We illustrate our point with an example (presented in detail in [24]) of a collaboration-based design and its mixin layers implementation. (Full source code is available, upon request.) This example presents a graph traversal application and was examined initially by Holland [16] and subsequently by VanHilst and Notkin [29]. This application defines three different algorithms on an undirected graph, all implemented using a 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. The application 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 the *VertexNumbering* operation, for instance, a *Workspace* object holds the value of the last number assigned to a vertex as well as the methods to update this number.

As shown in Fig 1, we can decompose this application into five independent collaborations—one encompassing the functionality of an undirected graph, another encoding depth-first traversals, and three containing the specifics of each graph algorithm (vertex numbering, cycle checking, and connected regions). Note that each collaboration

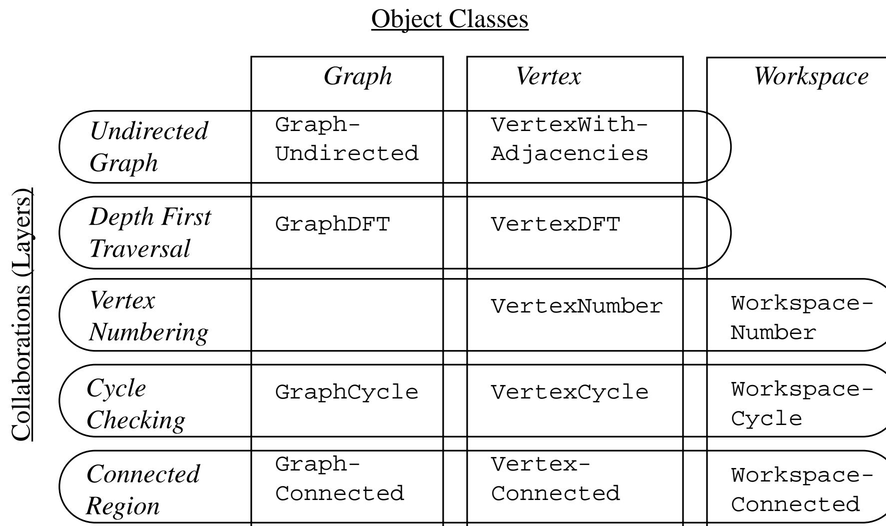

Fig 1: Collaboration decomposition of the example application: A depth-first traversal of an undirected graph is specialized to yield three different graph operations. Ovals represent collaborations, rectangles represent classes, their intersections represent roles.

captures a distinct aspect of the application and each object may participate in several aspects. That is to say, each object may play several roles. 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 actual depth-first traversal algorithm.

By implementing collaborations as mixin layers, the modular design of Fig 1 can be maintained at the implementation level. For instance, the "*Vertex Numbering*" collaboration can be implemented using a layer of the general form:

```
template <class Next>
class NUMBER : public Next {
public:
  class Workspace : public Next::Workspace {
    ... // Workspace role members
  };
  class Vertex : public Next::Vertex {
    ... // Vertex role members
  };
};
```

Note that no role (nested class) is prescribed for `Graph`. A `Graph` class is inherited from the superclass of `Number` (the class denoted by parameter `Next`).

As shown in [24], such components are flexible and can be reused and interchanged. For instance, the following composition builds `Graph`, `Vertex`, and `WorkSpace`

classes nested inside class `CycleC` that implement vertex numbering of undirected graphs using a depth-first traversal:[2]

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

By replacing `NUMBER` with other mixin layers we get the other two graph algorithms discussed. Many more combinations are possible. We can use the templates to create classes that implement more than one algorithm. For instance, we can have an application supporting both vertex numbering *and* cycle checking on the same graph by refining two depth-first traversals in order:

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

Furthermore, all the characteristics of an undirected graph are captured by the `UGRAPH` mixin layer. Hence, it is straightforward to apply the same algorithms to a directed graph (mixin layer `DGRAPH` interchanged for `UGRAPH`):[3]

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

This technique (of composing source components in a large number of combinations) underlies the *scalable libraries* [3] design approach for source code plug-and-play components.

3 Programming with C++ Mixins: Pragmatic Considerations

Since little has been written about the pragmatics of doing component programming using C++ mixins (mixin classes or mixin layers), we feel it is necessary to discuss some pertinent issues. Most of the points raised below concern fine interactions between the mixin approach and C++ idiosyncrasies. Others are implementation suggestions. They are all useful knowledge before one embarks into a development effort using C++ mixins and could serve to guide design choices for future parameterization mechanisms in programming languages. The C++ aspects we discuss are well-documented and other C++ programmers have probably also made some of our observations. Nevertheless, we believe that most of them are non-obvious and many only arise in the context of component programming—that is, when a mixin is designed and used in complete isolation from other components of the system.

Lack of template type-checking. Templates do not correspond to types in the C++ language. Thus, they are not type-checked until instantiation time (that is, composition time for mixins). Furthermore, methods of templatized classes are themselves consid-

---

2. The `DEFAULTW` mixin layer is an implementation detail, borrowed from the VanHilst and Notkin implementation [29]. It contains an empty `WorkSpace` class and its purpose is to avoid dynamic binding by changing the order of composition.
3. This is under the assumption that the algorithms are still valid for directed graphs as is the case with the original code for this example [16].

ered function templates.[4] Function templates in C++ are instantiated automatically and only when needed. Thus, even after mixins are composed, not all their methods will be type-checked (code will only be produced for methods actually referenced in the object code). This means that certain errors (including type mismatches and references to undeclared methods) can only be detected with the right template instantiations and method calls. Consider the following example:

```
template <class Super>
class ErrorMixin : public Super {
public:
  ...
  void sort(FOO foo) {
    Super::srot(foo);   // misspelled
  }
};
```

If client code never calls method `sort`, the compiler will *not* catch the misspelled identifier above. This is true even if the `ErrorMixin` template is used to create classes, and methods other than `sort` are invoked on objects of those classes.

Delaying the instantiation of methods in template classes can be used to advantage, as we will see later. Nevertheless, many common designs are such that all member methods of a template class should be valid for all instantiations. It is not straightforward to enforce the latter part ("for *all* instantiations") but for most practical purposes checking all methods for a single instantiation is enough. This can be done by explicit instantiation of the template class, which forces the instantiation of all its members. The idiom for explicit instantiation applied to our above example is:

```
template class ErrorMixin<SomeFoo>;
```

When "subtype of" does not mean "substitutable for". There are two instances where inheritance may not behave the way one might expect in C++. First, constructor methods are not inherited. Ellis and Stroustrup ([13], p.264) present valid reasons for this design choice: the constructor of a superclass does not suffice for initializing data members added by a subclass. Often, however, a mixin class may be used only to enrich or adapt the method interface of its superclasses *without* adding data members. In this case it would be quite reasonable to inherit a constructor, which, unfortunately, is not possible. The practical consequence of this policy is that the only constructors that are visible in the result of a mixin composition are the ones present in the outermost mixin (bottom-most class in the resulting inheritance hierarchy). To make matters worse, constructor initialization lists (e.g.,

```
  constr() : init1(1,2), init2(3) {}  )
```

can only be used to initialize direct parent classes. In other words, all classes need to know the interface for the constructor of their direct superclass (if they are to use con-

4. This wording, although used by the father of C++—see [28], p.330—is not absolutely accurate since there is no automatic type inference.

structor initialization lists). Recall, however, that a desirable property for mixins is that they be able to act as components: a mixin should be implementable in isolation from other parts of the system in which it is used. Thus a single mixin class should be usable with several distinct superclasses and should have as few dependencies as possible.

A possible workaround for this problem is to use a standardized construction interface. A way to do this is by creating a construction class encoding the union of all possible arguments to constructors in a hierarchy. Then a mixin "knows" little about its direct superclass, but has dependencies on the union of the construction interfaces for all its possible parent classes. (Of course, another workaround is to circumvent constructors altogether by having separate initialization methods. This, however, requires a disciplined coding style to ensure that methods are always called after object construction.) As a side-note, destructors for base classes are called automatically so they should not be replicated.

The second instance where subtypes are not substitutable in C++ occurs with top-level function templates. Assume a function template of the form:

```
template <class Next> void weird_function ( Mixin<Next> arg )
{ ... }
```

This function template will be instantiated correctly when called with an argument of type `Mixin<Base>`, but not when called with an argument of type `NewMixin<Mixin<Base>>`. Even though the latter type is a subtype of the former, subtyping is not involved in the function template instantiation policy of C++. The problem is solved only by ensuring that the template gets instantiated with an argument of type `Mixin<Base>` (e.g., there is an explicit call to `weird_function` with an argument of this type). Once this is done, the function generated by the template can be invoked with actual arguments that are subtypes of the corresponding formal argument types. That is, the following example (using `weird_function`, above) is valid code:

```
Base base;
Mixin<Base> derived;
weird_function(base);
  // Error if above line omitted!
weird_function(derived);
```

But if the third line is omitted, the example becomes invalid because the call `weird_function(derived)` is now illegal—a most counterintuitive result.

One may question whether the problem arises in a practical setting. Indeed, we encountered the problem when attempting to build a set of mixin components based on data structure classes from the Standard Template Library (STL) [27]. The equality operator for STL data structures (`operator==` in C++) is itself a function template of the same form as `weird_function`, above. For instance, the equality operator for linked lists is defined as:

```
template <class T, class Alloc>
inline bool operator== (const list<T,Alloc>& x,
                        const list<T,Alloc>& y)
{ ... }
```

This means that defining mixins (or other subclasses) of STL data structure classes requires redefining `operator==` for each new class. Again, this is not very desirable, given that we want mixins to act as components with low overhead. Mixins should know very little about their superclasses and functionality defined for the superclass should be transparently inherited. In practice, several mixin components are just thin wrappers adapting their superclass's interface. Having to redefine constructors and top-level function templates may be overly tedious in this case.

Synonyms for compositions. In the past sections we have used `typedefs` to introduce synonyms for complicated mixin compositions—e.g.,

```
  typedef A < B < C > > Synonym;
```

Another reasonable approach would be to introduce an empty subclass:

```
class Synonym : public A < B < C > > { };
```

The first form has the advantage of preserving constructors of component `A` in the synonym. The second idiom is cleanly integrated into the language (e.g., can be templatized, compilers create short link names for the synonym, etc.). Additionally, it can solve a common problem with C++ template-based programming: generated names (template instantiations) can be extremely long, causing compiler messages to be incomprehensible.

Designating virtual methods. Sometimes C++ policies have pleasant side-effects when used in conjunction with mixins. An interesting case is that of a mixin used to create classes where a certain method can be virtual or not, depending on the concrete class used to instantiate the mixin. This is due to the C++ policy of letting a superclass declare whether a method is virtual, while the subclass does not need to specify this explicitly. Consider a regular mixin and two concrete classes instantiating it (a C++ `struct` is a class whose members are public by default):

```
template <class Super>
struct MixinA : public Super {
  void virtual_or_not(FOO foo) { ... }
};

struct Base1 {
  virtual void virtual_or_not(FOO foo) { ... }
  ... // methods using "virtual_or_not"
};

struct Base2 {
  void virtual_or_not(FOO foo) { ... }
};
```

The composition `MixinA<Base1>` designates a class in which the method `virtual_or_not` is virtual. Conversely, the same method is not virtual in the composition `MixinA<Base2>`. Hence, calls to `virtual_or_not` in `Base1` will call the method supplied by the mixin in the former case but not in the latter.

In the general case, this phenomenon allows for interesting mixin configurations. Classes at an intermediate layer may specify methods and let the inner-most layer decide whether they are virtual or not.

As we recently found out, this technique was described first in [12].

Single mixin for multiple uses. The lack of template type-checking in C++ can actually be beneficial in some cases. Consider two classes `Base1` and `Base2` with very similar interfaces (except for a few methods):

```
struct Base1 {
  void regular() { ... }
  ...
};
struct Base2 {
  void weird() { ... }
  ... // otherwise same interface as Base1
};
```

Because of the similarities between `Base1` and `Base2`, it makes sense to use a single mixin to adapt both. Such a mixin may need to have methods calling either of the methods specific to one of the two base classes. This is perfectly feasible. A mixin can be specified so that it calls either `regular` or `weird`:

```
template <class Super>
class Mixin : public Super {
  ...
public:
  void meth1() { Super::regular(); }
  void meth2() { Super::weird(); }
};
```

This is a correct definition and it will do the right thing for both composition `Mixin<Base1>` and `Mixin<Base2>`! What is remarkable is that part of `Mixin` seems invalid (calls an undefined method), no matter which composition we decide to perform. But, since methods of class templates are treated as function templates, no error will be signalled unless the program actually uses the wrong method (which may be `meth1` or `meth2` depending on the composition). That is, an error will be signalled only if the program is indeed wrong. We have used this technique to provide uniform, componentized extensions to data structures supporting slightly different interfaces (in particular, the red-black tree and hash table of the SGI implementation of the Standard Template Library [22]).

Propagating type information. An interesting practical technique (also applicable to languages other than C++) can be used to propagate type information from a subclass to a superclass, when both are created from instantiating mixins. This is a common problem in object-oriented programming. It was, for instance, identified in the design of the P++ language [23] (an extension of C++ with constructs for component-based programming) and solved with the addition of the `forward` keyword. The same problem is addressed in other programming languages (e.g., Beta [19]) with the concept of *virtual types*.

Consider a mixin layer encapsulating the functionality of an allocator. This component needs to have type information propagated to it from its subclasses (literally, the subclasses of the class it will create when instantiated) so that it knows what kind of data to allocate. (We also discussed this example in detail in [25] but we believe that the solution presented here is the most practical way to address the problem.) The reason this propagation is necessary is that subclasses may need to add data members to a class used by the allocator. One can solve the problem by adding an extra parameter to the mixin that will be instantiated with the final product of the composition itself. In essence, we are reducing a conceptual cycle in the parameterization to a single self-reference (which is well-supported in C++). This is shown in the following code fragment:

```
template <class EleType, class FINAL>
class ALLOC {
public:
  class Node {
    EleType element;  // stored data type
  public:
    ...  // methods using stored data
  };

  class Container {
  protected:
    FINAL::Node* node_alloc() {
      return new FINAL::Node();
    }
    ... // Other allocation methods
  };
};

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

```
  class Container : public Super::Container {
    Node* header; // Container data members
  public:
    ...           // Interface methods
  };
};

class Comp : public BINTREE < ALLOC <int, Comp> > {/* empty */};
```

Note what is happening in this code fragment (which is abbreviated but preserves the structure of actual code that we have used). A binary tree data structure is created by composing a `BINTREE` mixin layer with an `ALLOC` mixin layer. The data structure stores integer (`int`) elements. Nevertheless, the actual type of the element stored is *not* `int` but a type describing the node of a binary tree (i.e., an integer together with three pointers for the parent, and the two children of the node). This is the type of element that the allocator should reserve memory for.

The problem is solved by passing the final product of the composition as a parameter to the allocator mixin. This is done through the self-referential (or *recursive*) declaration of class `Comp`. (Theoretically-inclined readers will recognize this as a *fixpoint* construction.) Note that `Comp` is just a synonym for the composition and it has to use the synonym pattern introducing a class (i.e., the `typedef` synonym idiom discussed earlier would not work as it does not support recursion).

It should be noted that the above recursive construction has been often used in the literature. In the C++ world, the technique was introduced by Barton and Nackman [2] and popularized by Coplien [9]. Nevertheless, the technique is not mixin-specific or even C++-specific. For instance, it was used by Wadler, Odersky and the first author [32] in Generic Java [7] (an extension of Java with parametric polymorphism). The origins of the technique reach back at least to the development of F-bounded polymorphism [8].

Hygienic templates in the C++ standard. The C++ standard ([1], section 14.6) imposes several rules for name resolution of identifiers that occur inside templates. The extent to which current compilers implement these rules varies, but full conformance is the best approach to future compatibility for user code.

Although the exact rules are complicated, one can summarize them (at loss of some detail) as "templates cannot contain code that refers to 'nonlocal' variables or methods". Intuitively, "nonlocal" denotes variables or methods that do not depend on a template parameter and are not in scope at the global point closest to the template definition. This rule prevents template instantiations from capturing arbitrary names from their instantiation context, which could lead to behavior not predicted by the template author.

A specific rule applies to mixin-based programming. To quote the C++ standard (14.6.2), "if a base class is a dependent type, a member of that class cannot hide a name declared within a template, or a name from the templates enclosing scopes".

Consider the example of a mixin calling a method defined in its parameter (i.e., the superclass of the class it will create when instantiated):

```
struct Base {
  void foo() { ... }
};

void foo() { }

template <class Super>
struct Mixin : public Super {
  void which_one()   { foo(); } // ::foo
};

Mixin < Base > test;
```

That is, the call to `foo` from method `which_one` will refer to the global `foo`, not the `foo` method of the `Base` superclass.

The main implication of these name resolution rules is on the way template-based programs should be developed. In particular, imagine changing a *correct* class definition into a mixin definition (by turning the superclass into a template parameter). Even if the mixin is instantiated with its superclass in the original code, the new program is *not* guaranteed to work identically to the original, because symbols may now be resolved differently. This may surprise programmers who work by creating concrete classes and turning them into templates when the need for abstraction arises. To avoid the potential for insidious bugs, it is a good practice to explicitly qualify references to superclass methods (e.g., `Super::foo` instead of just `foo`).

4 Related Work

Various pieces of related work have been presented in the previous sections. We cannot exhaustively reference all C++ template-based programming techniques, but we will discuss two approaches that are distinct from ours but seem to follow parallel courses.

The most prominent example is the various implementations of the STL. Such implementations often exercise the limits of template support and reveal interactions of C++ policies with template-based programming. Nevertheless, parameterized inheritance is not a part of STL implementations. Hence, the observations of this paper are mostly distinct from the conclusions drawn from STL implementation efforts.

Czarnecki and Eisenecker's generative programming techniques [10][11] were used in the Generative Matrix Computation Library (GMCL). Their approach is a representative of techniques using C++ templates as a programming language (that is, to perform arbitrary computation at template instantiation time). What sets their method apart from other template meta-programming techniques is that it has similar goals to mixin-based programming. In particular, Czarnecki and Eisenecker try to develop components which can be composed in multiple ways to yield a variety of implementations.

Several of the remarks in this paper are applicable to their method, even though their use of mixins is different (for instance, they do not use mixin layers).

5 Conclusions

We presented some pragmatic issues pertaining to mixin-based programming in C++. We believe that mixin-based techniques are valuable and will become much more widespread in the future. Mixin-based programming promises to provide reusable software components that result into flexible and efficient implementations.

Previous papers have argued for the value of mixin-based software components and their advantages compared to application frameworks. In this paper we tried to make explicit the engineering considerations specific to mixin-based programming in C++. Our purpose is to inform programmers of the issues involved in order to help move mixin-based programming into the mainstream.

References

[1] ANSI / ISO Standard: *Programming Languages—C++*, ISO/IEC 14882, 1998.

[2] J. Barton and L.R. Nackman, *Scientific and Engineering C++: An Introduction with Advanced Techniques and Applications*, Addison-Wesley, 1994.

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

[4] D. Batory, R. Cardone, and Y. Smaragdakis, “Object-Oriented Frameworks and Product-Lines”, *1st Software Product-Line Conference*, Denver, Colorado, August 1999.

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

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

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

[8] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell, “F-bounded Polymorphism for Object-Oriented Programming”, in *Proc. Conf. on Functional Programming Languages and Computer Architecture*, 1989, 273-280.

[9] J. Coplien, “Curiously Recurring Template Patterns”, *C++ Report*, 7(2):24-27, Feb. 1995.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[11] K. Czarnecki and U. Eisenecker, “Synthesizing Objects”, *ECOOP 1999*, 18-42.

[12] U. Eisenecker, “Generative Programming in C++”, in *Proc. Joint Modular Languages Conference (JMLC’97)*, LNCS 1204, Springer, 1997, 351-365.

[13] M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

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

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

[16] I. Holland, “Specifying Reusable Components Using Contracts”, *ECOOP 1992*, 287-308.

[17] R. Johnson and B. Foote, “Designing Reusable Classes”, *J. 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] O.L. Madsen, B. Møller-Pedersen, and K. Nygaard, *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[20] D.A. Moon, “Object-Oriented Programming with Flavors”, *OOPSLA 1986*.

[21] 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”, *J. of Object-Oriented Programming*, 5(6): October 1992, 27-41.

[22] Silicon Graphics Computer Systems Inc., *STL Programmer’s Guide*. See: `http://www.sgi.com/Technology/STL/`.

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

[24] Y. Smaragdakis and D. Batory, “Implementing Reusable Object-Oriented Components”. In the *5th Int. Conf. on Software Reuse (ICSR 98)*.

[25] Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”. In *ECOOP 98*.

[26] Y. Smaragdakis, “Implementing Large-Scale Object-Oriented Components”, Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, December 1999.

[27] A. Stepanov and M. Lee, “The Standard Template Library”. Incorporated in ANSI/ISO Committee C++ Standard.

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

[29] 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.

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

[31] M. VanHilst and D. Notkin, “Decoupling Change From Design”, *SIGSOFT* 96.

[32] P. Wadler, M. Odersky and Y. Smaragdakis, “Do Parametric Types Beat Virtual Types?”, unpublished manuscript, posted in October 1998 in the Java Genericity mailing list (`java-genericity@cs.rice.edu`).

[33] K. Weihe, “A Software Engineering Perspective on Algorithmics”, available at `http://www.informatik.uni-konstanz.de/Preprints/`.