Base-Class Composition with Multiple Derivation and Virtual Bases

Lee R. Nackman
lrn@watson.ibm.com

John J. Barton
jjb@watson.ibm.com

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598

Abstract

For systems of C++ classes using virtual functions, writing base classes that only specify virtual member functions and then writing other base classes that only implement those functions improves extensibility. When interface is separated from implementation, both interfaces and implementations can be extended separately by derivation. New classes can then be composed by multiple derivation, combining one interface and one implementation base class. We call this form of composition *base-class composition*.

Programmers familiar with the advantages of base-class composition fail to use it because of the performance penalty of multiple derivation and virtual base classes. Consequently, compiler writers, failing to see extensive applications of multiple derivation and virtual bases, have little incentive to eliminate the performance penalty. We highlight the advantages of the base-class composition pattern and show how the performance penalty can be eliminated by compiler optimization.

1 Introduction

In general, C++ classes combine interface and implementation, specifying both the member functions that can be called (interface) and object state (implementation). Derived classes can extend their base class's interface or they can reuse their base class's implementation or both. Building on the work of Martin [1], we observe that there are many advantages to writing base classes that either specify interface or provide implementation, but not both. With a few simple rules applied consistently, these two kinds of cooperating base classes become building blocks that can be combined using multiple derivation.

We call this design strategy *base-class composition*. It allows independent extension of interface, alternative implementation of interface, reuse of implementation, encapsulation of implementation, and avoids recompilation of clients when implementation is altered. Explaining its uses and its advantages are the first goal of this paper.

We believe that programmers familiar with the advantages of this composition fail to use it because of the performance penalty of multiple derivation and virtual base classes. Consequently, compiler writers, failing to see extensive applications of multiple derivation and virtual bases, have little incentive to eliminate the performance penalty. After we highlight the advantages of the base-class composition pattern, we show how the performance penalty can be eliminated by compiler optimization.

We shall focus our attention on aiding the construction of classes to be used through virtual function calls. A function that uses a class is a client of that class; it uses some services provided by the class. Some functions use an instance of a class only through its virtual functions and have no need to know the exact type of the object being used. Such functions can be written to use pointers or references to a base class with virtual functions. Let's call such a base class an *interface base class*. For example, a function that calls virtual `Shape::draw()` on a `Shape&` uses the object through an interface base class `Shape`. The object itself could be a `Circle`, `Square`, or `Triangle`. Let's call such a function a *use client*.

Base-class composition aids the construction and maintenance of classes for use clients. Two other important clients are *creation clients*, those that create objects (requiring specific types) and *downcast clients*, those that apply derived-type specific operations to objects given only references to common base classes of the objects. Base-class composition does not make creation clients more difficult to design nor does it make downcast clients less difficult.

The next section gives an example of base-class composition. Section 3 discusses characteristics of base-class composition and Section 4 compares it to the alternatives. The performance penalties of using base-class composition are described in Section 5. We show in Section 6 that a straightforward optimization technique can eliminate the costs of multiple derivation with virtual bases under certain circumstances and then, in Section 7, we discuss a further optimization applicable to private bases used in base-class composition. Related work is discussed in Section 8.

2 An Example of Base-Class Composition

To illustrate base-class composition, we use a simple but realistic example. Suppose we want to write a tool that manipulates C++ source code and that we need a way to represent C++ language elements that appear in the source code. We might define a class for each C++ language element to be represented. In particular, let's assume that we want to define classes to represent C++'s `union`, `struct`, and `class`. These are all *aggregates* in the sense that they contain source code entities such as member functions, member data, nested classes, and nested typedefs. However, only `class` and `struct` can have base classes.

It is reasonable to expect classes for representing C++ aggregates to meet the following criteria:

- All three kinds of aggregates should provide common functions representative of aggregation.
- The `class` and `struct` aggregates should provide common functions representative of aggregates that can have bases.
- All three aggregates should be able to use a common implementation of aggregation.
- The two aggregates that can have bases should be able to use a common implementation of access to base classes.

To focus on the issues related to separation of interface and implementation, we limit our example function interfaces: all aggregates respond to `numMembers()` and aggregates that can have bases respond additionally to `numBases()`. We mean for these functions to be representative of larger commonalities and differences between aggregates and aggregates with bases.

Figure 1 shows a class DAG meeting our design goals. Two classes, `Aggregate` and `AggregateWithBases`, are interface base classes, meaning that we intend for client functions to use pointers or references to these classes and that we intend to derive from these classes to implement the interfaces they specify. These classes specify member functions common to their derived classes. For base-class composition we restrict interface base classes to be abstract base classes with neither member data nor constructors.

Characteristics of all aggregates are specified by pure virtual member functions of `Aggregate`:

```
class ostream;
class Aggregate {
public:
  virtual int numMembers() const = 0;
  virtual void kind(ostream&) const = 0; // name of kind of aggregate.
  // ...
};
```

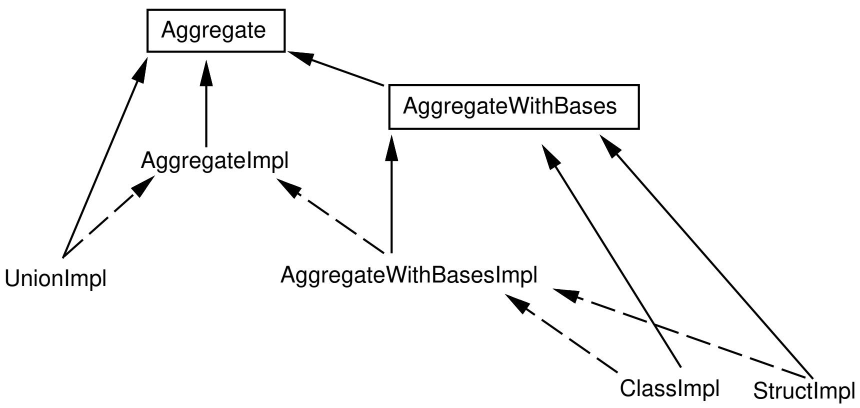

Figure 1: Class DAG for C++ aggregates using base-class composition. Boxes indicate interface base classes. Solid and dashed arrows indicate public and private derivation respectively.

Aggregates that may have bases have all the characteristics of aggregates plus characteristics related to bases. This is specified by publicly deriving `AggregateWithBases` from `Aggregate` and declaring additional pure virtual member functions:

```
class AggregateWithBases :
  public virtual Aggregate {
public:
  virtual int numBases() const = 0;
  // ...
};
```

The reason for using virtual derivation is explained below.

`AggregateImpl` implements the `numMembers()` virtual function of `Aggregate`:

```
class AggregateImpl :
  public virtual Aggregate {
public:
  AggregateImpl(int num_members) : _num_members(num_members) {}
  virtual int numMembers() const { return _num_members; }
  // ...
private:
  int _num_members;  // ...representation of members...
};
```

A `UnionImpl` would represent `union` objects by deriving from this class for code and data reuse:

```
class UnionImpl :
  public virtual Aggregate,
  private AggregateImpl {
public:
  UnionImpl(int num_members) : AggregateImpl(num_members) {}
  virtual void kind(ostream& os) const { os << "union"; }
};
```

`AggregateWithBasesImpl` reuses the implementation of the `Aggregate` layer of its `AggregateWithBases` interface by deriving privately from `AggregateImpl`. It also adds implementation for the `numBases()` virtual function specified by `AggregateWithBases`:

```
class AggregateWithBasesImpl :
  public virtual AggregateWithBases,
  private AggregateImpl {
public:
```

```
  AggregateWithBasesImpl(int num_members, int num_bases) :
    AggregateImpl(num_members),
   _num_bases(num_bases) {
  }
  virtual int numBases() const { return _num_bases; }
private:
  int _num_bases;
};
```

Using multiple derivation here enables `AggregateWithBasesImpl` to implement the `AggregateWithBases` interface on the one hand and to reuse the `AggregateImpl` implementation on the other hand.

Finally, `ClassImpl` is derived from `AggregateWithBasesImpl` using the same style:

```
class ClassImpl :
 public virtual AggregateWithBases,
 private AggregateWithBasesImpl {
public:
  ClassImpl(int num_members, int num_bases) :
     AggregateWithBasesImpl(num_members, num_bases) {
  }
  virtual void kind(ostream& os) const { os << "class"; }
  // ...
};
```

`StructImpl` would be similar.

With these classes we can build and process collections of aggregates. For example, we could construct an array of `Aggregate` pointers that point to `UnionImpl`, `ClassImpl`, and `StructImpl` objects. Then a client function that computes, say, the average number of members for each aggregate would look like this:

```
float avgNumMembers(int n_aggs, Aggregate* aggs[]) {
  int sum = 0;
  for (int i = 0; i < n_aggs; i++) sum += aggs[i]->numMembers();
  return sum / n_aggs;
}
```

Likewise, we can build and process collections of aggregates with bases by creating an array of `AggregateWithBases` pointers that point to `ClassImpl` and `StructImpl` instances. We could then write a client function that asks both for the number of elements as an aggregate and for the number of base classes, say a function that computes the average number of members for classes and structs that don't actually have bases:

```
float avgNumMembersInRoots(int n_aggs, AggregateWithBases* aggs[]) {
  int sum = 0;
  for (int i = 0; i < n_aggs; i++) {
    if (aggs[i]->numBases() == 0) sum += aggs[i]->numMembers();
  }
  return sum / n_aggs;
}
```

This example illustrates how the interface for `AggregateWithBases` layers on top of the interface for `Aggregate`, expressing the idea that aggregates with bases are aggregates. These use clients don't require `ClassImpl` objects to be `AggregateImpl` objects: the implementation is completely hidden from use clients.

3 Characteristics Of Base-Class Composition

Our example illustrates base-class composition. First notice that our example has two kinds of base classes. The interface base classes, `Aggregate` and `AggregateWithBases`, are base classes with virtual functions, but they are not general C++ classes. They have no implementation. The other kind of base class, AggregateImpl and AggregateWithBasesImpl could be called *implementation base* classes. They do not add specifications of virtual functions and they are not used as public base classes. Thus our classes separate interface and implementation.

Next notice the relationship between these base classes. The AggregateImpl implementation base class derives from the Aggregate interface base class, extending it but only adding implementation. The AggregateWithBases interface base class also derives from the Aggregate interface base class, extending it but only adding more interface specification. The extended interface is implemented in AggregrateWithBasesImpl by composing the extended interface with an implementation of the original interface (AggregateImpl) and adding implementation for the extended interface. This pattern of implementation, extension, and composition can continue to arbitrary depth; we call the pattern base-class composition.

The C++ language features of virtual functions, multiple derivation, virtual bases, and the dominance rule for name lookup in the class DAG [2] combine to enable base-class composition. Virtual functions, of course, make the notion of interface possible. Multiple derivation enables an implementation class to derive from both its interface and an implementation base. The combination of virtual bases and dominance connect the interface and the implementation.

Referring to Figure 1, we see a diamond-shaped pattern rooted at Aggregate. It has an interface base at the top of the diamond, with the interface extended on one leg (AggregateWithBases) and an implementation of the interface base on the other leg (AggregateImpl). The class at the bottom of the diamond (AggregateWithBasesImpl) completes the implementation of the extended interface specified on one path, building on the implementation along the other path.

AggregateWithBasesImpl inherits names from its direct base classes, AggregateWith-Bases and AggregateImpl, and it inherits the names of the indirect base class, Aggregate. Since AggregateImpl and AggregateWithBases are derived virtually from Aggregate, the name numMembers from AggregateImpl dominates the same name inherited along the DAG path through AggregateWithBases. The function numMembers() declared in the public interface base class Aggregate is implemented in AggregateWithBasesImpl by the AggregateImpl base class.

A degenerate triangular version of the diamond-shaped pattern also appears three times in the DAG of Figure 1. The degenerate version omits the extension of the interface. Two of the triangular patterns in Figure 1 are rooted at AggregateWithBases, with a partial implementation of the interface provided on one leg by AggregateWithBases, and the implementation completed on the other leg by ClassImpl (resp., StructImpl). Again, multiple derivation, virtual bases, and name dominance combine to yield base-class composition. The third triangular pattern is rooted at Aggregate.

Two design conventions—separation of interface and implementation and virtual interface base classes—must be adhered to by the programmer to make base-class composition work.

Separation of Interface and Implementation. Base-Class composition relies on the separation of interface and implementation. This means that the specification of the functions callable for an object are separated from the implementation of those functions; for C++, this means that member functions are specified in classes without member data. Classes with member data and implementation of the member functions derive from these classes.

As Martin discussed [1], separating interface and implementation in C++ requires omitting member data in interface base classes and using virtual functions. It is also advantageous to declare the functions to be pure virtual so that the compiler will detect attempts to instantiate the interface and detect failure to override base class functions in the derived class. Other languages, including Modula-3 [3] and Ada, provide separate constructs for interfaces and implementations.

Virtual Interface Base Classes. Derivations from interface base classes must be virtual if the base-class composition approach is to be applied [1]. Without virtual derivation, the interface base classes would be duplicated: ClassImpl would have three Aggregate interfaces. The

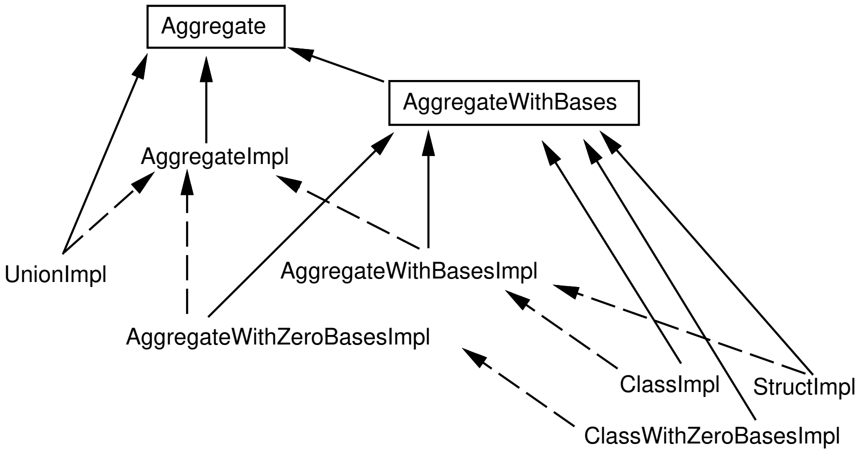

Figure 2: A class DAG extending the DAG in Figure 1 to include additional implementations of the `AggregateWithBases` interface, `AggregateWithZeroBases` and `ClassWithZeroBases`.

implementation names would not dominate the interface names along the `AggregateWithBases` branch and there would be no composition.

These design conventions—separating interface and implementation and deriving virtually from interface bases—represent the programming-time cost of base-class composition. Balanced against these up-front costs are the benefits in maintenance of independent extension, compositions, segregation of use and creation clients, and avoided recompilation. We examine each of these in the following paragraphs.

Independent Extension. Once we have adopted the separation of interface and implementation, the interface can be extended in two ways. Multiple implementations are one kind of extension. For example, we can add `AggregateWithZeroBasesImpl` as shown in Figure 2. This class eliminates the space for the member datum `_num_bases`:

```
class AggregateWithZeroBasesImpl :
  public virtual AggregateWithBases,
  private AggregateImpl {
public:
  AggregateWithZeroBasesImpl(int num_members) :
    AggregateImpl(num_members) {
  }
  virtual int numBases() const { return 0; }
};
```

Of course the saving in this case is trivial because `AggregrateWithBasesImpl` is trivial. Use clients of `AggreateWithBases` work against both implementations.

Adding virtual functions to the interface to create a richer interface is the other kind of extension. For example, we could create a `Class` interface derived from `AggregateWithBases` without affecting the implementation extensions derived from `AggregateWithBases`. These two kinds of extensions are for different purposes, alternative implementation versus additional interface.

Reuse Through Composition. To reuse the `AggregateImpl` implementation of the `Aggregate` interface in the implementation of the `AggregateWithBases` interface layer, `AggregateWith-BasesImpl` derives publicly from the interface `AggregateWithBases` and privately from the implementation `AggregateImpl`. Together, these two base classes form a composition that implements the `Aggregate` part of the `AggregateWithBases` interface. The virtual functions defined along the `AggregateImpl` branch (just `numMembers()` in this case) dominate the virtual functions in the `Aggregate` base class along the `AggregateWithBases` branch.

Once we adopt separation of interface and implementation and use `virtual` interface base classes, composition can become a ubiquitous tool in class design. For example, in Figure 2 we can create `ClassWithZeroBasesImpl` simply by composition:

```
class ClassWithZeroBasesImpl :
 public virtual AggregateWithBases,
 private AggregateWithZeroBasesImpl {
public:
  ClassWithZeroBasesImpl(int num_members) :
     AggregateWithZeroBasesImpl(num_members) {
  }
};
```

Here we attach to the extended interface `AggregateWithBases` and reuse one of it implementations. The usage rules for base-class composition are always the same: the interface is public and virtual and the reused implementation is encapsulated with the same consideration as member data, usually as a private base class.

Forcing Access Through Interfaces. In the preceding example code, using private derivation forces member functions to be called through interface references. For example,

```
int totalItems(const ClassImpl& c) {
   // WRONG: AggregateImpl::numMembers() const is a private member
   return c.numBases() + c.numMembers();
}
```

does not compile because public members inherited via private derivation are private. This segregates clients into use clients able to call through the interface only and creation clients unable to call interface functions on objects. This segregation encourages us to write code applicable to all `Aggregate` objects using `Aggregate` references or pointers and to avoid writing code tied to specific implementations of the interface like `ClassImpl`. Whether or not this segregation should be used is a matter of design: some libraries may wish to encourage object access through interface base classes only while others will allow functions at all levels of the class DAG to be called. Access declarations can be used to restore public access in cases in which forcing access through interfaces is not appropriate.

Avoiding Recompilation of Use Clients. Base-class composition also avoids recompilation of use clients when private implementations are redefined, a vital advantage when building large systems. While encapsulation ensures that use clients do not depend on implementation, separating interface and implementation provides a stronger decoupling. Use clients can be compiled to the interface, implementations can be compiled to the interface, and they need only be connected at link time.

4 Why Alternatives are Less Robust

In our experience, base-class composition provides a systematic and robust architectural pattern for object-oriented programs in C++. We have outlined the technique in the preceding section. Here we support our claim that it is more robust—resistant to unforeseen errors—and more maintainable than alternative designs. We pose alternatives in terms of our example.

Combined interface and implementation. We might dispense with the interfaces altogether. A class DAG like that shown in Figure 3 would provide the same implementation reuse as the one in Figure 1. Inheriting interface and implementation together combine four classes into two classes (`AggregateCombined` and `AggregateWithBasesCombined`) and eliminates all multiple derivation. While these may be counted as advantages, client functions are now tied to specific implementations. Modifying the implementation of `AggregateCombined` forces all use clients

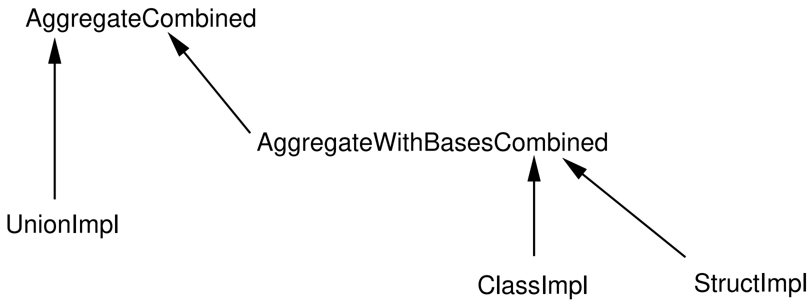

Figure 3: An alternative class DAG design to represent C++ aggregates that uses interface base classes combined with implementation. Compared to Figure 1, the content of `Aggregate` and `AggregateImpl` are combined in `AggregateCombined` and the content of `AggregateWithBases` and `AggregateWithBasesImpl` are combined in `AggregateWithBasesCombined`. Public derivation (solid arrows) must be used here to expose the virtual functions specified in the base classes.

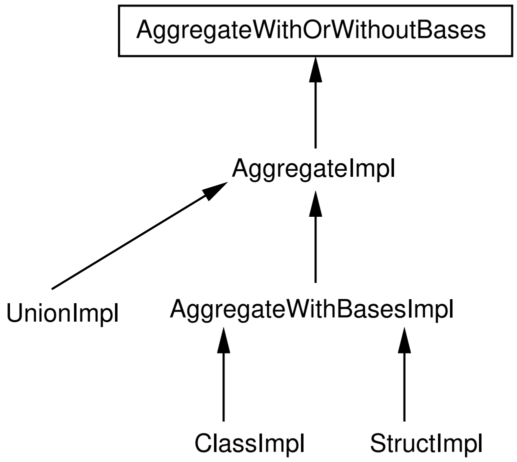

Figure 4: Alternative class DAG using a fat interface in representing C++ aggregates. Compared to Figure 1, this DAG combines the `Aggregate` and `AggregateWithBases` interface base classes into one large interface. Public inheritance must be used here in the derived classes to expose the specifications in the interface.

of `AggregateCombined`, `AggregateWithBasesCombined`, and any further layers to be recompiled when anything is changed.

Combining implementation and interface also prevents alternative implementations from being used by one set of client functions. For example, we cannot define a `ClassWithZeroBasesImpl` object as we did in the preceding section and then use it in use clients expecting `AggregateWithBasesCombined` references or pointers.

We can extend the interface of `AggregateWithBasesCombined` in the same manner that we built it from `AggregateCombined`. For this reason, combined interface and implementation classes work in systems that do not use virtual function interface clients as a major program design element.

Fat interfaces. The number of classes can be reduced by lumping all of the interface functions into one interface base class, say `AggregateWithOrWithoutBases`, as shown in Figure 4. Those parts of the interface not pertinent to a given derived class type are coded in some hopefully harmless way. For example, we could code `AggregateImpl::numBases()` for `AggregateImpl` to always

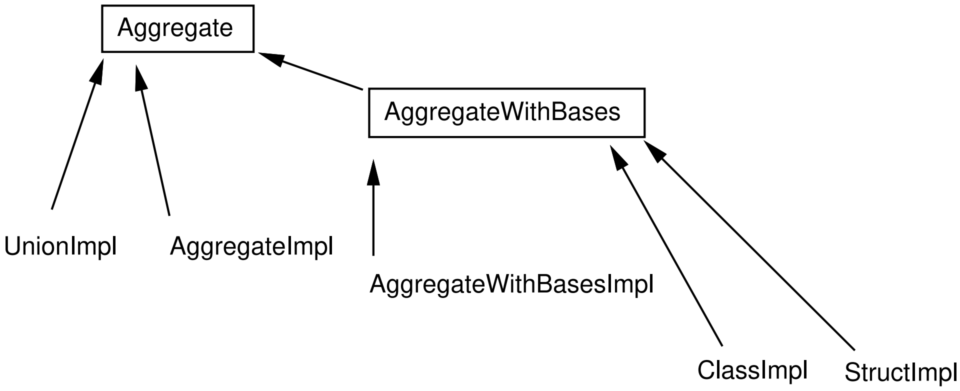

Figure 5: Alternative class DAG representing C++ aggregates using member function forwarding. Compared to the class DAG shown in Figure 1, this DAG has no multiple inheritance. The classes `AggregateImpl` and `AggregateWithBasesImpl` are used as private member data.

return an "impossible" value like `-1` or to throw an exception. Such lumped interfaces, called *fat interfaces* in [4, § 13.6], replace static type checking with either runtime checks or undetected errors.

In addition, the fat interface approach also makes us choose between implementation inheritance and encapsulation. In Figure 4, `ClassImpl` must be derived publicly from `AggregateImpl` to expose the `AggregateWithOrWithOutBases` interface. If we derive `ClassImpl` from the interface publicly and directly we cannot also inherit the implementation of `AggregateImpl` privately unless we build a structure equivalent to the base-class composition. If we extend the interface, adding functions to `AggregateWithOrWithoutBases`, all use clients must be recompiled, even if they use only the equivalent of the `Aggregate` interface.

We can add new implementations in the fat interface approach and these implementations don't require recompilation of use clients. For this reason, fat interfaces appear in small projects with heavy use of virtual functions.

Implementation Reuse via Member Subobjects. As a final alternative, we abandon implementation reuse through inheritance but retain the layered interface, as shown in Figure 5. Instead of obtaining an `AggregateImpl` subobject via inheritance, `AggregateWithBasesImpl` could have a member subobject, like this:

```
class AggregateWithBasesImpl :
  public AggregateWithBases {
public:
  AggregateWithBasesImpl(int num_members, int num_bases) :
    _aggregate_impl(num_members),
   _num_bases(num_bases) {
  }
  virtual int numMembers() const { return _aggregate_impl.numMembers(); }
  virtual int numBases()   const { return _num_bases;           }
private:
  AggregateImpl _aggregate_impl;
  int _num_bases;
};
```

The function `numMembers()` is said to be *forwarded* to the member `_AggregateImpl`. This seems fine until you try to compile the class and get an error message. `AggregateImpl` is an abstract base class since it doesn't implement the pure virtual function `kind()`. To use forwarding in this situation, you must add a dummy implementation of `kind()` to `AggregateImpl`.

Once the code is correct, this alternative is indistinguishable from the base-class composition as far as client functions are concerned: the interfaces are identical and the implementations are completely encapsulated. However, the forwarding functions themselves are a maintenance item not required by base-class composition. In systems of realistic complexity, this style of reuse becomes tedious and error-prone. Base-class composition expresses the same relationships without imposing the maintenance overhead of forwarding functions.

In our experience, larger more mature systems designed to use virtual functions often adopt the member subobject approach, limiting the use of derivation to building subtype relations.

5 The Cost of Base-Class Composition

If base-class composition is superior to alternatives, why isn't it used in more object-oriented C++ programs? History is part of the answer. Multiple derivation, virtual bases, and dominance lookup are relatively new additions to C++ and early experiences with these features were not altogether positive (see Section 8). But even those programmers aware of the advantages select alternatives as a practical engineering tradeoff. The problem is performance.

As far as we are aware, current C++ compilers use roughly the scheme outlined in [2] to implement virtual bases with virtual functions. Minor optimizations aside, this scheme adds to each object one virtual function table pointer per base class and creates one virtual function table per base class for each class. In addition, one pointer to each virtual base class is needed for each subobject declaring a virtual base class. Thus we would expect `ClassImpl` in Figure 1 to take 11 words of memory: two words for the integer members, five words for virtual function table pointers, and four words for pointers to virtual bases. (The compiler we tried this on used ten words, saving a word by sharing two virtual function table pointers.) The size overhead carries a proportional runtime overhead since each pointer in the object must be initialized when the object is created.

Since each layer of base-class composition adds at least two base classes with virtual functions, these overheads increase with increasing layers. Such scaling works against the application of base class composition to the very kinds of problems—large systems—at which the technique is most adept.

6 Optimization of Base-Class Composition

The costs of base-class composition can be eliminated by a straightforward compiler optimization. This claim would be best proven by a C++ compiler that did not have the overheads of existing implementations. To convince compiler writers to implement such an optimization, we need both the motivation that we presented in the preceding sections and convincing arguments that the optimization will succeed. In this section we outline the optimization and argue that it will succeed.

The core of our argument lies in recognizing that the restricted interface base classes we need for base-class composition only specify the contents of virtual function tables, contents that are determined at compile time. Thus we can transform the class DAG at compile time to eliminate the pointers needed for more general virtual base classes, as long as the virtual function tables are filled correctly.

We begin with a definition: a *pure abstract base class* is an abstract base class (a class having at least one pure virtual function) with no data members and only C++-generated constructors, possibly derived from other pure abstract base classes. This restricted kind of interface base class is the kind we advocate for base-class composition. A compiler can test for a pure abstract base class unambiguously by examining a class and its base classes.

We claim that a DAG containing pure abstract base classes can be transformed into an equivalent DAG having no virtual base class derivation involving a pure abstract base class. By equivalent we mean that programmers will not be able to detect the difference by ordinary means—more on that shortly. Obviously, if this claim is true, no virtual base pointers are needed since there are no virtual bases.

Removing virtual derivation "unfolds" the pure abstract base class portion of the DAG into a tree, replicating some of the pure abstract base classes. Each resulting tree branch can be implemented

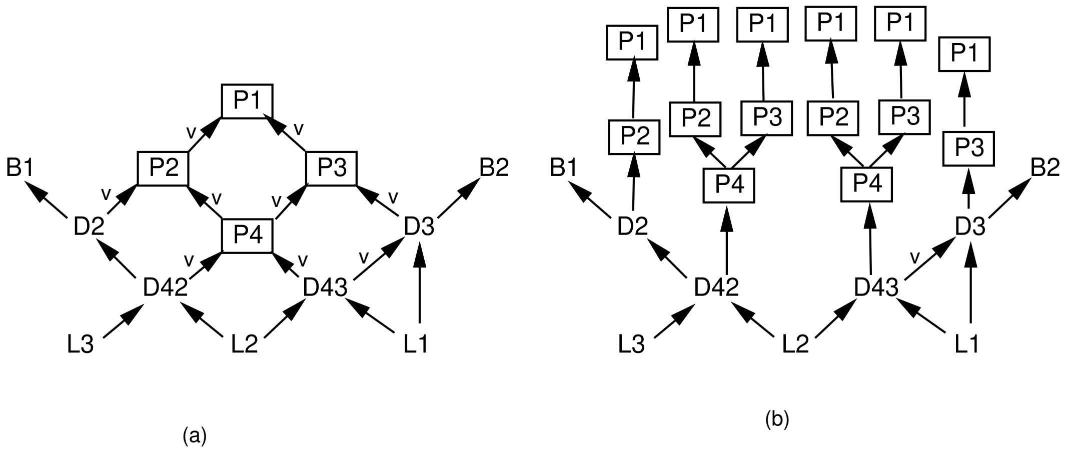

Figure 6: A hypothetical class DAG, (a), and the result of optimizing it (b).

with a single virtual function pointer [2]. Consequently, the transformation eliminates virtual base pointers and most of the pointers added by multiple derivation.

To understand this transformation, consider the example class DAG in Figure 6a. Classes `P1`, `P2`, `P3`, and `P4` are pure abstract base classes; the other classes are arbitrary. The class names starting with `D` border on the part of the DAG that will be transformed; those starting with `B` and `L` are just other parts of the DAG. Arrows point to base classes and virtual derivations are marked with a `v`. For example, the derivation of `D43` from `D3` is virtual but `D3` is not a pure abstract base class.

The transformed DAG is shown in Figure 6b. With the virtual derivations from pure abstract base classes changed to non-virtual derivations, the pure abstract base classes are duplicated in the resulting DAG. For the present DAG, base `P1` appears 6 times as base classes for type `L2` rather than once. However, since none of the classes that are duplicated contain data, the storage required for any object from any class in the transformed DAG is less than that of classes in the original DAG.

Now we claim that these DAGs are equivalent in the following sense: the source code using the original DAG can be rewritten to use the transformed DAG such that the rewritten code works the same as the original in all ways that do not depend on implementation dependent object-layout details. Object sizes, offsets of members, and the structure of virtual function tables will change (good!), but otherwise the program will be the same.

After the DAG transformation, the compiler must (a) forward member function calls defined along one replicated path to the other path, and (b) disambiguate pointer and reference conversions from derived classes to replicated pure abstract base classes. For example, member functions in `D2` that override pure virtual functions in `P1` or `P2` will have to be overridden in `D42`. `P1` pointers initialized with `D42` pointers will have to be converted up through one path, say `P4` to `P2` to `P1`. All paths are equivalent since `P1` is replicated by design.

The rewriting is sufficient because there are only two things one can do with a pure abstract base class: (1) initialize a reference or pointer of the base type with a reference or pointer, respectively, of one of its derived types, or (2) call one of its member functions through a reference or pointer. The initializations can be done with replicated classes caused by removing the virtual specifiers just as well as to the shared classes because the contents of the replicated classes are fixed at compile time. Calls to member functions give the same result for both shared and replicated classes because the functions have the same definitions.

Programmers using objects from classes derived from pure abstract base classes cannot detect whether or not these objects contain shared or replicated subobjects from these bases. Again, the lack of member data in pure abstract base classes allows the optimization: there is no shared data to point to or initialize. In Figure 6b, class `D43` derives virtually from `D3` even in the transformed DAG because `D3` is not a pure abstract base class and cannot be optimized in the manner described here.

7 Optimization of Private Base Classes

The particular pattern of base-class composition can be further optimized: Private base classes in the transformed DAG whose only public base classes are pure abstract base classes can be rewritten as private members without virtual functions of their own. For example, if `D42` derived privately from `D2` and `D2` derived privately from `B1` then the `D2` portion of the DAG would be stored as a private member of `D42` and, further, this member would not have any virtual function pointers inside of it.

To accomplish this second optimization, the compiler must rewrite classes into a concrete and an interfaced form. For a class like `D2` in Figure 6, call the type of the concrete form `ConcreteD2`. Class `ConcreteD2` is exactly like `D2`, except that it does not derive from any pure abstract base class and its member functions are not declared virtual. A new interfaced `D2` class is written that derives from the same pure abstract base classes of the original `D2` and declares the same member functions as `D2`, but implements these functions by forwarding to a private `ConcreteD2` member datum.

All clients of the interfaced `D2` see the same behavior as they saw from the original `D2`. However, the compiler now has a private representation class, `ConcreteD2`, that does not have virtual functions and hence has no virtual function pointers. This private representation can be used whenever `D2` appears as a private subobject, either as a private member datum or, as in the case of base-class composition, as a private base class.

This second optimization succeeds because of encapsulation and because, unlike base-class pointers or references that can bind to different objects types at run-time, member objects have a fixed known type. As the compiler implements a private subobject, it has all of the source code that can call member functions on that subobject (encapsulation) and it can resolve virtual function calls on that subobject at compile time without going through a virtual function table (one type). Thus private subobjects can be implemented using the concrete private representation of a class.

To illustrate the impact of these optimizations on a recognizable example, we return to the C++ aggregate example. Here is part of the result of applying these optimizations by hand:

```
class ConcreteAggregateImpl {
public:
  ConcreteAggregateImpl(int num_members) : _num_members(num_members) {}
  int numMembers() const { return _num_members; }
private:
  int _num_members;
};

class AggregateImpl :
  public Aggregate {
public:
  AggregateImpl(int num_members) : _aggregate_impl(num_members) {}
  virtual int numMembers() const { return _aggregate_impl.numMembers(); }
private:
  ConcreteAggregateImpl _aggregate_impl;
};

class ConcreteAggregateWithBasesImpl {
public:
  ConcreteAggregateWithBasesImpl(int num_members, int num_bases) :
    _aggregate_impl(num_members),
    _num_bases(num_bases) {
  }
  int numBases() const { return _num_bases; }
  int numMembers() const { return _aggregate_impl.numMembers(); }
private:
  ConcreteAggregateImpl _aggregate_impl;
  int _num_bases;
};
```

```
class AggregateWithBasesImpl :
  public AggregateWithBases {
public:
  AggregateWithBasesImpl(int num_members, int num_bases) :
    _agg(num_members, num_bases) {
  }
  virtual int numBases() const { return _agg.numBases(); }
  virtual int numMembers() const { return _agg.numMembers(); }
private:
  ConcreteAggregateWithBasesImpl _agg;
};
```

The concrete version of each class provides pure implementation, without any connection to the rest of the class DAG. As a result, there is no overhead from virtual base pointers or virtual function pointers. These concrete classes provide implementation reuse. Since each non-interface class is derived through a single inheritance chain, only one virtual function table pointer is needed. Thus, the overhead of this scheme is one word: an instance of the transformed `ClassImpl` takes three words, two for the integer member data and one for a virtual function table pointer.

Separation of interface and implementation is central to the success of the transformation. Imagine for the moment that our `Aggregate` base class had some member datum. As a virtual base class, a single copy of this datum would be included in a `ClassImpl` object. But in transforming this system to single-inheritance, a private `AggregateImpl` member datum would replace the private base class and two copies of the `Aggregate` member would be enclosed in a `ClassImpl` object: one from the `AggregateImpl` member and one remaining in the `Aggregate` base class. The pointers introduced by C++ compilers to support virtual function calls in the presence of multiple inheritance and virtual bases give offsets to adjust the `this` pointer to accommodate base class member data. Separation gives interfaces with no data; no data means no pointer offsets.

8 Related Work

Separation of Interface and Implementation. Section 3 follows the work of Martin [1]. He started with an "ideal" model for interface-based programming and showed that a C++ programming style could almost be equivalent. He concluded with a suggestion that would improve the language support for this kind of style. Since Martin's paper appeared, the C++ language has been altered and the technique we call base class composition is the same as the style advocated by Martin but using the new language rules. As Martin predicted, the result is a clean mechanism for programming with interfaces, as we illustrate here.

Martin's assessment of the performance penalty omitted runtime initialization of the virtual-base and virtual function table pointers. In our experience, C++ programmers prefer the cost of maintaining forwarding functions and even live with mixed interface and implementation rather than accept the space and time overheads. For this reason, we investigated compiler optimization for this important technique.

We have also distinguished the kinds of program designs—those that rely on use clients—that will benefit from separation of interface and implementation. Programs that do not use virtual function calls will only see the programmer-time cost of base class composition and no benefit.

We Need Multiple Derivation. Cargill has argued [5, 6] that many cases of multiple derivation can be reimplemented using single derivation with some advantages and that, given the complex semantics of multiple derivation, it is not a useful language feature. In [6], he gives three examples using multiple inheritance. The two he argues are failures do not separate interface and implementation; moreover the implementations are not layered to allow reuse. His third example, the one he thinks may be a reasonable use of multiple inheritance, uses two interfaces but no implementation reuse.

We believe Cargill's conclusion was reached without the benefit of enough examples of large systems of classes designed for extensibility. Waldo[7] analyzed Cargill's arguments against multiple derivation by suggesting that three kinds of base classes might be distinguished: 1) those for implementation inheritance, 2) those for interface inheritance, and 3) those for data inheritance. He argued that Cargill focused on the first, but that the latter two are more important in large systems and that multiple derivation is needed to support the use of such base classes.

Waldo's classes for implementation inheritance correspond to our classes with combined interface and implementation. We agree with his arguments for interface inheritance, but we see a broader role for its use. By splitting combined interface and implementation classes then composing these classes via multiple derivation we achieve the benefits Waldo cites for interface inheritance and we achieve implementation reuse. Base-class composition provides simple and consistent guidelines allowing us to ignore the complexity of the general multiple derivation in C++.

We Need The Dreaded Virtual Base Diamond. Meyers [8] also argues against multiple derivation and especially against virtual bases used in the "dreaded diamond-shaped inheritance graph" (p. 165). He argues that one cannot predict when a virtual base should be used; we say that every derivation from an interface base should be virtual. He argues that constructors for virtual bases are problematic; we say don't put data in abstract base classes and you won't need constructors in virtual bases. He argues that ambiguities can arise in multiple derivation and that dominance is mysterious; we say that using multiple derivation in a disciplined way with one branch adding new virtual functions and the other implementing previous virtual functions harnesses the mysterious powers of the dominance rule. He argues that virtual bases do not allow casting a pointer to a base class to a pointer to a derived class; we agree with his assessment that this is not very important.

Meyers then goes on to give an example of multiple derivation using public interface and private implementation base classes. He sees this example as useful and comprehensible, but says (p. 165) that "it's no accident that the dreaded diamond-shaped inheritance graph is conspicuously absent." However, if he simply added virtual derivation to his graph, he could have completely eliminated the member function definitions in his final class, together with their maintenance.

Meyers, like Cargill, exposes many problems with the use of multiple derivation and the use of virtual bases. Their arguments focus on practical examples and they admit possible exceptions. We agree that the uses they explore do have problems. However, we blame failure to separate interface and implementation for most of the problems and find important uses for both multiple derivation and virtual bases used for base-class composition.

Cargill [6] and Meyers [8] advocate manual rewriting as means of avoiding the overhead of multiple derivation and virtual base classes. In effect they favor a coding style that replaces base-class composition by hand optimization. Here we have demonstrated that base-class composition is simpler and easier to maintain; it can be a key building block for layered systems and the optimization can be done automatically.

Not All Public Base Classes Should be Virtual. Sakkinen[9] discusses the C++ inheritance model and concludes that public base classes should always be virtual while private base classes should always be nonvirtual. Base class composition does conform to these guidelines, but we disagree with the guidelines in general. Without virtual functions, public base classes have a very different role in class designs and we cannot decide to make them virtual in all cases. With mixtures of interface and implementation, we must let the required object state dictate the choice of virtual or nonvirtual bases.

In another paper,[10] Sakkinen discusses initialization problems with virtual base classes. Again, base class composition uses no constructors in virtual bases and initialization is therefore not relevant to the programmer.

9 Conclusion

Base-class composition allows large, layered systems of interfaces to be implemented robustly and simply. Separate interface and implementation base classes are composed to form a base for further implementation without compromising extensibility or encapsulation. We believe that when virtual functions are used by client functions—when programming through interfaces—base class composition should become an important tool for C++ programmers. However, the performance barrier must be overcome first. We have proposed optimizations that allow compilers to eliminate all but one of the pointers needed to implement C++ for systems using base class composition.

Acknowlegements

We appreciate the suggestions for improving this paper that were made by Michael Karasick, Derek Lieber, Chris Laffra, Barry Rosen, Michael Fraenkel, Noel Sales, Ralph May, Paul Golick, and Lars Hougaard. We also thank Ernie Choi for his encouragment.

References

[1] Bruce Martin. The separation of interface and implementation in C++. In **USENIX C++ Conference Proceedings**, pages 51–63. USENIX Association, April 1991.

[2] Margaret A. Ellis and Bjarne Stroustrup. **The Annotated C++ Reference Manual**. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1990.

[3] Samuel P. Harbison. **Modula-3**. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[4] Bjarne Stroustrup. **The C++ Programming Language**. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, second edition, 1991.

[5] T.A. Cargill. Does C++ really need multiple inheritance. In **USENIX C++ Conference Proceedings**, pages 315–323. USENIX Association, April 1990.

[6] Tom Cargill. **C++ Programming Style**. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992.

[7] Jim Waldo. Controversy: The case for multiple inheritance in C++. **Computing Systems**, 4(2):157–171, 1991.

[8] Scott Meyers. **Effective C++: 50 Specific Ways to Improve Your Programs and Designs**. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992.

[9] Markku Sakkinen. A critique of the inheritance principles of C++. **Computing Systems**, 5(1):62–110, 1992.

[10] Markku Sakkinen. How should virtual bases by initialized (and finalized)? **C++ Report**, 5(3):44–50, 1993.