SANDIA REPORT

SAND2007-4078
Unlimited Release
Printed October 2007

The Pure Nonmember Function Interface Idiom for C++ Classes

Roscoe A. Bartlett

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2007-4078
Unlimited Release
Printed October 2007

The Pure Nonmember Function Interface Idiom for C++ Classes

Roscoe A. Bartlett

Abstract

A pure nonmember function interface to an abstract C++ class might provide the best approach for keeping clean minimal interfaces, insulating client code from changes to an interface, and providing a uniform interface when other nonmember nonfriend functions are used. The proposed idiom is the logcial combination of the Nonvirual Interface (NVI) idiom and and the Nonmember Nonfriend Function idiom. This idiom also applies equally well to concrete C++ classes and can even be used with great advantages for constructors as well.

Contents

Appendix

1 Introduction

Object-oriented programming has been used and refined for many decades in a variety of programming languages. Some of the most basic descriptions of object orientation speak of *programming with objects* and refer to such concepts as *object methods*, *polymorphism*, and *encapsulation*. These concepts can be expressed in different ways in different programming languages and each language lends itself to different idioms for how object orientation can be used to its fullest. Here we focus on the C++ language and combine some of the more modern idioms being advocated for C++ to propose a unifying idiom involving the consistent use of nonmember functions. This use of nonmember functions decouples external clients of abstract interfaces and concrete classes from the details and changes to the public interfaces of these abstract classes. Here we will draw on the advice of several respected authors on C++ programming [5, 4].

Our primary focus is on issues related to object-oriented programming in the C++ language and specifically the interaction between an abstract interface (consisting of pure virtual functions), clients that use objects through the interface, and subclasses of the interface that provide concrete implementations of the virtual functions. A secondary focus of this discussion addresses how nonmember functions can also be used as the interface to concrete classes and even for constructors and describes the benefits of such an approach.

With respect to abstract interfaces, our main goal here is to describe a light weight approach for developing C++ interfaces and encapsulation mechanisms which protect clients of an abstract C++ interface from changes to the specification of the interface's virtual functions. This is especially critical when the interface represents an important interoperability mechanism and is part of a library which may have many diverse and unaccessible external clients which the library developers can not directly access to change. Minimizing the impact of code refactorings is even more important in C++ than in other languages because of the lack of good quality refactoring tools for C++ code. Even when you have good refactoring tools, it is nontrivial to push refactorings out to external clients of your class library and few if any tools for any language support this feature currently.

What we want is to have an approach to developing, maintaining, and using abstract C++ interfaces that:

- *Provides for the absolute minimal abstract C++ interfaces*: An abstract C++ interface is the critical specification of the capabilities of an object which must be able to cover the needs of a large set of potential clients and allow great flexibility and efficiency in the implementation of subclasses. The more minimal an interface is, the more likely it will be adopted by a larger community and the easier it will be to develop powerful "Decorator", "Composite" and other such general subclasses. Minimal and efficient interfaces are especially critical for interoperability.
- *Maintains a uniform, consistent, and convenient interface for the clients of the abstract interface*: We want clients to be able to access the capabilities of the object in a clean way that is robust to changes in the interface.
- *Avoids many of the "gotchas" associated with object oriented programming in C++*: In particular, we want to avoid difficulties associated with overloaded virtual functions [2, Item 73], virtual functions with default arguments [4, Item 37], and other such problems.

- *Allows for changes to an abstract interface's virtual function set in a way this is consistent with the above goals*: As requirements become more clear or change over the life of a piece of software, changes to the specification of the virtual function set for an interface will be inevitable in order to satisfy the new requirements in an efficient and safe way, and to maintain a minimal interface. We want to avoid a sub-standard abstract interface that is cluttered with backward-compatible functions for older clients. Ideally, the integrity and the quality of the current incarnation of an interface should not suffer from having been incrementally developed where compromises where made to support older clients at the expense of the interface. We want the interface to be the same quality as if it were nearly totally redesigned after the fact.

2 The NVI and Nonmember Nonfriend Function Idioms

In particular, two idioms have been advocated that are designed to address many of the issues raised above: The Nonvirtual Interface (NVI) idiom [5, Item 39], and the "nonmember nonfriend function" idiom [5, Item 44]. Other guidelines that are pertinent to our discussion are "prefer minimal classes to monolithic classes" [5, Item 33], "prefer providing abstract interfaces" [5, Item 36], "practice safe overriding" [5, Item 38] [2, Gotcha 74], and "avoid overloading virtual functions" [2, Gotcha 73].

The Nonvirtual Interface (NVI) idiom [4, Item 35] advocates making all virtual functions non-public (i.e. either private or protected) and making all public functions nonvirtual. For example, we might have an interface that looks like:

```
class BlobBase {
public:
  // Non-virtual public interface.
  // Note: Default argument values are defined here and here only!
  void foo(int a=0) { implNonconstFoo(a); }
  void foo(int a=0) const { implFoo(a); }
protected: // or private:
  // Pure virtual non-public functions to be overridden
  virtual void implNonconstFoo(int a) = 0;
  virtual void implFoo(int a) const = 0;
};
```

The details of the NVI idiom are given in [5, Item 39] and [4, Item 35] but basically the idiom allows clients to call regular member functions and overloaded member functions on an object without the problems associated with overloaded virtual functions. The NVI idiom also avoids problems with default function arguments since the default values are only defined in the non-public, nonvirtual function interface.

Another idiom that is advocated in [5, Item 44] and [4, Item 23] is to prefer writing a function as a nonmember nonfriend function unless it needs access to private or protected members. This increases encapsulation and improves modularity. Typically, this idiom is described in the context of concrete classes which actually have private data, but it is also applicable for abstract interfaces as well. If some capability can be performed just using the existing public interface, then that capability should be implemented as a nonmember nonfriend function. Adding another nonvirtual function to the interface (or worse making the new function virtual with a default implementation) mostly just clutters up the abstract interface and complicates maintenance. For example, some function `goo(...)` could be implemented in terms of `BlobBase::foo(int)` as:

```
void goo( BlobBase &obj )
{
  obj.foo(0);
  obj.foo(1);
}
```

The NVI idiom and "nonmember nonfriend function" idiom, can and should be used together, but they are also somewhat at odds with each other. The NVI idiom implies that all operations that are

directly implemented as virtual functions on the abstract interface would be accessed using corresponding public nonvirtual member functions. The "nonmember nonfriend function" idiom dictates that all other functions would be implemented as nonmember nonfriend functions. However, the straightforward combination of these two idioms has several disadvantages:

- *The client interface is a mix of member and nonmember functions*: The most obvious disadvantage of having an interface composed of both nonmember and member functions is that it can be hard for the developers of client code to remember how to call an operation. For instance, is the operation `foo(...)` called as `obj.foo(i)` or as `foo(obj,i)`?
- *Changes to the virtual function structure are difficult to handle*: A change to the virtual function structure requires that either clients be changed or that the interface be polluted with public functions that no longer need direct access to the nonpublic virtual functions. For example, what if requirements for the abstract interface change such that it would be beneficial, from a design point of view, to change the specification of a virtual function. The change might involve a modification to the signature and/or the behavior of the function. Such a change in the virtual function would naturally involve a similar change in the corresponding public nonvirtual member function that calls the virtual function. Let's also assume that the current capability of the function in question can be maintained through a simple function that calls the newly updated function. Now the problem; how do we implement this change and how does this change impact the current clients of the interface? There are one of two possible ways to refactor the code: a) move the function from a member function to a nonmember function, or b) leave the current public nonvirtual member function in the abstract interface and make it call the newly updated member function. Both of these choices are fraught with problems.

Let's examine the two possibilities for handling changes to the virtual function structure of the abstract interface mentioned above. As an example, consider a new set of requirements where the `foo()` member functions need to be changed to accept a `Bar` object (represented through its own abstract interface) instead of just an integer, and the functions also need to accept an extra boolean argument. In addition, let's assume that the old meaning and behavior of the `foo()` functions can be retained by using a default implementation of `Bar` called `DefaultBar`. We consider the two approaches for dealing with such a change below.

a) If we want to keep the abstract interface minimal and be consistent with the "nonmember nonfriend function" idiom, then we want to choose option 'a' which involves moving the old public nonvirtual member `foo()` functions out of the interface and making them new nonmember nonfriend functions. In this case, the updated class interface `BlobBase` would look like:

```
namespace BlobPack {

class Bar;

class BlobBase {
public:
  // Public non-virtual client interface
  void foo( const Bar &bar, bool flag = false )
    { implNonconstFoo(bar,flag); }
```

```
  void foo( const Bar &bar, bool flag = false ) const
    { implFoo(bar,flag); }
protected: // or private:
  // Pure virtual non-public functions to be overridden
  virtual void implNonconstFoo( const Bar &bar, bool flag ) = 0;
  virtual void implFoo( const Bar &bar, bool flag ) const = 0;
};

} // namespace BlobPack
```

and the nonmember form of the old `foo()` functions would look like:

```
void BlobPack::foo( BlobBase & obj, int a)
{
  obj.foo(DefaultBar(a),false);
}

void BlobPack::foo( const BlobBase & obj, int a)
{
  obj.foo(DefaultBar(a),false);
}
```

However, this refactoring would require changing the calling syntax used by all client code that currently calls the old version of the `foo()` member function. This change is simple to make since one just needs to replace `blob.foo(i)` with `foo(blob,i)` and one could almost write a script to perform the refactoring. However, this type of automated refactoring could never be performed 100% correctly and preexisting clients outside of the library developer's control (i.e. clients of our libraries) could not be changed easily. While this approach maintains a clean abstract interface and is consistent with both the NVI and the "nonmember nonfriend function" idioms, is has the disadvantage of requiring clients to change their code, which may be undesirable, impractical, and/or too expensive.

b) If we want to minimize the impact on existing clients (i.e. if our library is widely used by external clients out of our control), then we might want to to choose option 'b' to leave the old public nonvirtual functions in the abstract interface and to augment the interface with the new public nonvirtual functions corresponding to the refactored nonpublic virtual functions. The refactored class interface in this case would look something like:

```
#include "DefaultBar.hpp"

namespace BlobPack {

class BlobBase {
public:
  // Old public nonvirtual functions that do not need direct access
  void foo(int a = 0)
    { foo(DefaultBar(a),false); }
  void foo(int a = 0) const
    { foo(DefaultBar(a),false); }
```

```
  // Public nonvirtual functions that need direct access
  void foo( const Bar &bar, bool flag = false )
    { implNonconstFoo(bar,flag); }
  void foo( const Bar &bar, bool flag = false ) const
    { implFoo(bar,flag); }
protected: // or private:
  // Pure virtual non-public functions to be overridden
  virtual void implNonconstFoo( const Bar &bar, bool flag ) = 0;
  virtual void implFoo( const Bar &bar, bool flag ) const = 0;
};

} // namespace BlobPack
```

The refactoring shown above has the advantage that the clients don't need to be changed (other than needing to be recompiled). However, the problem with this approach of course is that it no longer maintains a clean minimal interface and is in direct violation of the "nonmember nonfriend function" idiom. Over time, such refactorings will result in a bloat of the abstract interface which is discouraged by many experts in object-oriented programming in C++ [5, Item 33].

In the next section, an approach for addressing the problems of combining these two idioms is presented which involves the adoption of a pure nonmember function interface.

3 The Pure Nonmember Function Interface Idiom for Abstract Classes

Here we present a variation of the NVI idiom that is more complementary with the “nonmember nonfriend function” idiom. The idea is the replace the pubic nonvirtual member functions in the abstract interface with nonmember friend functions. A simple interface using the “pure nonmember function interface” idiom would look like:

```
namespace BlobPack {

class BlobBase {
  // Prototypes for nonmember friend functions that
  // will directly call virtual functions.  Note that default
  // argument values are defined here and here only.
  friend void foo( BlobBase & obj, int a = 0 );
  friend void foo( const BlobBase & obj, int a = 0 );
protected: // or private:
  // Pure virtual non-public functions to be overridden
  virtual void implNonconstFoo(int a) = 0;
  virtual void implFoo(int a) const = 0;
};

} // namespace BlobPack
```

and the nonmember friend functions would be implemented as:

```
void BlobPack::foo( BlobBase & obj, int a )
{
  foo.implNonconstFoo(a);
}

void BlobPack::foo( const BlobBase & obj, int a )
{
  foo.implFoo(a);
}
```

Note that the C++ standard allows friend functions to be declared directly within a C++ class declaration. Therefore, declaring a nonmember friend function is not much more verbose than declaring a member function.

Other functions that can be implemented in terms of the existing capabilities on the object without requiring privileged access would be implemented as nonmember nonfriend functions such as:

```
 void goo( BlobBase &obj )
{
   foo(obj,0);
   foo(obj,1);
 }
```

This approach has all of the same advantages of the NVI idiom with respect to allowing for function overloading without problems and for allowing for a single definition of default parameter values. Note that it is critical that the virtual functions themselves must remain non-public since we can't allow clients to be calling these directly (for lots of reasons). Therefore, these special nonmember functions must be friends in order to call the non-public virtual functions.

Even through at first sight replacing the public nonvirtual member functions with corresponding nonmember friend functions looks to be more complicated, there are several advantages to doing this:

- *The client accesses capabilities in a more consistent way*: A client invokes every operation on an object using a nonmember function, independent of how that function was treated. For example, the client would call `foo(obj,i)` or `goo(obj)` consistently as nonmember functions without having to worry how these are implemented now or in the future.

- *Changes to the structure of the virtual function set can be handed without affecting clients and without cluttering the abstract interface*: If a virtual function needs to be modified in some way, then the nonmember function that calls that virtual function can be changed and the old nonmember friend function can be turned into a plain nonmember nonfriend function which calls the new friend function and can be removed from the abstract interface.

To see how changes to the virtual function structure can be handled without impacting clients (other than require that they be recompiled), let's consider the same refactoring scenario described above where a new set of requirements are introduced where the `foo()` functions need to be changed to accept a `Bar` object (represented through its own abstract interface `BarBase`) instead of just an integer, and the new `foo()` functions also need to accept an extra boolean argument. Using the "pure nonmember function interface" idiom, the refactored interface and supporting code would look something like:

```
namespace BlobPack {

class Bar;

class BlobBase {
  // Forward prototypes for nonmember friend functions that
  // will directly call virtual functions
  friend void foo( BlobBase& obj, const Bar &bar, bool flag = false );
  friend void foo( const BlobBase& obj, const Bar &bar, bool flag = false );
protected: // or private:
  // Pure virtual non-public functions to be overridden
  virtual void implNonconstFoo( const Bar &bar, bool flag ) = 0;
  virtual void implFoo( const Bar &bar, bool flag ) const = 0;
};

} // namespace BlobPack
```

where the direct nonmember friend functions now have the implementations:

```
void BlobPack::foo( BlobBase & obj, const Bar &bar, bool flag )
{
  foo.implNonconstFoo(bar,flag);
}

void BlobPack::foo( const BlobBase & obj,  const Bar &bar, bool flag )
{
  foo.implFoo(bar,flag);
}
```

Now, what about all of the clients that relied on the old definition of the `foo()` functions? As was stated above, let's assume that the old meaning and behavior of the `foo()` functions can be retained by using a default implementation of `Bar` called `DefaultBar` and a value of `flag=false` which gives the following nonmember nonfriend functions:

```
void BlobPack::foo( BlobBase & obj, int a )
{
  foo(obj,DefaultBar(a),false);
}

void BlobPack::foo( const BlobBase & obj, int a )
{
  foo(obj,DefaultBar(a),false);
}
```

The above new nonmember nonfriend `foo()` function overloads could then be included in the same file as the other "standard" nonmember functions where `goo()`, for instance, is also declared and defined. After this refactoring, clients that currently use expressions like `foo(obj,0)` now just need to be recompiled and that is it!

As described above, the "pure nonmember function interface" idiom allows for changes in the virtual functions of an abstract interface without requiring any changes to current client code and without compromising the integrity and quality of the refactored interface. While the "pure nonmember function interface" idiom solves the problem of having to refactor client code when an interface changes, it does not address how changes to the virtual function set affects subclasses that implement the interface's virtual functions. The next section describes how changes to an interface's virtual function set affects subclasses. As more refactorings are performed over time, older functions for different types of clients can be partitioned into different files to manage complexity. It is much easier to change a few `#includes` than to change actual source code.

4 The Full Impact of Changing Virtual Functions on Abstract Classes

As mentioned earlier, there are clarifications, changes, and augmentations to requirements for software that beg for changes in the structure and behavior of the virtual functions on an abstract interface. The "pure nonmember function interface" idiom described above takes care of insulating clients from most types of changes to the virtual function set, but how do these changes affect subclasses of the abstract C++ interface that override these virtual functions? There are two main categories of subclasses of abstract C++ interfaces to consider: a) those that are owned and controlled by the library, and b) those that are developed by external users and are out of the control of the library developers. Subclasses can also be classified as i) those that are direct subclasses of the base abstract interface (e.g. such as "Decorator" and "Composite" subclasses), and ii) those that are indirect subclasses of the base abstract interface and don't directly override the top-level virtual functions.

Consider the simplified `BlobBase` interface and some of its subclasses shown in Figure 1. This example is provided to illuminate the issues involved when refactoring the virtual functions in a base class interface. In this example diagram, we show several of the different categories of subclasses mentioned above. The intermediate subclasses `TypeABlobBase` and `TypeBBlobBase` are designed to provide support for implementing the `BlobBase` interface for two different general types of subclasses for more specific types of use cases. For instance, `TypeABlobBase::foo(...)` is an implementation of `BlobBase::foo(...)` that provides most of the needed behavor for "type A" blobs and defers the rest of the more specific behavior to the pure virtual function `TypeABlobBase::typeAFoo(...)` to be implemented by subclasses. These kinds of intermediate type 'a.i' subclasses are very common in object oriented class hierarchies. The type 'a.ii' concrete subclasses `InternalDefaultTypeABlob` and `InternalDefaultTypeBBlob` provided by the library give good default implementations of "type A" and "type B" Blob subclasses. The classes `ExternalTypeABlob` and `ExternalTypeABlob` are type 'b.ii' subcalsses and are implemented by external code developers to satisfy some more specific needs than are provided by the library. The subclass `InternalDecoratorBlob` is provided by the library to support a common type of Blob decorator and represents a type 'a.i' subclass. The subclass `ExternalDecoratorBlob` is a more specialized decorator implementation that is created and lives outside of the control of the "BlobPack" library develoeprs and is therefore a type 'b.i' subclass.

With this example Blob class hierarchy in place, now consider the impact of refactoring the virtual function in a base interface `BloblBase::foo(...)` as described above. Any subclasses that are owned by or accessible by the library developers can be changed at a reasonable cost in most cases. In our example in Figure 1, this means tha all of the classes in the "BlobPack" package can be refactored at a reasonable cost since they are under the "BlockPack" library developer's control. Also, as will be shown, indirect subclasses can largely be insulated from changes to the base abstract interface in many cases if they are derived from well-designed intermediate subclasses which live in the library.

The key to insulating most concrete subclasses from changes to the top-level virtual functions is then is to create a set of appropriate intermediate subclasses, tailored to specific types of use cases, which are owned by the library and define all of the virtual functions on the base class interface and translate these to the more specific use cases with other virtual functions. This is the role of the

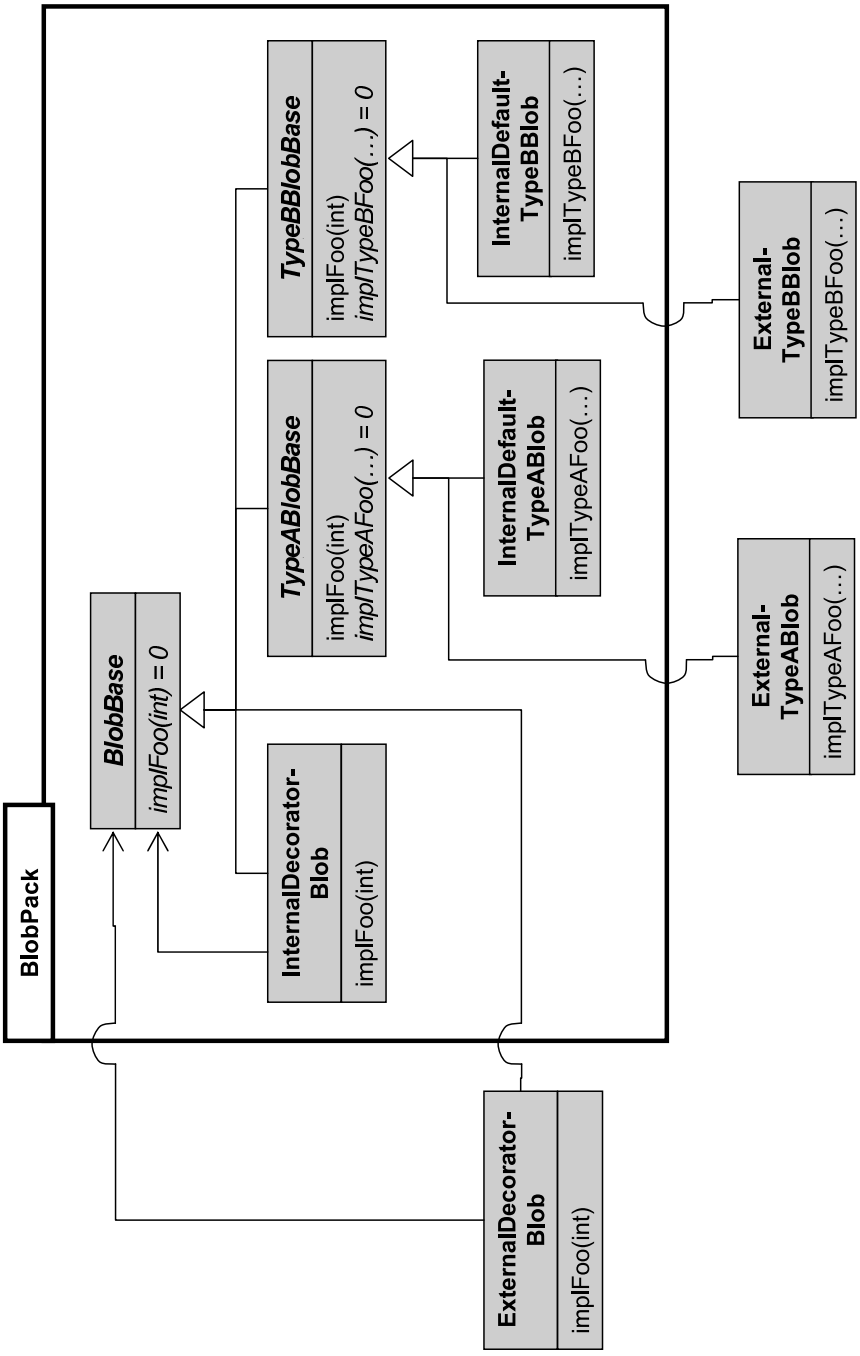

Figure 1. UML class diagram : Blob software before refactoring.

`TypeABlobBase` and `TypeBBlobBase` subclasses shown in our example. Specific categories of clients can then drive from these tailored intermediate subclasses and not have to directly implement any of the virtual functions in the base class interface in most cases.

While the development and use of tailored intermediate subclasses can insulate most types of derived subclasses from changes to the higher-level virtual functions, a remaining stumbling block are those inaccessible external subclasses that directly derive from the base abstract interface (i.e. type 'b.i' subclasses as defined above) such as the `ExternalDecoratorBlob` subclass shown in Figure 1. Examples of these types of direct subclasses would be classic "Composite" or "Decorator" subclasses that use some form of direct delegation on subordinate objects.

Now consider the refactored `BlobBase` interface and subclasses shown in Figure 2. Here, the refactored `TypeABlobBase` and `TypeBBlobBase` intermediate subclasses insulate the concrete subclasses `InternalDefaultTypeABlob`, `InternalDefaultTypeBBlob`, `ExternalTypeABlob`, and `ExternalTypeBBlob` from changes to the base interface and they only need to be recompiled. The situation for the necessarily direct concrete dectorator subclasses `InternalDecoratorBlob` and `ExternalDecoratorBlob` is a little different in that they must be refactored as well in order to remain 100 % general decorator classes. The `InternalDecoratorBlob` subclass is not much of a problem since it is maintained by the "BlobPack" library developers and it's current (nonmember) public interface will likely not be broken due to the refactoring. The problematic subclasses are those external type 'b.i' subclasses that directly derive from the base interface and that are out of the control of the library developers such as the `ExternalDecoratorBlob` subclass. In the most general case, these type 'b.i' subclasses will have to be manually refactored by the external developers. There are strategies where the cost of performing such external refactorings can be lessened but avoiding a refactoring altogether is usually not possible. We will not discuss specific strategies for minimizing the cost of such refactorings any further here.

The goal of this section was to round out the discussion of the full impact of changing the virtual function set in a base class interface in how such a refactoring can be absorbed in the subclasses of the interface and at what cost. As described above, in many cases, intermedate subclasses can insulate many different types of concrete subclasses from significant changes to the base class's virtual functions and can therefore make such refactorings reasonable and affordable. However, in the absents of very sophisticated refactoring tools for C++ (which simply do not exist at the time of this writting and may never exist), we can not fully protect external subclass developers for such refactorings. It should be noted, however, that for many types of more mature class libraries, that there should be relatively few examples of direct external "Composite" or "Decorator" subclasses such as `ExternalDecoratorBlob` which are the most significant problem from a refactoring standpoint. Note however, that modern Agile software engineering methodologies really mandate the need for refactoring and therefore we must actively plan for change and the refactorings that are needed to manage complexity.

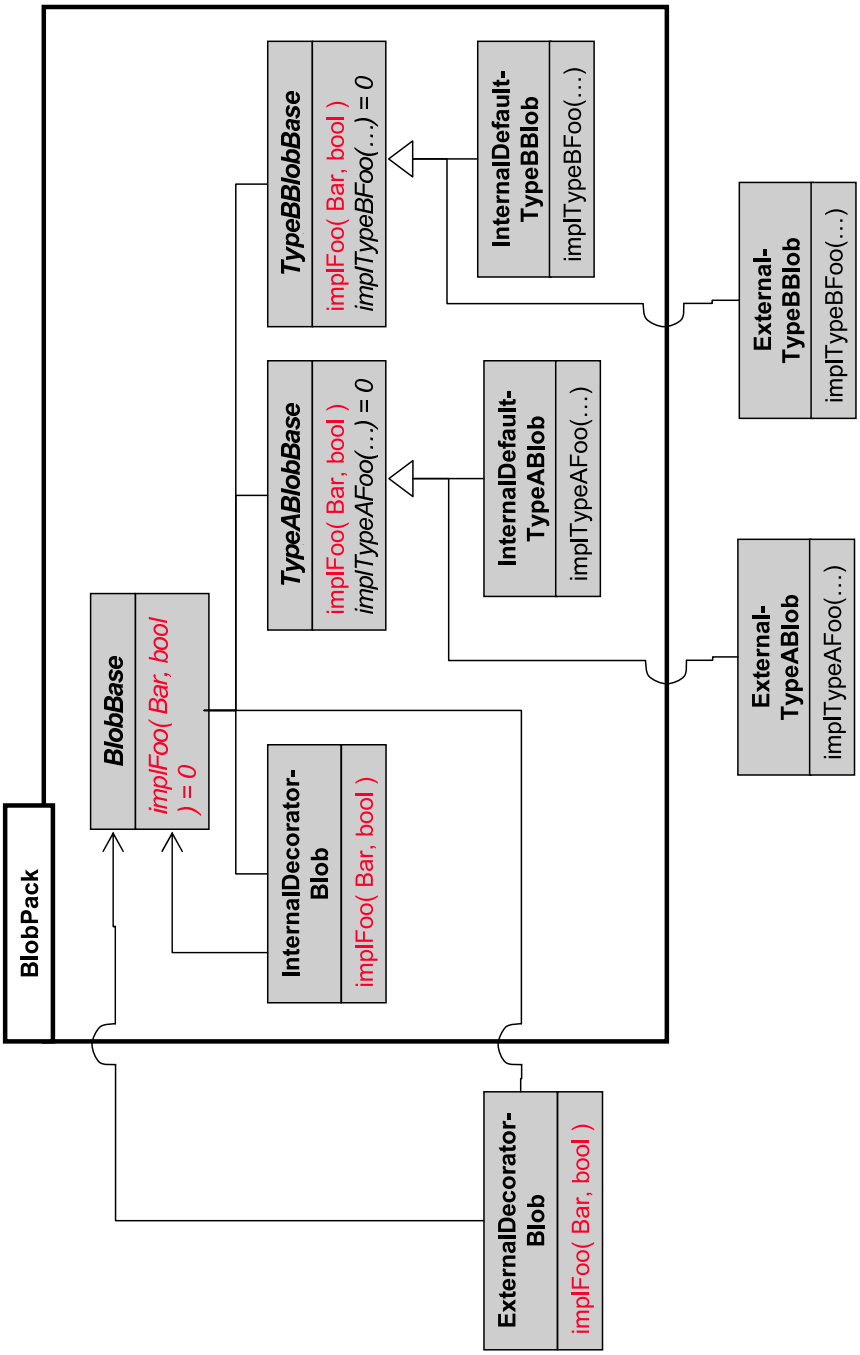

Figure 2. UML class diagram : Blob software after refactoring.

5 The Nonmember Constructor Function Idiom for Concrete Classes

While the key focus of the "pure nonmember function interface" idiom described here is in connection with abstract C++ classes, it is also applicable to concrete classes as well. In fact, the discussion in [4, Item 23] primarily deals with nonmember functions used with concrete classes. In the context of abstract classes, the "pure nonmember function interface" idiom helps to encapsulate the virtual functions. In the context of concrete classes, the "pure nonmember function interface" idiom helps to better encapsulate private members. For most practical purposes, clients do not really need to differentate between concrete and abstract classes when reference semantics are being used for the objects in question.

However, the one major difference between an abstract class and a concrete class is that you can construct an object for a concrete class but not for an abstract class (unless you construct an object from a concrete subclass of the abstract class of course in which case we are back to talking about concrete classes). Therefore, constructors are one of the major details that differentiate concrete classes and abstract classes. There are many good arguments for defining and having users call nonmember constructor functions as apposed to directly calling member constructor functions.

Nonmeber constructor functions can be defined to create and return objects by value (i.e. to construct objects on the stack) or to return (smart) pointers to dynamically allocated objects. Both types of nonmember constructor functions are useful in different contexts. Typically, small objects that are manipulated with value semantics will be given nonmember constructor functions that return the created objects by value (and typically use the return-value optimization to achieve good performance). On the other hand, larger objects that are manipulated through reference or pointer sementics will be given nonmember constructor functions that return (smart) pointers to allocated objects. Experienced C++ developers typically manage these dynamically allocated objects with some type of smart pointer class [5, Item 13]. Here, we will refer to a smart pointer class that exists in the Trilinos Teuchos package called `Teuchos::RCP` [1] which uses a nonember constructor function `Teuchos::rcp(rawPtr)`. Any other high quality smart pointer class such `boost::shared_ptr` could be used as well as part of this discussion.

The nommeber constructor function idom that creates dynamically allocated objects takes the form:

```
class Widget {
  // Declaration of nonmember constructor function
  friend RCP<Widget> widget();
public:
  // Public interface
  void display(std::ostream&);
private: // or protected
  // Non-public constructor
  Widget();
};

// Definition of nonmember constructor function
inline
RCP<Widget> widget()
{
  return rcp(new Widget());
```

```
}
```

The above nonmember constructor function ensures that every `Widget` object that is created is dynamcially allocated and wrapped in an `RCP` object. The convention estabilished here is to typically use a function name that starts with a lower case letter (e.g. `widget(...)`) which corresponds to the class name which begins with an upper-case letter (e.g. `Widget`). This naming convention for nonmeber constructor functions is short and makes for well documented client code.

Using this nonmember constructor function, client code can then create and use `Widget` objects as follows:

```
RCP<Widget> w = widget();
w->display(std:cout);
```

There are sevaral advantages to the nonmember constructor function idiom applied to dynamcially allocated objects over raw calls to `new Type(...)`:

- *Nonmember constructor functions can be overloaded or can be given different names to make the result of the construction more clear.* While member constructor functions must always be overloaded (i.e. taking different argument lists) to handle different construction states, nonmember constructor functions can actually use different names to make them more clear. For example, the nonmeber constructor functions `ovalShape( height, width )` and `roundShape( diameter )` may be more clear than using the direct overloaded constructors `Shape( height, width )` and `Shape( diameter, diameter )` or `Shape( diameter )`.
- *Nonmember constructor functions can directly return smart pointer wrapped objects.* This avoids having to call an extra function to create the wrapped pointer. For example, compare:

  ```
  RCP<Widget> w = Teuchos::rcp(new Widget(...));
  ```

  to

  ```
  RCP<Widget> w = widget(...);
  ```

- *The template argument(s) in a templated nonmember constructor function can be figured out automatically by the compiler in many cases.* This avoids having to provide template arguments when the types can be determined from the formal arguments. If you allocate a templated class directly, you must always give the template arguments explicitly but this many not be required when using the nonmember function which allows you to replace:

  ```
   new GenericWidget<VeryLongAndUglyInputType>(input)
  ```

  with

  ```
  genericWidget(input)
  ```

in many situations.

- *Nonmember constructor functions help to avoid memory leaks when exceptions are thrown*. For example, the statement:

```
WidgetC wc( rcp(new WidgetA(...)), rcp(new WidgetB(...)) );
```

might create a memory leak if an exception is thrown by one of the constructors (see [5, Item 13]). The reason that a memory leak might occur is that a C++ compiler is allowed to evaluate `new WidgetA(...)` and `new WidgetB(...)` before calling the `rcp(...)` functions. If the constructor `WidgetB(...)` throws an exception after the `WidgetA(...)` constructor has been invoked but before the `RCP` object wrapping the `WidgetA` object is constructed, then the memory created by `new WidgetA(...)` will never be reclaimed. If you use nonmember constructor functions, other other hand, then you would have:

```
WidgetC wc( widgetA(...), widgetB(...) );
```

and no memory leaks will be created if an exception is thrown since each argument is returned as a fully formed `RCP` object which will clean up memory if any exception is thrown.

- *Nonmember constructor functions increase the encapsulation of your classes by not requiring direct access to private data (see [5, Item 44]) and allowing less duplication of default member values*. You can provide many different special case constructors without cluttering up the class which can just have a general purpose constructor and or a set of post-construction initialization functions. This also helps to improve maintaince by only having a single default constructor that sets default member values only once.

- *Nonmember constructor functions that return smart pointer wrapped objects avoids raw pointers at the application programming level*. Raw pointers are the root cause of segfaults and memory leaks in C++ and as a C++ community we should move to the development of an environment where we can avoid them in high-level code (see “Hide pointer operations” in [3, Section 7.1]).

- *Nonmember constructor functions allow a single simply written class to create const and non-const encapsulations of other objects*. For example, the following class and nonmember constructor functions allow clients to safely wrap contained objects:

```
class Wrapper {
  friend RCP<Wrapper> nonconstWrapper(const RCP<Contained> &contained);
  friend RCP<const Wrapper> wrapper(const RCP<const Contained> &contained);
public:
  RCP<Wrapper> getContained() { return contained_; }
  RCP<const Wrapper> getContained() const { return contained_; }
private:
  RCP<Contained> contained_;
  Wrapper(const RCP<Contained> &contained)
    {  contained_ = contained; }
};

RCP<Wrapper> nonconstWrapper(const RCP<Contained> &contained)
```

```
{
  return rcp(new Wrapper(contained));
}

RCP<const Wrapper> wrapper(const RCP<const Contained> &contained)
{
  return rcp(new Wrapper(rcp_const_cast<Contained>(contained)));
  // Note: This const_cast is safe since the const Wrapper
  // object that is returned will only give clients an RCP to a
  // const Contained object.  However, this assumes that the Wrapper
  // class will be written carefully to protect const of the wrapped
  // object in non-const member functions.
}
```

There are many other alternative ways to develop wrapper classes that protect const of contained objects. However, a more detained discussion about how to write wrapper classes that protect the const-ness of contained objects in a maintainable and safe way is beyond the scope of the “pure nommeber function interface” idiom which is our main focus here.

6 Summary

Here we have presented a sort of composite C++ idiom called the "pure nonmember function interface" idiom which is composed out of two other idioms, the non-virtual interface (NVI) idiom [5, Item 39] and the "nonmember nonfriend function" idiom [5, Item 44]. We argue that the proposed "pure nonmember function interface" idiom is the logical union of these two other idioms when issues of code evolution and refactoring are considered in an environment where refactoring tools are absent or impractical to use.

In summary, the "nonmember nonfriend function" idiom:

- results in maximum code encapsulation,
- provides a uniform (nonmmeber function) client interface,
- avoids problems with overloaded virtual functions,
- avoids problems with default parameter values in virtual functions,
- insulates clients from refactorings to the virtual function structure of abstract interfaces (and therefore makes such refactorings reasonable and affordable), and
- preserves the minimality and integrity of abstract interface even after numerious changes in requirements and subsequent refactorings.

While there are many advantages to the "pure nonmember function interface" idiom, it is not without some cost. Some of the potential disadvantages of a pure nonmember function interface are:

- Calling a nonmember function may require explicit namespace qualification (or a using declaration) and/or explicit template arguments in order to get the right function to be called which is almost never needed for a non-template member function call.
- The implementations of nonmember friend functions are a little more verbose than member functions. For example, one has to explicitly qualify the object argument (i.e. `blob`) instead of an implicit `this->`. Some programmers may consider it an advantage to be more explicit however and some languages (e.g. Python and Perl) always require explicit qualification to the object in member function implementations.
- Documentation may become more complicated to develop and access since it is not clear where to document the behavior of a function. Should the documentation be in the public nonmember function (better for the client) or should it be in the declaration of the protected/private virtual function (better for subclass implementors)? Clearly some documentation guidelines need to be worked out in order to address these issues.

Since there are some disadvantages to developing and using a pure nonmember function interface for an abstract or concrete class, one should not automatically choose it over a more common member function interface. If a particular class is not widely used, is not widely accessible to

clients, and/or is unlikely to change, then developing a pure nonmember function interface may be overkill and not worth the (all be it small) extra work. However, in the face of uncertainty one should lean toward using the pure nonmember function interface idiom since it allows great freedom in refactoring code with minimal impact on clients.

References

[1] R. A. Bartlett. Teuchos::RefCountPtr : An intruduction to the Trilinos smart reference-counted pointer class for (almost) automatic dynamic memory management in C++. Technical report SAND04-3268, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, 2004.

[2] S. Dewhurst. *C++ Gotchas: Avoiding Common Problems in Coding and Design*. Addison Wesley, 2003.

[3] S. McConnell. *Code Complete: 2nd Edition*. Microsoft Press, 2004.

[4] S. Meyers. *Effective C++: Third Edition*. Addison Wesley, 2005.

[5] H. Sutter and A. Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines and Best Preactices*. Addison Wesley, 2005.

A Member verses Nonmember Functions in C++

It is instructive to consider the implications of member and nonmember functions in C++ and how these differ with other object-oriented languages. Specifically, we will consider three different object-oriented languages: C++, Java, and Python. We can broadly classify languages as more strongly typed and less strongly typed. In a stronger typed language such as C++ and Java, the semantics of an object are determined at compile time. For example, in C++ and Java, the set of member functions that are callable on a class object like `BlobBase` is known before the program even starts to execute. In fact, all of the member functions on a C++ and Java class must be declared in a single header file within a single class declaration (e.g. `class BlobBase { ... };`)[1]. In Python, however, a member function can be added to an object at any time at runtime while a Python program is executing and therefore calling an operation on an object through member function results in no lack of flexibility like it does in C++. As for member and nonmember functions, Python and C++ can have nonmember functions while Java can not. Therefore, when one talks about member and nonmember functions, conventions used in Java are mostly meaningless.

Okay, so C++ and Python both allow member and nonmember functions. So why is there not a "nonfriend nonmember function" idiom for Python like there is for C++? To begin to answer this question, first consider that Python does not have any true encapsulation. If any piece of Python code wants to grab the private data for a object, they can just do it. Therefore, there are no features like `private` and `friend` in a language like Python without encapsulation. Another difference between C++ and Python is that in Python, any code can add a new member function to any Python object at any time dynamically at runtime. Therefore, Python code that uses the member function syntax `obj.foo()` to invoke some operation is no less flexible then using nonmember function syntax like `foo(obj)`. Therefore, since many programmers think that object orientation means calling member functions such as `obj.foo()`, there is no disadvantage to allowing them to do so in a language like Python.

The situation for C++ is quite different. By having the client insist on using member function syntax such as `obj.foo()`, one is already placing several restrictions on how that operation is implemented and gets invoked. In particular, having a client use a member function (e.g. `obj.foo()`) to invoke an operation on a C++ object requires that:

1. The function must be declared as a member function on the class in a single declaration (i.e. within `class ClassName { ... };`) in a single file. Sure the function's definition can can be redefined by some subclass but the object itself must determine how to perform the operation.

2. The function must be set at compile time and can not changed dynamically.

3. A member function can access private and producted members and therefore errodes encapsulation, even if none of the non-public members are accessed initially. If encapsulation by gentleman's agreement is sufficient for you then you are working with the wrong programming language and perhaps Perl or Python might be more to your liking.

[1] Some of the functions callable on a class object can of course be defined in base classes but that does not change the point of the discussion here.

On the other hand, using the nonmember function syntax in C++ `foo(obj)` opens the door for a great many different possibilities for how the operation gets invoked. Some of the possibilities for the implementation of a nonmember function are:

1. The nonmember function could be a direct call to the member function on an object (e.g. `foo(obj)` means the same thing as `obj.foo()`). Therefore, the use of the nonmember function is as least a flexible as directly calling a member function. In addition, the nonmember function syntax almost always uses less ASCII characters. For example, writing `foo(obj)` only takes eight ASCII characters to write, while `obj.foo()` takes nine ASCII characters. This is a trival difference but the point is made.
2. The nonmember function could perform a few different types of tasks before calling the member function on some object. With this approach, we can add different layers of capabilities in a distributed way.
3. The nonmember function could actually be involved in a sophisticated type of multi-dispatch system where the exact operation to call would be determined by the traits of the objects involved at runtime. For example, a nonmember function using multi-dispatch like `foo(objA,objB)` could call one of a number of different operations based on the types or properties of the objects `objA` and `objB`.

In addition, nonmember functions that are declared in the same namespace as the types of the objects in their argument lists will be automatically looked up and called without requiring namespace qualification (this is know as Argument Dependent Lookup (ADL))[2].

But does not object-oriented programming mean that all operations on objects are called through member functions? Some programmings (and users) might currently expect and even insist on this but many current object-oriented C++ experts would not agree. For example, Scott Meyers in [4, Item 23] states:

> Object-oriented principles dictate that data and the functions that operate on them should be bundled together, and that suggests that the member function is a better choice. Unfortunately, this suggestion is incorrect. It's based on a misunderstanding of what object-orientation means.

If nonmember functions are so less flexible than member functions, then why even bother using member functions at all? Why not just use nonmember friend functions in the place of member functions as is advocated in the "pure nonmember function interface" idiom for all C++ classes? Well, there are a few reasons that you would want to just use member functions declared and defined within the class. First, some functions like the assignment operator are required to be member functions as are a few other operator functions. In these cases, you have not choice but the use public member functions. Second, if the C++ classes you are writing are expected to be very stable or if changes to these classes can be easily propagated to all clients using the class, then using member functions is attractive since it just involves less typing, uses less ASCII characters and may be easier to follow.

[2] In some types of templated code, calling a nonmember function without namespace qualification does not always work but there are usually various work arounds to make this manageable.

Friends and member functions are a fundamental part of the implementation of true encapsulation in C++. They help to provide a system by which we can enforce encapsulation while still allowing programs to be written to do things. By requiring that friends and member functions be declared right in a class declaration, one can easily enumerate and track down all code that has access to private members which facilitates safe code refactorings which is the whole reason for encapsulation in the first place[3]. Therefore, we should think about friends and member functions in C++ as the basic tools for implementing encapsulation and not primarily as interfaces for clients. By adopting the "pure nonmember function interface" idiom, we relieve clients from having to worry about whether a function needs to acces encapsulated data (in which case it needs to be a friend) or not (in which case it does not need to be a friend), which is not really any of their business anyway.

Other than for a bit of laziness and personal preference for writing `obj.foo()` as apposed to `foo(obj)`, there really is not a strong argument against using the "pure nonmember function interface" idiom.

Since Java does not have the concept of nonmember functions, one could use a separate static class to declare these non-privileged functions and one could therefore have the same benefits as in C++ when using the "pure nonmember function interface" idiom. However, since there would no longer be any namespace lookup, clients would have to explicitly qualify the static class's name when calling the functions. Also, there are many high-quality refactoring tools for Java that make it much easier to propagate refactorings to client code. Therefore, the lack of good refactoring tools, the convenience of automatic namespace lookup, and other issues make the "nonmember function interface" more attractive for C++ than for Java or other object-oriented languages.

[3] If code never changed then why would anyone care about encapulation?

B Relationship between the Pure Nonmember Function Interface Idiom and Handle Classes in C++

A handle class is a concrete class that is meant to mimic the object that it wraps through a pointer to that object. Typically, a handle class is used to wrap an object that uses reference or pointer semantics, such as objects represented through an abstract interface. For example, a simple handle class for a `BlobBase` object might look like:

```
namespace BlobPack {

class Blob {
public:
  Blob( RCP<BlobBase> &blob ) : blob_(blob) {}
  void foo( const Bar &bar, bool flag ) { foo(*blob_,bar,flag); }
  void foo( const Bar &bar, bool flag ) const { foo(*blob_,bar,flag); }
private:
  RCP<BlobBase> blob_;
};

} // namespace BlobPack
```

The above handle class allows for multiple handle objects to reference the same underlying `BlobBase` object and a handle object can be reassigned after constructed. There are many nuances to consider when one develops a handle class. However, a detailed treatment of handle classes is beyond the scope of this discussion.

The “pure nonmember function interface” idiom and the “handle” idiom are similar in that both provide a layer of indirection between clients and the specification of the virtual function set which insulates clients from changes to the abstract interface.

However, the “pure nonmember function interface” idiom offers some potential benefits over a strict handle interface that uses member functions on the handle class:

1. *Nonmember functions can be split into multiple files while member functions on a handle class can not*: Splitting capabilities across multiple files allows for greater segmentation in the capability set for a set of objects, it reduces unnecessary dependences and allows for growth in the set of capabilities without impacting current clients (or even require them to be recompiled) (see [4, Item 23] a specific discussion of this issue). However, if most functions defined on the handle object are expressed as nonmember functions, then this would allow for the same flexibility to segment capabilities and this problem is eliminated.
2. *Nonmember functions do not suffer from some of the more confusing aspects of handle classes*. For example, a semantics of a handle class must be carefully spelled out as to how objects can be shared, what the copy constructor and assignment operator functions do, and other such details. Nonmember functions do not have to deal with any of these issues.

On the other hand, if extra state needs to be added to the object in order to perform extended operations, then using a handle class might be an attractive approach. However, if the handle class

allows multiple handle objects or other clients to point to the same underlying object, then allowing for extra data in the handle object can be risky since that data can easily be invalidated in many cases.

There are cases where the syntactic advantages of using a handle class are compelling and therefore handles should be used in these cases. Even if a handle class is to be used, most operations performed with handle objects should be implemented as nonmember functions for reasons of avoiding bloat of the handle class and allowing for incremental and distributed capabilities. Of course, if one allows for both the use of member functions on the handle class and nonmember functions, then this brings up all of the same issues as it did for the interface class itself. However, since a handle class should almost never be used as a means of interoperability, one can be more lenient about what functions are added as member functions and therefore handle classes used as a convenience can tolerate refactorings better and tolerate more bloat. Additionally, since the member functions on a handle class should never need privileged access to the underlying object, there will never need to change the handle class's interface; only augmentations will be required. Therefore, the handle idiom and the pure nommember function idiom can be used together and complement each other.

DISTRIBUTION:

2	MS 9018	Central Technical Files, 8944
2	MS 0899	Technical Library, 4536

Sandia National Laboratories