Libraries that aren't perfect

Dealing with third party code

Charles Weir
© Charles Weir, November 2004

Introduction

Almost every C++ project needs to involve third-party code. Graphics, operating system access, communications, application-specific processing - there are many libraries and frameworks available. And we need to use them; it's almost never worth our while to reinvent their functionality.

Most of the more familiar libraries are straightforward to use from the project manager's point of view. The library arrives with more or less documentation, and we develop our code to work to its defined interfaces. If there are any deficiencies in the library we tend to work around them and maybe report them as bugs to the vendors. Later we receive further releases: often the interfaces remain unchanged and all we need to do is to recompile everything; sometimes we may need to change our code to satisfy new interface declarations. Frustrating as the latter case often is, there's one thing that makes it easy to manage: the changes we need to make are always in our code, not in the library (see Figure 1).

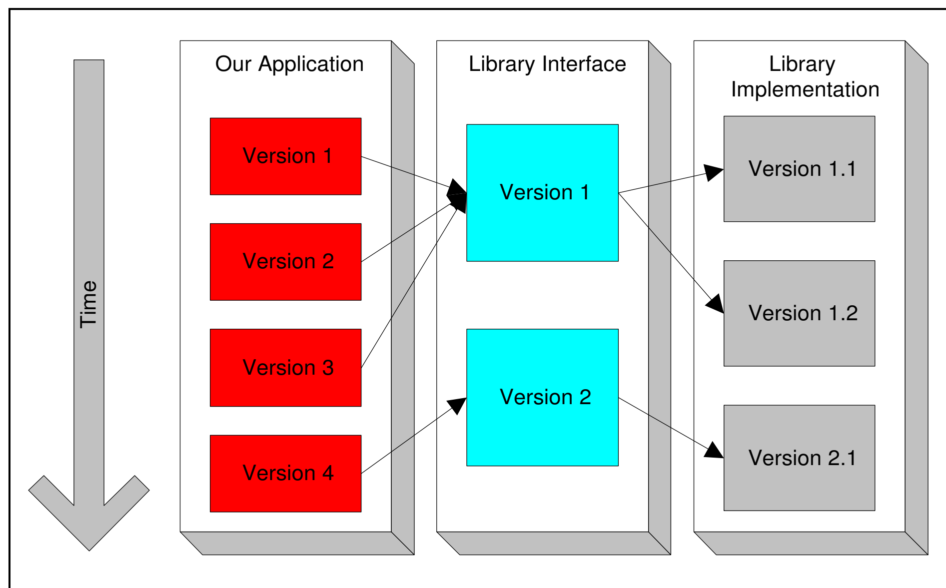

Figure 1: A well-designed library

Virtually all of the mass-market libraries and frameworks work in this way: Microsoft's MFC, RogueWave's toolkits, ObjectSystems SystemsToolkit, for example. They have a mass market; they are very heavily tested and their suppliers receive masses of feedback from the users on all the possible requirements that the libraries may need to support.

However there is a completely different style of interfacing to third-party code. Sometimes it's simply impossible to get the functionality we require without changing the third-party code in some way. Perhaps the third-party architects didn't anticipate the particular changes we need; perhaps the third-party implementers were sloppy in some part of the implementation. In any case we need to change the code.

This situation is particularly common where the third-party framework provides the major functionality of the system. There are now many large systems written in C++, which claim to be frameworks, to allow us to tailor the system to our own situation - by writing C++ code. For

example there are frameworks for banking systems, accounting systems and stock control systems, to name but three areas. In each case the core functionality is the same for every system, but every purchaser has their own additional functionality they want to add. Some changes are easier to add than others - but surprisingly often there will be some need somewhere to change the third-party code.

Of course if the vendor doesn't release the source code, we're in trouble. There's no way to make our changes and we have to admit defeat (or bully the vendor until they do what we want). Almost always, though, the vendor will release the source code - if only as a form of documentation!

However there's a major problem. The third-party vendor will make further releases, and we will want the enhancements they contain (see Figure 2). How should we manage the changes we make to the third-part C++ code so we don't end up with a fantastic amount of work every time there's a new release?

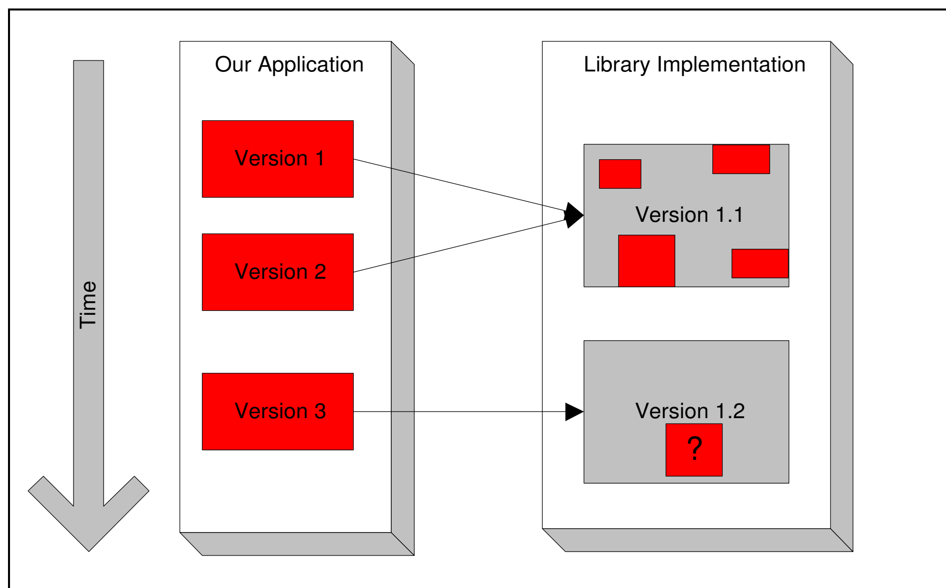

Figure 2: What happens with an imperfect library?

In this article I'll examine this question, and discuss:

- The major principles for managing third-party code
- Tools and techniques for locating changes to simplify the job of keeping everything up to date.
- Three of the most common techniques for modifying third-party classes
- Six guidelines for framework designers to make their code easier to extend

Throughout this document I shall use the phrase '*framework code*' to refer to the code we want to change as little as possible (the code supplied by a third-party). The phrase *'our code'* will mean our own additions to this.

The Principles

There are four major principles for managing our changes to third-party code. The principles are simple and obvious; but it's surprising how difficult they are to follow in practice. I'll examine each in turn.

Principle 1: Keep our code separate from their code

This principle is common sense: if our code is separate then changes to the library code will happen independently from our code.

In C++ a unit of code is a file. Keeping code separate means keeping it in a different file. So this rule means that:

> *Any significant functionality we need to write should be in a file owned by ourselves, not by the library.*

This is not trivial to achieve. It is often easier to hack a few lines of C++ into the third-party code rather than to design a way of placing the functionality in a separate file. However the latter requires much less work in the long run.

Principle 2: Minimise the number of changes in their code

The reason for this is obvious; every change represents more work for us whenever we receive a new release of the third-party software. However the consequences are more subtle. For example, this means we should avoid changes that involve a search-and-replace throughout the code. A single **`#define`** in a header file is better than changing a name throughout all the source code.

Principle 3: Mark any changes in their code so they are easy to locate

This rule, like the first, is more honoured in the breach than the observance. Few programmers do it. But like the first rule it's well worth following.

There are two simple techniques for marking code. One is to use a standard tag in a comment. For example if our development team is called XYZ, we might mark a line we've added as follows:

```
CallInhouseProcessing();      // added XYZ
```

A useful technique where the change is simply to add a common word is to use the pre-processor to create a unique name. For example, a common addition is to make functions virtual so that we can modify their behaviour in a derived version of the class:

```
#define XYZ_VIRTUAL virtual
```

Then the word itself acts as a marker, and we can search for it when we incorporate new versions of the framework, as shown in the example below:

```
class Invoice {
// ...
XYZ_VIRTUAL void CalculateNetValue();
// ...
};
```

Principle 4: Use code merging tools to incorporate changes

No matter how many of the previous mechanisms we use to mark and reduce the amount of changes to the framework code, there are still likely to be changes to carry forward to new versions of the framework. Re-inserting these changes is likely to be a manual task; it's difficult to automate it. However here are several techniques that can help:

- Using a checksum on every file.
  It is useful to have a checksum on every framework file to provide a simple way to see if it has changed since the last release. If it hasn't changed then we can keep any existing modifications to it. You may need to write a special checksum program that ignores release identification information generated by the source code control system.

- Use a smart differences tool.
  Given that a file has changed since the last release, provided that the changes are small we can use one of the standard source code control 'merging' tools to incorporate our changes from our version into the new version. There are some very good graphical tools available - see what your source code environment provides.

- Use source code control tools
  In most projects we will be using a source code control tool to manage the changes to the our own code. We should use this source code control on the library code too. Almost all of these tools support 'branching paths' for each file, allowing two sets of modifications to coexist. We can incorporate our changes to each release of the library code as one branch, and incorporate the next release of the library as another branch. Then we can use the differences tools as above to merge the two branches.

If the change is too complicated for these simple merging techniques then we're reduced to using entirely manual techniques to re-implement our former changes in the new system. If that happens then the techniques above will still make this as painless as possible.

Techniques for extending code

There are several C++ techniques we can use to modify library code without breaking the first two principles (Principle 1: Keep our code separate from their code, and Principle 2: Minimise the number of changes in their code).

About relationships

To discuss these techniques, I'll start by distinguishing two kinds of relationship. '*Using*' relationships are associations between two objects; the *user* instance knows about the *used* one, but not vice versa. We normally implement these relationships in C++ using pointers or embedded instances. C++ represents '*Inheritance*' relationships using derivation; typically the base class has no knowledge of the derived class.

We'll also have to distinguish two types of object: *object types* are sometimes referred to as 'business objects'. They represent major entities in the business processing: examples might be a Deal (in a banking application), an Invoice (in an accounting application) or a sensor (in a machine control application). *Value types* represent data items that can be attributes of the business objects: examples might be AmountOfMoney, Date, String, or Percentage[1].

Changing object types using derivation

The 'Gang of Four' book (reference 1) describes several patterns that are effective when we wish to modify the behaviour of object types provided by the library code. In particular, the *adapter* pattern changes the behaviour of the class used so that it can fit within a different set of code.

Perhaps the most familiar 'object-oriented' technique for changing the behaviour of a class is to derive from it. Then any framework code working with pointers or references to the original class will work also with instances of our derived version.

This is suitable for changing classes used as object types since instances of object types are almost always used as pointers or references. We can add to an existing class by deriving from it and overriding functions.

There are several potential problems here, and a variety of possible solutions:

- Getting the correct instance created

  Usually the framework code will create a class using the class name:

```
Deal* newDeal = new Deal( someParameters );
```

[1] For the rigorously minded, the difference in C++ is that value types are identified by their contents; object types are identified by their address. For example a string is a value type: two strings are identical if they have the same contents regardless of their addresses.

We want to change the type of the deal produced to our own version. Once we've done that all the code using new deal will use our versions of all the functions, and we'll get the behaviour we want.

Ideally we want to persuade the framework vendor to avoid any direct mention of the class name. The simplest approach is to use an implementation pattern (see reference 1) called the 'Factory Method'. This is simply a function that takes the parameters passed to the constructor and contains the code above. If we want to change the name of the class created, we need only change the one line in the factory method[2].

If the framework doesn't use that pattern, our only alternative is to modify the framework code and to mark appropriately:

```
Deal* newDeal = new XYZDeal( someParameters );   // XYZ
```

- Non-virtual functions

  For this approach to work we have to be able to override and replace functions. If they're not virtual this will not work correctly. This may mean adding the keyword **`VIRTUAL`** to the base class definition (see 'Principle 3: Mark any changes in their code so they are easy to locate').

- Consistency with the old code

  It is important when we override a member function, that our version calls the overridden library version if at all possible. This ensures that if the library version changes, the changes are seen by the new version of the code. If however we are completely replacing the library implementation of a function, then sometimes we may even need to call the function that *it* derives from: the grandparent version rather than the parent.

- Large functions.

  If we need to make a small change in the behaviour of a large function we may have a problem. If we copy the entire function there is a danger that its implementation may change in a later version; the copy will not keep track. There's no easy solution, but it may sometimes be better to modify the library function to call other small or null methods and override those instead. This is the Template Method pattern (reference 1).

- Copying and assignment

  We have to be careful if we need to write a copy constructor or assignment operator for our new class. Since the library code knows only about the original class, it will pass references to this; our implementation must cast these accordingly. The signatures might be as follows:

```
XYZDeal::XYZDeal( const Deal& );    // Parameter is actually XYZDeal
Deal& XYZDeal::operator=( const Deal& ); // nb. Base version is virtual
```

  Of course it's much easier if the library class defines a virtual function that allocates and returns a copy of the current instance; in that case all we must do is to override it. This function is often called '**`clone`**', as follows:

```
Deal* Deal::clone();
```

---

[2]An even more powerful approach is the (parameterised) Abstract Factory pattern (reference 1). A well-designed Abstract Factory will avoid any need for us to modify the framework code at all.

- Business Objects used by value.

  There's a major problem if the framework uses one of the business objects by value; for example by embedding an instance within another business object. If this happens, the derivation approach will not work. I'll explore some other approaches later in this article.

Using the pre-processor and linker to replace an implementation

This approach is suitable when we need to replace just the implementation of one or more member functions of a class. We add pre-processor directives to comment out the parts of the implementation we want to change (or, if it is an entire file, we don't compile it). In a separate file of our own code, we add the new implementations of the parts we've changed and then link the result with the rest of the system. Figure 3 shows an example.

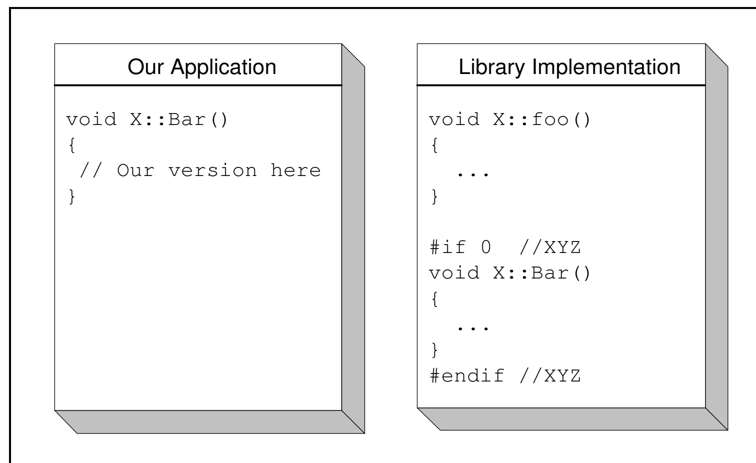

Figure 3: Using the pre-processor and linker

Replacing classes using the pre-processor

This is suitable for replacing or changing classes used as value types. It is also suitable for object types that have embedded instances within other classes. In both cases the class name will appear throughout the framework code, so we cannot use derivation to modify the behaviour. Suppose for example we want to change the implementation of a class **`String`** throughout the framework. Everywhere we'll find code that names the class directly, such as:

```
class x {
// Variable and member function declarations...
String y;
String x;
};
```

Clearly the derivation approach (See 'Changing object types using derivation') will not work, since the compiler has the type **`String`** hard-coded throughout the framework. We cannot afford to use the editor to change every occurrence of **`String`** in the framework to, say, **`XYZString`**; it would run counter to our Principle 2: Minimise the number of changes in their code. Equally, simply replacing the framework header file, **`String.h`**, would also mean changing a large amount of code, and would prevent us from using the framework implementation of **`String`** in our own implementation.

What we want, instead, is to have the compiler *automatically* change every reference to **`String`** to a reference to our own implementation **`XYZString`**. Also we also want our implementation to have access to the framework implementation of the original **`String`** class, so we can minimise the amount of extra code we have to write.

Yes, there is a technique to achieve this. And it requires minimal additions to the existing framework code. This technique uses the *pre-processor* to make the replacement of **`String`** with **`XYZString`** during compilation.

For this to work, there needs to be a *global* header file which is included by every piece of code that refers to **`String`**. The file **`String.h`** or its equivalent would not be suitable; some files may only have forward references to **`String`** rather than pulling in the entire header file:

```
class String;
```

Into the global header file we add a single **`#define`** statement. In the class definition file, **`String.h`** we add a few additional pre-processor statements.

Of course our new implementation, **`XYZString`**, must provide the same public member functions as the class **`String`** does. Ideally it will include an instance of the old one and use it to provide most of the functionality but we may choose instead to re-implement everything.

Example of the pre-processor technique

I can explain the technique using a simple example. The following is a complete example of a possible change to a class **`String`**. It replaces a simple implementation with one that uses reference counting. Such a change might well be appropriate to give a particular performance improvement for a specific project. The code examples give a complete implementation, but omit details irrelevant to the technique.

The new implementation uses an instance of the existing **`String`** class. It forwards function calls to it, making additions or changes as necessary. In the global header file, we add a single line:

```
#define String XYZString    // XYZ
```

The library header file **`String.hxx`** becomes as follows. The only changes we've added are those marked with the **`XYZ`** comment:

```
#undef String               // XYZ

class String
{
public:
    String(char* s = "")        { alloc( s ); }
    String( const String& s )   { alloc( s.rep ); }
    ~String()                   { delete [] rep; }
    char& operator[] (int i)    { return rep[i]; };
    int length() const          { return strlen( rep ); }
    operator const char*() const { return rep; }

    String& operator=( const String& s );
private:
    char* rep;
    void alloc( const char* s );
};

#include <XYZString.hxx>    // XYZ
#define String XYZString    // XYZ
```

We need to add one extra line to the code file for the library implementation:

```
#undef String               // XYZ

String& String::operator=( const String& s )  {
    if (&s != this) { delete rep; alloc( s.rep ); }
    return *this;
}

void String::alloc( const char* s ) {
    rep = new char[strlen(s)+1]; strcpy( rep, s );
}
```

The new file **`XYZString.hxx`** now contains only our own code; since it uses the library implementation internally it will not change significantly even if the implementation of **`String`** changes:

```
class XYZString  {  private:
    class Rep  {
        friend class XYZString;
        Rep( char* s_ ) : s( s_ ), refCount( 1 ) {}
        String s;
        unsigned refCount;
    };
public:
    XYZString( char* s = "" )        : rep( new Rep( s ) ) {}
    XYZString( const XYZString& s ) : rep( s.rep ) {
                                    rep->refCount++; }
    XYZString& operator=( const XYZString& s );
    ~XYZString() { decCount(); }
    char operator[](int i) const { return rep->s[i]; };
    char& operator[](int i) { copyOnWrite(); return rep->s[i]; };
    int length() const { return rep->s.length(); }
    operator const char*() const { return rep->s; }
private:
    Rep* rep;
    void copyOnWrite();
    void decCount() { if (--rep->refCount == 0) delete rep; }
};
```

The corresponding code file requires no specific pre-processor constructs:

```
XYZString& XYZString::operator=( const XYZString& s ) {
    if (rep != s.rep)
        { decCount();  rep = new Rep( (char*)(const char*)s.rep->s ); }
    return *this;
}
void XYZString::copyOnWrite()
{   if (rep->refCount == 1) return;
    rep->refCount--;
    rep = new Rep( *rep );
    rep->refCount = 1;
}
```

And here is an example main program that uses it.

```
#include <iostream.h>
#include <GlobalHeaderFile.hxx>   // Global header file.
#include <String.h>

int main()
{
    String x( "Hello world\n" );
    cout << x << endl;
    return 0;
}
```

Guidelines for framework designers

We can summarise the problems I've been discussing as a set of guidelines for framework designers. The article has already discussed the reasons for each in some detail. The first rule applies to all of the framework code:

- Keep files small; avoid having more than one significant class in a file.

For object types and classes we'll extend using derivation, we can provide more guidelines:

- Keep functions small
- Make functions virtual
- Don't have stack instances or embed instances of one object type within another.

- Don't use **`new ClassName`** within the framework code; instead have a separate factory method for each class.
- Avoid the copy constructor and assignment operator; use a **`clone`** function instead

It takes skilful design and much experience to design a good extensible framework. However these guidelines provide a starting point for the implementation

Conclusion

In this article I've outlined four principles for managing changes to third party code:

Principle 1: Keep our code separate from their code

Principle 2: Minimise the number of changes in their code

Principle 3: Mark any changes in their code so they are easy to locate

Principle 4: Use code merging tools to incorporate changes

I've also shown three practical coding techniques you can use to modify library code while keeping to the first two principles:

Technique	**Application**
Changing object types using derivation	To change the behaviour of an object type.
Using the pre-processor and linker to replace an implementation	To completely replace the implementation of one or more member functions.
Replacing classes using the pre-processor	To change the behaviour of a value type.

Finally I've distilled all the potential problems in using these techniques to produce a set of guidelines for framework vendors.

It's rare indeed to find a complex framework well-enough designed to anticipate all our requirements. With these techniques we can produce maintainable code to build on any framework, no matter what our requirements may be.

References

1. *Design Patterns*, Gamma, Helm, Johnson and Vlissides, Addison-Wesley 1994, ISBN 0-201-63361-2

The Author

Charles Weir is a consultant software architect specialising in O-O design. He is based in England and can be reached as Charles.Weir@iee.org.uk