

Split Idioms

by Stephen C. Dewhurst

Word has somehow got around that a split infinitive is always wrong. This is of a piece with the sentimental and outworn notion that it is always wrong to strike a lady.

James Thurber, *The Owl in the Attic and Other Perplexities*

Break any of these rules sooner than say anything outright barbarous.

George Orwell, *Politics and the English Language*

I've always thought that both the curious observation from Mr. Thurber and the more staid advice from Mr. Orwell had application to object-oriented design. In earlier columns, I may have inadvertently given the impression that our C++ programming idioms are in some way a straightjacket that controls every aspect of the design process. Far from it. Properly used, idiom can simplify the process of design and communication of a design, leaving the designer free to expend creativity where it's needed. However, there are times when use of even the most sensible and common programming idiom is inappropriate to the context of a design, and the designer is forced to depart from the standard approach to a problem.

In this installment, we'll examine two of our most cherished C++ programming idioms, and show how departing from them in an intelligent and reasoned manner can result in a better design than we would have achieved by following them blindly.

Overloading Virtual Functions

In general, we avoid overloading virtual functions, because doing so is likely to defeat polymorphism. If we define polymorphism as the ability of an object to respond to several different interfaces, then we must ensure that an operation on the object exhibits the same behavior no matter what interface is used to access it. Consider the following base class fragment:

```
class Shape {
public:
    //...
    virtual void rotate( int );
    virtual void rotate( double );
};
```

Consider a derived class produced by a designer who has determined that only the integer version of

rotate requires different behavior in the derived class.

```
class Blob : public Shape {
public:
    //...
    void rotate( int );
};
```

Here we have an unhappy confluence of overloading and overriding—which have nothing to do with each other. The behavior of a Blob object will vary depending on the interface used to access it.

```
Blob *blob = new Blob;
Shape *shape = blob;
shape->rotate( 12.3 ); // OK
blob->rotate( 12.3 ); // oops! rotate(int)!
```

So, for well-behaved polymorphism, it is almost always a good idea to avoid overloading virtual functions.

Virtual Visiting

Almost always. Let's consider an instance of the Visitor¹ pattern for a hierarchy of financial instruments, where the hierarchy uses multiple inheritance to leverage subsystems written in terms of interface classes.

```
class Saveable { // persistent interface
    virtual void save() = 0;
    // ...
};

class Priceable { // pricing interface
    virtual void price() = 0;
    // ...
};

class Deal {
public:
    virtual void accept( DealVisitor & ) = 0;
    virtual void validate() = 0;
    //...
};

class Bond : public Deal, public Priceable {
    // Bonds can be priced
public:
    void accept( DealVisitor & );
```

```
    void validate();
    void price();
    //...
};
class Swap : public Deal, public Priceable, public Saveable {
    // Swaps can be priced and saved
public:
    void accept( DealVisitor & );
    void validate();
    void price();
    void save();
    //...
};

class Eq : public Deal
    // can't do a thing with equities lately...
    { /* ... */ };
```

The use of a Visitor allows us to effectively add new operations to the Deal hierarchy without recompilation—an important property if we don't have access to the source code and need to immediately perform some maintenance. If we follow our idiom above, we'll produce a DealVisitor interface with differently named functions for each deal type.

```
class DealVisitor {
public:
    virtual void visitBond( Bond * ) = 0;
    virtual void visitSwap( Swap * ) = 0;
    virtual void visitEq( Eq * ) = 0;
};
```

The accept operation defined by each concrete Deal type must be written to specifically invoke the corresponding member of the DealVisitor.

```
void Swap::accept( DealVisitor &v )
    { v.visitSwap( this ); }
void Bond::accept( DealVisitor &v )
    { v.visitBond( this ); }
void Eq::accept( DealVisitor &v )
    { v.visitEq( this ); }
```

This allows us to easily add new operations to the hierarchy without recompiling it.

```
class ProcessDeal : public DealVisitor {
    void visitBond( Bond *d ) {
        d->validate();
        d->price();
    }
    void visitSwap( Swap *d ) {
        d->validate();
        d->price();
        d->save();
    }
    void visitEq( Eq *d ) {
        d->validate();
    }
};
```

However, this design has a problem. We've employed a Visitor in the Deal hierarchy in order to be able to easily maintain the hierarchy without having access to the source. After all, if we were able to straightforwardly modify and recompile the Deal hierarchy, we would simply have added the ability to directly process Deals to the Deal classes themselves.

```
class Deal {
public:
    virtual void accept( DealVisitor & ) = 0;
    virtual void validate() = 0;
    virtual void process() = 0;
    //...
};
```

What happens if we add a new class to the hierarchy and would like to process it? The Visitor pattern works tremendously well if the hierarchies it visits are stable, but breaks loudly when the structure of a visited hierarchy is modified. Generally, this loud breakage is advantageous, because it detects errors that other approaches, such as runtime type queries, tend to overlook. However, in this case we would like to easily add a new Deal type and process it without recompiling the existing source. The problem is that adding a new Deal class requires modification of the DealVisitor class, which in turn requires recompilation of the Deal hierarchy. No good.

Let's overload the virtual visit functions in the DealVisitor. Live a little.

```
class DealVisitor {
public:
    virtual void visit( Deal * );
    virtual void visit( Bond * ) = 0;
```

```
virtual void visit( Swap * ) = 0;
virtual void visit( Eq * ) = 0;
};
```

Here we've used the differing argument types of the `visit` members to distinguish the functions. One advantage of this approach is that the source code of the various `deal accept` functions is simplified.

```
void Swap::accept( DealVisitor &v )
{ v.visit( this ); }
void Swap::accept( DealVisitor &v )
{ v.visit( this ); }
void NewDealType::accept( DealVisitor &v )
{ v.visit( this ); }
```

The `accept` operations for `Swap`, `Bond`, and `Eq` still invoke the appropriate type-specific `visit` operations. However, the use of overloading has allowed us to explicitly add a “hook” to the `DealVisitor` base class that can accept any type of `Deal` that is not handled by more specific `visit` operations. This means that our `NewDealType` can be visited by any `DealVisitor`, and will be processed with whatever default behavior is provided in `DealVisitor::visit(Deal*)`. In the future, if we are able to modify the `DealVisitor` hierarchy to add a `visit(NewDealType *)` to the interface, `NewDealType::accept` will be automatically modified on recompilation to invoke the more specific `visit` operation. No source code changes.

The real advantage of this approach is that it permits some seriously macho hacking (all in a good cause, of course).² Given the presence of the catchall hook in the base class, we can derive a special-purpose `DealVisitor` that is aware of the existence of `NewDealType`.

```
class ProcessDealHack : public ProcessDeal {
    void visit( Deal *d ) {
        if( NewDealType *n = dynamic_cast<NewDealType *>(d) )
            // process the NewDealType
        else
            // base processes unrecognized deal types
            ProcessDeal::visit( d );
    }
};
```

“What’s this?” you may be saying, “Is this the guy who’s been railing against overuse of runtime type information?” Hey, read the title of the article. This is a (hopefully) temporary hack, and a very clever one. The appropriately-named `ProcessDealHack` hijacks `ProcessDeal::visit(Deal *)` to check for any new `Deal` types that may have come into existence since the compilation and release of the `Deal` hierarchy. This is only possible if we overload the virtual `visit` functions in the `DealVisitor` base class, in contradiction of the usual design idiom.

Note also, that the major difficulty engendered by overloading virtual functions occurs when an object is accessed through both its base class and derived class interfaces. In the case of an instance of the Visitor pattern, the concrete derived visitor types are accessed only through the base class interface. In the code

examples above, we've made this general policy explicit by overriding the public base class virtual functions with private derived class functions.³

Copy Operation Implementation

One of the most common and useful C++ idioms is the copy operation idiom. Every abstract data type in C++ should make a decision about its copy assignment operator and copy constructor. Either the compiler should be allowed to write them, the programmer should write them, or they should be disallowed.

If the programmer writes these operations, we know exactly how they should be written. (However, the “standard” way of writing these operations has evolved over the years. This is one of the advantages of idiom over dictate; idiom evolves to suit the current context of use.)

```
class X {
public:
    X( const X & );
    X &operator =( const X & );
    //...
};
```

While the C++ language permits a lot of leeway in their definitions, it is almost invariably a good idea to actually declare these operations as they are above: both operations take a reference to a constant, and the copy assignment is non-virtual and returns a reference to a non-const. Clearly, neither of these operations will change its operand. It wouldn't make sense.

```
X a;
X b( a ); // a won't change
a = b; // b won't change
```

Non-Standard Copying

Except sometimes. The standard C++ `auto_ptr` template has some very unusual requirements. It is a resource handle charged with cleaning up heap-allocated storage when it is no longer needed.

```
void f() {
    auto_ptr<Blob> blob( new Blob );
    //...
    // automatic deletion of Blob
}
```

Fine, but what happens when the student interns are set loose on this code?

```
void g( auto_ptr<Blob> arg ) {
    //...
    // automatic deletion of Blob
```

```
}  
void f() {  
    auto_ptr<Blob> blob( new Blob );  
    g( blob );  
    // another deletion of Blob!!!  
}
```

One approach might be to unambiguously disallow copy operations for `auto_ptr`, but that would severely limit their use, and make impossible a number of useful `auto_ptr` idioms. Another approach might be to invisibly add a reference count to the `auto_ptr`, but that would increase the expense of employing a resource handle. The approach taken by the standard `auto_ptr` was to intentionally depart from the copy operation idiom.

```
template <class T>  
class auto_ptr {  
public:  
    auto_ptr( auto_ptr & );  
    auto_ptr &operator =( auto_ptr & );  
    //...  
private:  
    T *object_;  
};
```

(The standard `auto_ptr` also implements template member functions corresponding to these non-template copy operations, but the observations for those are similar.) Here the right hand side of each operation is non-const! When one `auto_ptr` is initialized by or assigned to by another, the source of the initialization or assignment gives up ownership of the heap-allocated object to which it refers by setting its internal object pointer to null.

As is often the case when departing from idiom, there was some initial confusion about how `auto_ptr` should be properly used. However, this departure has allowed the development of a number of very productive new idioms centered around object ownership issues, and the use of `auto_ptr`s as “sources” and “sinks” of data looks to really be a very profitable new design area. In effect, a required departure from an existing, successful idiom has resulted in a family of new idioms.

Gotta Split

We could obviously examine many other idioms that have been successfully and properly circumvented. However, the lesson of our two examples above is clear. To slavishly follow idiom in the face of an obviously better design is often (as Winston Churchill was said to sardonically remark when an over-zealous editor criticized his ending a sentence with a preposition) “an impertinence up with which [we should] not put.” It is better in such situations to rationally follow the better design and boldly go where no programmer has gone before.

end of article

¹ Gamma, et al., *Design Patterns*, Addison Wesley, 1995, p. 331.

² John Vlissides, *Pattern Hatching*, Addison Wesley 1998, p. 79. I am not accusing the author of being macho, unless he would like to be so accused.

³ Don't try this in Java. It won't work. This is, I think, an example of the danger of trying to encode "good" programming style in the programming language itself rather than in its idiomatic usage.