

Removing Compilation Dependencies between C++ Classes

Charles Weir

© Object Designers Limited, September 1993

C++ has a major weakness when it comes to compilation. At present there are no code management tools or development environments capable of distinguishing between the public and private sections of class declarations. So if you change the private data or private definitions within any of your class declarations, the compiler will need to recompile every file that uses that class - even if the public interface has not changed at all. A small change to one of the private function declarations in a base class could well require the complete recompilation of a vast system - even though nothing significant has changed.

There are two different approaches to solving this problem. They are known as Abstract Base classes, and Cheshire Cat classes. This article looks at the implications of both approaches, and examines each in detail. As an illustration, we shall attempt to implement a Motor object, which represents a motor controlled by our software. The external interface can start and stop this motor, and set its speed. A motor is represented by an instance of a C++ class, *Motor*; we shall examine the implementation of this class.

Abstract Base Classes

An abstract base class declares only pure virtual functions; these are functions with declarations but no implementation. Using this approach, the public header file for an object declares its supported functions using an abstract base class. In a different header or implementation file, a derived class defines the actual implementation. Thus the abstract base class creates an "Abstract Interface" to the class; code that uses an instance of the class can refer to it solely in terms of the abstract base class, and need not have any compilation dependencies on the implementation class. For example, in the public header file, *motor.h*, you might have:

```
class Motor {
public:
    virtual void start() = 0;    // Starts the motor
    virtual void stop() = 0;    // Stops the motor
    virtual void setSpeed( float ) = 0; // Sets the speed
};
```

And in an implementation file:

```
class MotorImplementation : public Motor {
public:
    MotorImplementation( char* name ); // Stores name for debugging..
    void start();
    void stop();
private:
    // ... Implementation specific data goes here.
};
```

One problem with this approach is that there is no way to create an instance of the class without including the definition of the implementation class; the compiler cannot create a call to *new* unless it "knows" the size of the object it is constructing.

One way around this is to define one or more creation functions as part of the abstract interface. These are static functions which create a new instance of the implementation, and return a pointer of the appropriate type. Since they are deputising for the constructors, they will take the same parameters as their corresponding constructor functions.

```

class Motor {
public:
    static Motor* create( char* ); // Answers a new motor instance.
    virtual void start() = 0;      // Starts the motor
    virtual void stop() = 0;       // Stops the motor
    virtual void setSpeed( float ) = 0; // Sets the speed
};

```

It's tempting to make this create function **inline**. We cannot do that, though, without giving knowledge of MotorImplementation to clients, and defeating the entire purpose of the construction. So the **create** function goes in the implementation file:

```

Motor* Motor::create( char* name )
{ return new MotorImplementation( name ); }

```

The syntax for creating a new Motor on the heap is now rather non-standard, but certainly keeps the users of the item from needing any knowledge of the MotorImplementation class:

```

Motor* pMotor1 = Motor::create( "Motor 1" );

```

There is no way to create a Motor instance on the stack, in static memory, or embedded within another object; but in all other ways this Motor class behaves identically to one implemented without an abstract interface. There is a cost, since all its functions must be **virtual**, but in practice there are few circumstances where this cost becomes significant.

Cheshire Cat Classes

There is another way to achieve a similar separation between interface and implementation. You may remember that in the book "Alice in Wonderland", the Cheshire Cat fades away until just the Smile remains. With the Cheshire Cat approach, the Smile is the interface, visible much further than the remainder of the implementation; the name was invented by Glockenspiel, who use it for their class libraries. The class declaration in the public header file looks normal, but its only item of private data is a pointer to the instance of the implementation class. Each public function in the interface class merely passes the request on the corresponding function in the implementation class; hence these classes are often called "handle" classes, since they provide a handle to the underlying implementation:

```

class MotorImplementation; // Forward declaration.

class Motor {
public:
    Motor( char* );
    ~Motor();
    void start();
    void stop();
    void setSpeed( float );
protected:
    MotorImplementation * p;
};

```

The implementation files define the class MotorImplementation; each function in Motor calls the corresponding function in MotorImplementation.

```

class MotorImplementation {
friend class Motor;
protected:
    MotorImplementation( char* );
    void start();
    void stop();
    void setSpeed( float );
private:
    // Private functions and data go here...
};

Motor::Motor( char* s )
    : p( new MotorImplementation( s ) ) {}

Motor::~~Motor()    { delete p; }

```

```

void Motor::start() { p->start(); }
void Motor::stop() { p->stop(); }
void Motor::setSpeed( float s ) { p->setSpeed( s ); }

```

Only instances of the `Motor` class, (and of sub-classes of the `MotorImplementation` class) will access the functions in `MotorImplementation`, so these functions are *protected*.

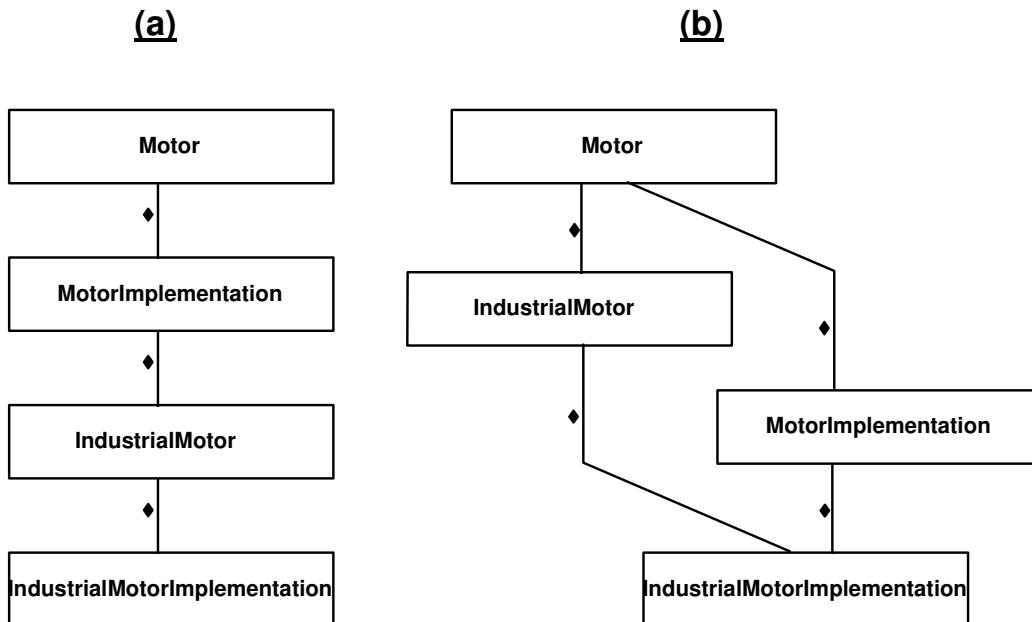
This approach has avoided the irritations of the Abstract Interface approach: we can create a Cheshire Cat *Motor* on the heap, in static memory, or on the stack. In use it behaves identically to the simple implementation. The cost is higher, though: the mechanism to redirect calls to the implementation is more costly both in code written and in execution time (each function requires two C++ function calls, a virtual function is equivalent to about 1.5 non-virtual calls); the Cheshire Cat class must have a destructor even if the implementation has none; and there is an additional heap allocation and de-allocation for each instance created.

The Implications of Inheritance

Suppose we want to have another version of `Motor` with additional functionality; *IndustrialMotor*, say. This has a different (although compatible) implementation of the *start* function, and is intelligent enough to know its actual speed; it has an extra function, *getSpeed*, for this purpose. So we create a derived class, *IndustrialMotor*, and redefine functions as necessary. Clearly we want to keep the same scheme (Abstract Interface, or Cheshire Cat) for the derived class as for the base class.

Inheritance with Abstract Interfaces

Using the abstract interface approach there are the following two possibilities for the inheritance hierarchy:



Scheme (a) is straightforward, and uses only single inheritance. Unfortunately, it does not achieve the desired result; to use `IndustrialMotor`, a file must include the definition of `MotorImplementation`, and that is exactly what we are trying to avoid. So scheme (a) fails.

Scheme (b) uses multiple inheritance. In fact it needs more than just simple multiple inheritance. Normally, each instance of a derived class contains an instance of the base class; thus with simple multiple inheritance, an instance of `IndustrialMotorImplementation` will contain two instances of the class `Motor`, one from `MotorImplementation`, and one from `IndustrialMotor`. The solution to this problem is "virtual inheritance", which tells the compiler to create only one instance of the shared base class:

```
class MotorImplementation : public virtual Motor
{ /* ... */ };

class IndustrialMotor : public virtual Motor
{ /* ... */ };

class IndustrialMotorImplementation :
    public IndustrialMotor, public MotorImplementation
{ /* ... */ };
```

This solves the problem in this case; however further derived classes will encounter the same problem. So a more useful definition of `IndustrialMotorImplementation` would be:

```
class IndustrialMotorImplementation :
    public virtual IndustrialMotor, public virtual MotorImplementation
{ /* ... */ };
```

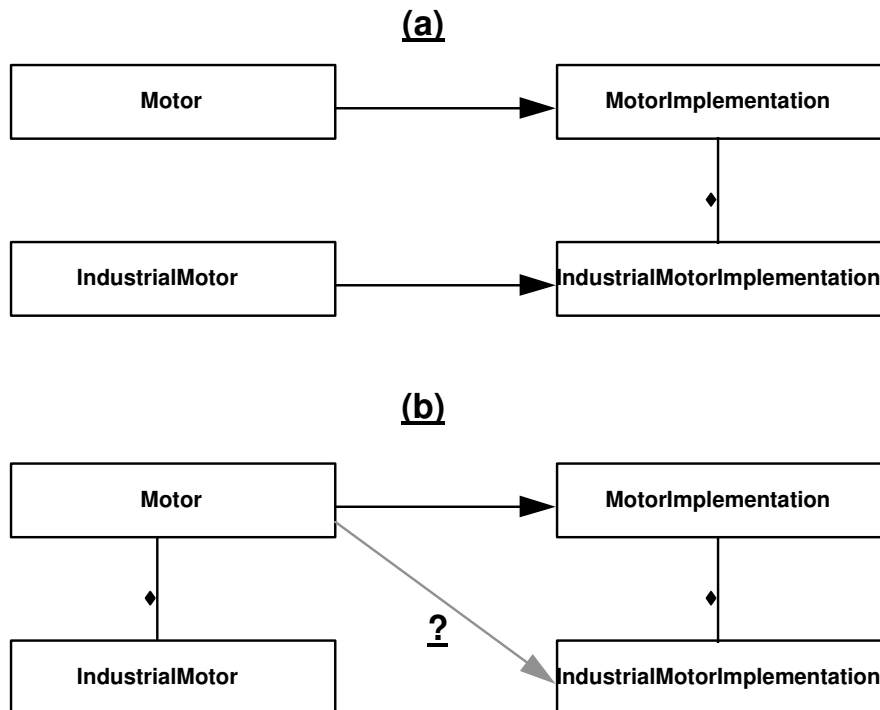
Unfortunately, there are some restrictions on what we can do with virtual inheritance. The compiler will call only the default constructor on the base class; constructor initialisation lists cannot pass parameters to the constructors of virtual base classes. So if we need to pass parameters to the constructor of the base class, we're stuck. In addition, this approach will stress the implementations of our compiler and debugger, since multiple virtual inheritance is a wonderful arena for obscure compiler bugs. Although the best current implementations are unlikely to give us serious problems, we must be prepared for surprises.

So we can achieve inheritance, but at the cost of added complexity, of compiler stress, and of additional restrictions on what we can do with inheritance. How does the Cheshire Cat approach fare by comparison?

Inheritance and the Cheshire Cat

The situation is rather different with the Cheshire Cat approach. If we try to get inheritance we end up with several possibilities. In particular, should the handle classes inherit from each other, or not, and what are the implications of this inheritance?

The following diagram shows the two possibilities.



In (a) the Handle classes do not inherit; only the implementations do. This makes the implementation of the classes simple, at the cost of duplicating all of the public functions in `Motor` in the derived class. However, there is a problem if we need a generic pointer that can point either to a `Motor` or a `IndustrialMotor`. The Abstract Interface approach provided this polymorphism nicely; here it is not possible at all. So approach (a) will not help us if we want the inheritance to be visible to users of the objects.

The approach shown in (b) gets around this problem. However it immediately raises problems of its own. We want a new instance of `IndustrialMotorImplementation` when we create a new `IndustrialMotor`, and a new `MotorImplementation` when we create a new `Motor`. But, because the `IndustrialMotor` constructor calls the `Motor` constructor, we will end up with a `MotorImplementation` in both cases; this is not the behaviour we need.

One possible approach would be to move the allocation of the memory to a separate function, and to make this function virtual, so that the constructor for `Motor` calls the version overridden in `IndustrialMotor`. Unfortunately, this approach simply doesn't work: if the compiler works according to the specification in the Annotated Reference Manual, it will ignore the virtual-ness of the function and simply generate a call to `Motor`'s version of the function. This strange behaviour ensures that the derived instance does not receive function calls before it has been constructed.

A more successful approach is to define an additional protected constructor for `Motor`, which does not allocate memory. Then we can use the constructor initialisation list for `IndustrialMotor` to call that constructor rather than any other. The parameter for this protected constructor is the already defined type `MotorImplementation*`, which should not be a parameter to any publicly visible constructor for `Motor`.

As it stands, the `IndustrialMotor` class will correctly call functions defined by `Motor` and implemented by `MotorImplementation`. However it requires some more work before we can define new functions. The pointer in `Motor` can be `protected`, rather than `private`, but it is still of type `MotorImplementation*`. So if we are to use it to call functions

specific to `IndustrialMotorImplementation` we must cast the pointer. We can define an inline function to do this cast.

There's still another potential problem, the destructor. We cannot prevent the destructor being called for the `Motor` instance, and it will call **`delete`** for the pointer as for a `MotorImplementation`; even though it is actually an instance of the derived class. As usual, the solution is to declare the destructor for the `MotorImplementation` class as **`virtual`**; there is no need to define a separate destructor for the `IndustrialMotor` class.

The resulting code in the public header files might look similar to the following:

```
class MotorImplementation;    // Forward declaration.

class Motor {
public:
    Motor( char* );
    ~Motor();
    void start();
    void stop();
    void setSpeed( float );
protected:
    Motor (MotorImplementation* x) : p( x ) {};
    MotorImplementation * p;
};

class IndustrialMotorImplementation;

class IndustrialMotor : public Motor
{
public:
    IndustrialMotor( char* );
    float getSpeed();
protected:
    IndustrialMotor( IndustrialMotorImplementation* );
private:
    IndustrialMotorImplementation * asIMI()    // Casting function.
    { return (IndustrialMotorImplementation*) p; }
};
```

Note that there is no reason to make functions virtual in `Motor` or `IndustrialMotor`, nor is there a need to re-define these functions in derived classes; instead, we can make the functions virtual in the implementation classes:

```
class MotorImplementation {
friend class Motor;
protected:
    MotorImplementation( char* );
    ~MotorImplementation() {} // Ensures that constructors are virtual.
    virtual void start();    // This function must now be virtual
    virtual void stop();    // so make the rest so for consistency...
    virtual void setSpeed( float );
private:
    // Private functions and data go here...
};

class IndustrialMotorImplementation : public MotorImplementation {
friend class IndustrialMotor;
protected:
    IndustrialMotorImplementation( char* );
    void start();    // Re-implemented function.
    virtual float getSpeed(); // Newly defined function.
};

Motor::Motor( char* s )
    : p( new MotorImplementation( s ) ) {}

Motor::~~Motor()    { delete p; }

void Motor::start() { p->start(); }
void Motor::stop() { p->stop(); }
void Motor::setSpeed( float s ) { p->setSpeed( s ); }
```

```

IndustrialMotor::IndustrialMotor( char* name )
    : Motor( new IndustrialMotorImplementation( name ) )
{}

IndustrialMotor::IndustrialMotor( IndustrialMotorImplementation* x )
    : Motor( x ) {};

float IndustrialMotor::getSpeed()
{ return asIMI()->getSpeed(); }

```

With this "boiler plate" code for each class, inheritance will work correctly. This inheritance doesn't add any further overheads or complexity for the compiler.

Conclusion

So, in practice, both Abstract Interface and Cheshire Cat classes can separate the interface of a class from its implementation, and both approaches can handle inheritance. However each has its drawbacks. Cheshire Cat classes:

- Require rather more "boiler-plate" code for each class and function defined.
- Have a (usually insignificant) processing overhead on each function call, and a space overhead on each instance created.

By comparison, Abstract Interface classes:

- Cannot be created in the normal way, using *new*.
- Make heavy demands on the compiler implementation.

Thus for a class library one might chose the Cheshire Cat approach; for a large, time-critical, application using a very reliable compiler one might choose Abstract Interface classes; a small application would probably use neither. The choice will depend, therefore, on the demands of your particular application, and the support provided by the compiler you are using.