Reducing Code Entropy

Adrian Kosmaczewski

2007-03-18

This is a rant: I am tired of seeing virtual methods implemented in child classes that, at some point or another, call the method of the same name in the base class. For me this is a sign of poor architecture. A bad, bad smell in code.

Let's say that you have a base class, called, well, `BaseClass` (the examples are in C++, but you can take this idea to any other OO language):

```
class BaseClass
{
public:
    // constructor, destructor, copy
    // constructor, assignment operator...

    virtual void doSomething();
}

void BaseClass::doSomething()
{
    // provide some basic behavior here,
    // and a comment that says something like:

    // IMPORTANT NOTE to implementors of subclasses:
    // this method should always be called before your code!
}
```

And then you derive a class from this base class, called, well, `DerivedClass`:

```
class DerivedClass : public BaseClass
{
public:
    // constructor, destructor, copy
    // constructor, assignment operator...

    virtual void doSomething();
}
```

```
void DerivedClass::doSomething()
{
    BaseClass::doSomething();    // WTF???

    // do something else...
}
```

As stupid as it might sound, this code is hard to understand, debug and maintain, because it includes an unneeded dependency from the child to the base class - that is, another one, besides the fact that one derives from the other!

The problem is that the `doSomething()` message appears here as actually two messages: one which is mandatory and generic (like initialization or verification), and the other, a specialized one (which is why people use polymorphism, actually). One is public, the other is not. Mixing both messages creates a semantic problem in the above code, which ultimately makes the code harder to understand.

This might lead a myriad of other small things; what if another developer (just landed in the team), months later has to add a new child class of `BaseClass`, and forgots to (or just does not know that she should) call the `BaseClass::doSomething()` method? This might cause pain, either because it generates a bug that's hard to track, either because the developer spends time trying to understand why the new class does not work while she's working on it. In any case, it is a cost that you, as an architect, might have avoided in the first place. And if your designs are poorly documented (which is a sad truth), or if the developer that created the code has gone away from the team (which happens a heckuva lot of times every day), well, your chances of losing time and effort in your project will grow accordingly.

In my opinion, such an approach goes against the very idea of polymorphism, and I prefer to refactor such a code this way:

```
class BaseClass
{
public:
    // constructor, destructor, copy
    // constructor, assignment operator...

    void doSomething();

protected:
    // a pure virtual, abstract method
    // without implementation, working as a
    // "hook" for subclass implementors
    virtual void doSomethingElse() = 0;
}
```

```
void BaseClass::doSomething()
{
    // provide some mandatory, basic behavior here

    // hook for subclasses to extend this behavior!
    doSomethingElse();
}
```

And then:

```
class DerivedClass : public BaseClass
{
public:
    // constructor, destructor, copy
    // constructor, assignment operator...

protected:
    // provide some extended behavior!
    virtual void doSomethingElse();
}

void DerivedClass::doSomethingElse()
{
    // do something else...
}
```

Ahhh… I much more prefer this way of doing things:

- First of all, it makes `BaseClass` an abstract entity, an interface, a pure thought creation, a divine entity above all the earthling vacuum; you want to program against interfaces, and not be tempted to use them as concrete things. That's what subclasses are for.
- The call to `doSomething()` will always, always (let me repeat that, always) be called, no matter how sloppy the creator of `DerivedClass` is. The worst thing you can have is an empty `DerivedClass::doSomethingElse()` implementation. In that case, no harm.
- If the developer is so, so, so sloppy that it forgets to implement `DerivedClass::doSomethingElse()`, well, guess what: at some point in time, either the compiler or the linker will complain! And this is a good thing (at least in languages that feature a compiler, of course!)
- You've unknowingly created a nice API, by the way. You can document this with a good UML diagram, and it will stand out in your Doxygen or JavaDoc documentation.
- It's easy to understand and hard to break, as all good designs should be.
- Your developers will love your work, and your karma gets a bonus point. The next life you might as well approach a higher level of consciousness and save Mankind from its horrible destiny. (OK, maybe this is too much, but hey, who knows?)

- The approach is called the “Template Design Pattern” and this will add another bonus, but this time for your CV ;)

Hope this helps!