Tutorial Corner: Attack of the Clones

Today I saw a newsgroup posting that asked the following question:

> "What is the exact purpose of the `clone()` function? After all, it merely gets an exact copy of the object at which its formal argument is pointing? What is it going to do with a mere copy when it has the actual object to work with?"

This is a reasonable and common question. It does seem rather counter intuitive to have to implement such an operation, but many times you can actually have the ability to point to something without actually knowing precisely what it is you're pointing to. However, the situation is such that, whatever it is you're pointing to, you want another one just like it! Let's look at an analogous real world situation.

Anything but Lutfisk

Suppose you find yourself in a Swedish restaurant, and you'd like to order a meal. Unfortunately, your knowledge of Swedish is limited to technical correspondence, cursing, or (typically) a combination of the two. The menu is in Swedish, and you can't read Swedish, but you do notice a gentleman on the other side of the room who is really enjoying his meal. Therefore, you call over the waiter and say:

> *If that gentleman is having fish, I'd like fish. If he's having spaghetti, I'd like spaghetti too. Otherwise, if he's having eel, then eel it is. However, if he's decided on the preserved kumquats, then I'll have that.*

Does this sound reasonable? Of course not. (For one thing, you probably don't want to order spaghetti in a Swedish restaurant.) There are two basic problems with this procedure. First, it's tedious and inefficient. Second, it can fail. What would happen if you come to the end of your sequence of questions and you still haven't been able to guess what the other diner is eating? The waiter will walk off, leaving you stranded and hungry. Even if you do happen to know the entire content of the menu, and are therefore guaranteed of (eventual) success, if the menu is modified between your visits to the restaurant, your list of questions may become invalid or incomplete.

The proper approach, of course, is simply to call the waiter over, and say:

> *I'd like what that gentleman is having.*

If the waiter is a literalist, he'll snatch up the other diner's half-finished meal and bring it to your table. However, that sort of behavior can lead to hurt feelings and even a food fight. This is the sort of unpleasantness that can occur when two diners try to consume the same meal at the same time. If he knows his business, the waiter will deliver an exact copy of the other diner's meal to you, without affecting the state of the meal that is copied.

These are the two major reasons for cloning; you must or you prefer to remain in ignorance about the precise type of object you're dealing with, and you don't want to effect change or be affected by changes the original object.

Virtual Constructors and Prototypes

A member function that provides the ability to clone an object is traditionally called a "virtual constructor" in C++. Of course, there are no virtual constructors, but producing a copy of an object often involves indirect invocation of its class's copy constructor through a virtual function, giving the effect—if not the reality—of virtual construction. More recently, this approach has been called an instance of the Prototype pattern.[1]

Of course, we do have to know something about the object to which we refer. In our case, we know that what we want is-a meal.

```
class Meal {
  public:
    virtual ~Meal();
    const string &id() const;
    virtual void eat() = 0;
    virtual Meal *clone() const = 0;
    //…
};
```

The `Meal` type provides the ability to clone with the `clone` member function. The `clone` function is actually a rather specialized kind of Factory Method[2] that manufactures an appropriate product while allowing the invoking code to remain in ignorance of the exact type of context and product class. Concrete classes derived from `Meal` (that is, those meals that actually exist and are listed on the menu) must provide an implementation of the pure virtual `clone` operation.[3]

```
class Lutfisk : public Meal, private Punishment {
  public:
    Lutfisk( const Lutfisk & );
    void eat();
    Lutfisk *clone() const;
};
class Spaghetti : public Meal {
  public:
    Spaghetti( const Spaghetti & );
    void eat();
    Spaghetti *clone() const;
```

---

[1] The reference is, of course, *Design Patterns*, Gamma et al., p. 117. Kevlin Henney has told me that the virtual constructor idiom is very different from the Prototype pattern, but I don't see a significant difference. However, one must always take Kevlin seriously.

[2] Ibid., p. 107.

[3] Notice the use of the covariant return type in the overriding derived class function. Even though the overridden base class virtual function `Meal::clone` returns a `Meal *`, it is legal for `Lutfisk::clone` to return a `Lutfisk *`. See *C++ Gotchas*, p. 228 for more detail. Note that some outdated but commonly used versions of C++ compilers do not support covariant return types. If yours doesn't, just have `Lutfisk::clone` return a `Meal *`.

```
//…
Lutfisk *Lutfisk::clone() const
    { return new Lutfisk( *this ); }
Spaghetti *Spaghetti::clone() const
    { return new Spaghetti( *this ); }
```

With this simple framework in place, we have the ability to produce an exact copy of any type of `Meal` without precise knowledge about the actual type of the `Meal` we're copying. In the code below, note that there is no mention of concrete derived classes, and therefore no coupling of the code to any current *or future* types derived from `Meal`.

```
Meal *m = thatGuysMeal(); // whatever he's having…
Meal *myMeal = m->clone(); // …I want one too!
```

In fact, we could end up ordering something that we've never even heard of. In effect, with Prototype, ignorance of the existence of a type is no barrier to creating an object of that type. The "generic" code above can be compiled and distributed, and later be augmented with new types of `Meal` without the need for recompilation.

Ignorance is Bliss

This somewhat silly example serves to illustrate some of the advantages of ignorance in software design, particularly in the design of software structured as a framework that is designed for customization and extension: The less you know, the better.

For example, consider a very simple GUI framework consisting of a button type and a callback action that is executed if the button is pressed. For flexibility, we implement the action as a function object. That is, rather than use a function pointer to indicate what action to execute when the button is pressed, we use a class object that stands in place of a function. The framework code looks like this:

```
class Action {
  public:
    virtual ~Action();
    virtual void operator ()() = 0;
};

class Button {
  public:
    Button( const char *label );
    ~Button();
    void press() const;
    void setAction( Action & );
  private:
    std::string _label;
    Action *_action; // doesn't own, problematic…
};

Button::Button( const char *label )
```

```
    : _label( label ), _action( 0 ) {}

void Button::setAction( Action &action )
    { _action = &action; } // problematic…

void Button::press() const
    { if( _action ) (*_action)(); }
```

This code will work, in that we can derive new types of action from `Action` and attach them to `Button`s, while leaving `Button` and `Action` in ignorance of these new `Action` types. For instance, we could choose to do nothing at all:[4]

```
class NullAction : public Action {
  public:
    void operator ()()
        {}
};
```

Or we could decide to attract attention if a button is pressed.

```
struct Beep : public Action {
  public:
    void operator ()()
        { std::cout << '\a' << std::flush; }
};
```

Or we could decide to execute a tree of actions:[5]

```
class Macro : public Action {
  public:
    ~Macro();
    void add( Action *a )
        { _a.push_back( a ); }
    void operator ()() {
        for( std::list<Action *>::iterator i(_a.begin());
            i != _a.end(); ++i )
            (**i)();
    }
  private:
    std::list<Action *> _a;
};
```

Given the above, we can set an action to be executed when a button is pressed:

---

[4] This is an instance of the Null Object pattern. See, for example, `http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/NullObject.pdf`

[5] This is an instance of the Composite pattern. See GOF, p. 163. We can also decorate `Action`s with application of a Decorator (GOF, p. 175), but enough is enough!

```
// Initialize the button somewhere…
Button b( “Press Me” );
Macro m;
m.add( Beep() );
m.add( NullAction() );
m.add( Beep() );
b.setAction( m );
//…
// Somewhere else entirely, press the button…
b.press(); // oops!
```

We have a problem. The problem is in the setting of the action in `setAction`. Because it has no more specific information about the action other than that it is some kind of `Action`, the `Button` has no choice but to simply set a pointer to the action. Unfortunately, in the code above, the `Action` objects will have disappeared before the button is pressed, and the `Button` will contain a bad pointer to `Action`. Another problem could arise if multiple `Button` objects referred to the same `Action` object.

A better (or safer) implementation would allow an `Action` to be cloned, so that the `Button` can have sole ownership and full control of its `Action`.

```
class Action {
  public:
    virtual ~Action();
    virtual void operator ()() = 0;
    virtual Action *clone() const = 0;
};
```

Now the button can choose to work with its own exact copy of the action, with confidence that it will not disappear or be modified in the interim.[6]

```
class Button {
  public:
    //…
    void setAction( const Action & );
  private:
    std::string _label;
    Action *_action; // owns!
};

void Button::setAction( const Action &action ) {
    Action *newAction = action.clone();
    delete _action;
    _action = newAction;
```

[6] Thanks to Thorsten Ottosen for pointing out that an earlier version of this code was not exception safe in the presence of a failed clone attempt.

```
}
```

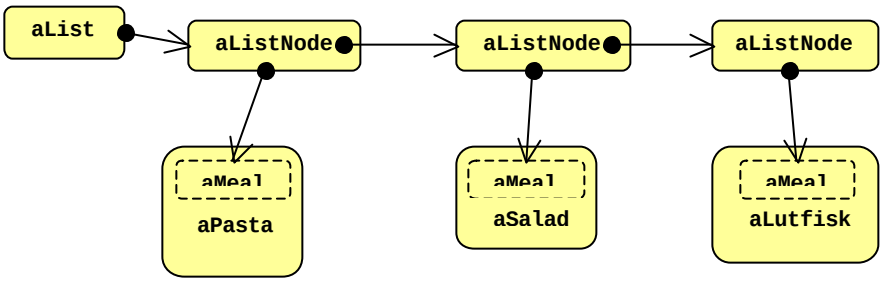
In effect, the button doesn't know what kind of action has been passed to it, but it wants another one just like it! Of course, the individual actions now have to know how to clone themselves:

```
NullAction *NullAction::clone() const
    { return new NullAction; }

Macro *Macro::clone() const {
    std::auto_ptr<Macro> m( new Macro );
    for( std::list<Action *>::const_iterator i(_a.begin());
            i != _a.end(); ++i )
        m->add( **i );
    return m.release();
}

void add( const Action &a )
        { _a.push_back( a.clone() ); }
```

An Old Prototype Trick

Occasionally, one hits on a design situation in which one is forced to write framework code that can deal not only with types it knows nothing about, or types that didn't exist when the framework was compiled, but types that didn't exist when the framework code started *executing*.

The clone operation can help here if the object required can be specified by some sort of identifier.[7] All we have to do is populate a data structure with a set of prototypical instances of the available types. For instance, we can represent the available choices on the menu of a Swedish restaurant by a list of `Meal`s:



This data structure can be searched at runtime for a meal that has the proper identifier. If such a meal exists, it can be cloned.

---

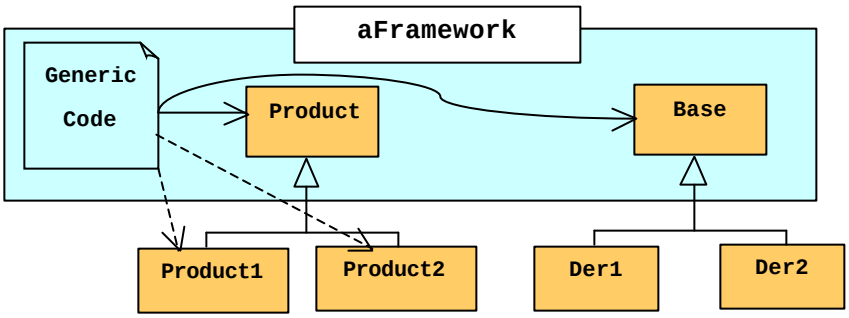
[7] Actually, by any set of requirements or set of questions one could pose to an object. That an object type has a particular identifier is just a special case; it's the requirement that the object can answer the question, "Is your name X?" We could just as well have asked, "Are you something other than lutfisk?" or "Are you commonly ordered by regular visitors to this restaurant?"

```
Meal *
giveMeSome( const string &id ) {
    for( ListNode *m = aList; m; m = m->next )
        if( m->meal->id() == id )
            return m->meal->clone();
    return 0;
}
```

Note that the data structure may be populated with derived class objects explicitly in an initialization phase, or implicitly using static or dynamic loading.[8]

Prototypes and Frameworks

A common problem in framework design concern the necessity of creating concrete types, when the framework code has been written, compiled, and distributed long in advance of the existence of the concrete types it must instantiate.



The clone operation/virtual constructor idiom/Prototype pattern is one common way of dealing with this situation.

[8] When dynamic loading is used to add a new type to the data structure at runtime, the technique is often called “dynamic inheritance.” It’s one of several ways to add a new type to an executing program.