Pattern Hatching

To Code or Not to Code, Part II

John Vlissides and Andrei Alexandrescu
C++ Report, June 2000

We've got a lot to ground to cover, so we'll cut right to the chase. If you haven't read Part I,[1] then run, don't walk, to your March issue and do so now. If you're missing that issue due to extenuating circumstances—your homeland only recently legalized *C++ Report*, for example—just drop John a note and he'll see what he can do.

One aspect of Andrei's Generic Pattern Implementations (GPI) that bears emphasis, or at least more emphasis than we gave it last time, is that GPI templates aren't patterns themselves; they're *implementations* of patterns. If that's obvious to you, consider yourself enlightened. Many respectable people, however, mistakenly view patterns as cookie-cutter solutions to the coding problem *du jour*. To them, the distinction is not so obvious.

Make no mistake: patterns have to be tailored to each problem by the sweat of one's brow and the firing of one's neurons. Customization is a big part of the pattern concept, and it has a big say in a pattern's effectiveness. It also makes patterns different from conventional tools of the programming trade—a difference that's lost on many folks. That explains why patterns are so often compared to components, data structures, frameworks, and other familiar fare.

So when we say, "GPI uses templates and generic programming techniques to capture common design pattern implementations," doubtless some readers will gloss over "implementations" and leap to the conclusion that GPI is patterns incarnate. It's not; it merely offers a way of expressing design pattern implementations a bit more succinctly and explicitly than bare C++. And because GPI works within C++ rather than without, you can use the full power of the language to tailor GPI to your needs. That's a crucial property given the power patterns derive from customization.

Typelists

Last time we availed ourselves of things called *typelists* without explaining how they work. All we said was they let you manipulate collections of types at compile-time much like you manipulate collections of values at run-time. Typelists debuted in the declaration of `WidgetFactory`, an example of an AbstractFactory class:[2]

```
class Button;
class ScrollBar;
class Menu;

typedef AbstractFactory <
    TYPELIST_3(Button, ScrollBar, Menu)
> WidgetFactory;
```

As you've probably guessed, a typelist of *n* types is declared using a macro of the form `TYPELIST_`*n*. But there's more going on here than meets the eye.

Typelists are founded on this simple template:

```
template <typename H, typename T>
struct Typelist {
    typedef H Head;
    typedef T Tail;
};
```

Although the template takes only two types, it's really open-ended because you can pass a `Typelist` as an argument. Hence the type

```
Typelist< char, Typelist<signed char, unsigned char> >
```

is effectively a typelist of three elements: `char`, `signed char`, and `unsigned char`.

By convention, typelists are tail recursive. While it's certainly possible to write

```
Typelist<Typelist<char, unsigned char>, char>
```

the GPI library stipulates that "well-formed" typelists never have a `Typelist` as their first argument. LISP-ers will recognize this as the difference between well-formed lists and arbitrary S-expressions. The rest of us will be content knowing only that this assumption makes the GPI implementer's life a whole lot easier.

If these typelist declarations look ugly to you, you're not alone. And they get uglier with each additional parameter. Consider a typelist of integral types:

```
typedef Typelist <
    signed char,
    Typelist< short int, Typelist<int, long int> >
> SignedIntegrals;
```

Not terribly readable. So GPI encourages you to transform the recursive structure into simple enumeration. This comes at the expense of some seriously tedious code—not your own, but GPI's. That is, the library supplies dozens of macros of the form we saw before, `TYPELIST_`*n*:

```
#define TYPELIST_2(T1, T2) Typelist<T1, T2>
#define TYPELIST_3(T1, T2, T3) Typelist<T1, TYPELIST_2(T2, T3)>
// etc.
```

Each macro uses the previous one, making it easy to extend the upper limit should you be unlucky enough to require more than several dozen types in a single list.

Now `SignedIntegrals` can be expressed more clearly and sweetly:

```
typedef TYPELIST_4 (
    signed char, short int, int, long int
) SignedIntegrals;
```

Operations on typelists

What can you do with a typelist? Lots of things, including

- computing its length
- removing an element, either by type or by position
- finding a type
- fetching the type at a given index (indexed access)
- removing duplicates—that is, transforming `TYPELIST_3(int, char, int)` into `TYPELIST_2(int, char)`

- sorting by inheritance relationship—for example, transforming `TYPELIST_3(MoreDerived, Base, Derived)` into `TYPELIST_3(Base, Derived, MoreDerived)`

Let's look at the simplest of these: calculating the length of a typelist. First off, assume that a single type is a typelist of length one. The length of a bigger typelist can then be defined recursively as 1 plus the length of the tail of that typelist. We say that in C++ like so:

```
template <class T> struct Length {
    enum { Value = 1; }
};

template <class T, class U>
struct Length < Typelist<T, U> > {
    enum { Value = 1 + Length<U>::Value; }
};
```

Now watch `Length` in action as it's applied to the `SignedIntegrals` typelist defined earlier:

```
int len = Length<SignedIntegrals>::Value;
```

The template argument here is a typelist, so the compiler will use the second definition of `Length`. Then the compiler will evaluate `Value`, which will require instantiating the tail of the typelist, `TYPELIST_3(short int, int, long int)`. But this too is a typelist, so the second `Length` template gets instantiated again.

The compiler continues this recursive instantiation process until the list is reduced to a single type, `long int` in this case. That's when the first version of `Length` finally kicks in to provide the initial value of 1. The number of recursion levels corresponds to the number of types in the typelist. As the recursion unwinds, 1 gets added to the value for each level of recursion, the resulting sum being the length of the typelist.

Clever? Perhaps, but nothing compared to other typelist operations in GPI. Modesty and space don't allow for their treatment here, but be not dismayed: Andrei covers them thoroughly in his upcoming book.[3] These operations and the type manipulations they allow are key to the magic of GPI.

Typelists in action

ABSTRACT FACTORY prescribes an `AbstractFactory::create...` operation (e.g., `createScrollBar`, `createButton`) for each ConcreteProduct type. Unfortunately, there's no way to produce such a slew of operation names with C++ templates directly. So GPI gets creative—*very* creative.

Again, the approach is recursive:

- For a single type `T`, the `AbstractFactory` class template declares one `doCreate(T*)` member function.
- For a typelist of `T` and `U`, `AbstractFactory` generates one `doCreate(T*)` member function, plus it declares all that an `AbstractFactory` for `U` would declare.

The second bullet introduces the recursion.

C++ does make it easy to have one class declare all that another class declares; it's called *inheritance*. Here's the gist of how inheritance and typelists are combined to templatize the AbstractFactory participant:[*]

---

[*] These and following template declarations exclude constructors, destructors, and other details that aren't germane to the discussion.

```
template <class T>
class AbstractFactory {
protected:
    virtual T* doCreate(T*) = 0;
    typedef T ProductList;
};

template <class T1, class T2>
class AbstractFactory< Typelist<T1, T2> > :
    public AbstractFactory<T1>,
    public AbstractFactory<T2> {

protected:
    using AbstractFactory<T1>::doCreate;
    using AbstractFactory<T2>::doCreate;

    typedef Typelist<T1, T2> ProductList;
};
```

This echoes the recursive approach to calculating a typelist's length. Figure 1 shows the hierarchy that results from the declaration

```
typedef AbstractFactory <
    TYPELIST_3(Button, ScrollBar, Menu)
> WidgetFactory;
```

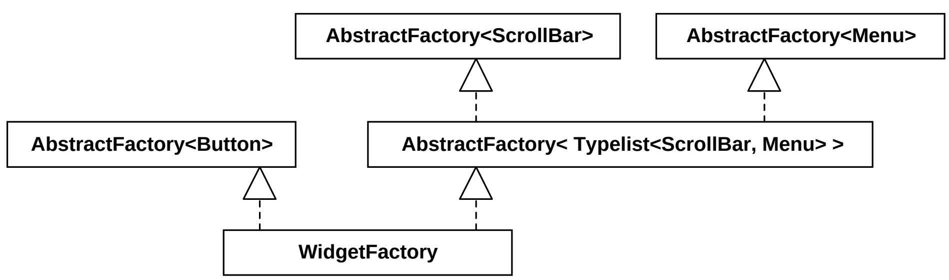

Figure 1: `WidgetFactory` interface hierarchy

`WidgetFactory` inherits directly from `AbstractFactory<Button>` and indirectly from both `AbstractFactory<ScrollBar>` and `AbstractFactory<Menu>` through a *node class*, namely `AbstractFactory< Typelist<ScrollBar, Menu> >`. The node class acts like a funnel, collecting and propagating operations down the hierarchy. In the end, an `AbstractFactory` instantiated with a typelist will inherit from an instantiation of `AbstractFactory` for every type in that typelist—the point of the exercise.

Now to implement this `WidgetFactory` interface for a given ConcreteFactory name. Last time we defined `MacWidgetFactory` like this:

```
typedef ConcreteFactory <
    WidgetFactory,
    TYPELIST_3(MacButton, MacScrollBar, MacMenu)
> MacWidgetFactory;
```

The `WidgetFactory` operations get implemented one at a time, again using template recursion. There are two specializations of `ConcreteFactory`: one for a single ConcreteProduct type, and one for a typelist of ConcreteProduct types. The former looks like this:

```
template <class AbstFact, class ConcProd>
class ConcreteFactory : public AbstFact {

    typedef AbstFact::ProductList ProductList;
    typedef LastType<ProductList>::Type Product;

protected:
    using AbstFact::doCreate;

    virtual Product* doCreate (Product*) {
        return new ConcProd;
    }
};
```

This template implements `doCreate` for the *last* element in `AbstFact`'s product list. That element, obtained with GPI's `LastType` operation, becomes `doCreate`'s return type (corresponding to the AbstractProduct participant in the pattern). Of course, what `doCreate` actually returns is a new ConcreteProduct of type `ConcProd`.

Now let's see how the other `ConcreteFactory` template works, the one specialized for a typelist of ConcreteProducts. Basically it implements `doCreate` for the head of the list and recurses to its tail through inheritance.

```
template <
    class AbstFact,
    class ConcProd,
    class OtherConcProds

> class ConcreteFactory <
    AbstFact,
    TYPELIST_2(ConcProd, OtherConcProds)

> : public ConcreteFactory<AbstFact, OtherConcProds> {

protected:
    typedef typename AbstFact::ProductList ProductList;

    enum {
        index = Length<ProductList>::Value -
            Length<OtherConcProds>::Value - 1
    };

    typedef typename TypeAt<ProductList, index>::Type Product;

    using ConcreteFactory<AbstFact, OtherConcProds>::doCreate;

    virtual Product* doCreate (Product*) {
        return new ConcProd;
    }
};
```

Don't get excited—this is simpler than it looks. On each recursion, the `enum` gets evaluated first, computing the index of the next product in the `AbstractFactory`'s typelist. Then the `Product typedef` gets evaluated, identifying the AbstractProduct at that index using GPI's `TypeAt` operation. With the AbstractProduct type in hand, the compiler implements `doCreate` for it like before.

When the dust settles, we're left with a simple, linear inheritance structure: each class in Figure 2 contributes one `doCreate` operation as the hierarchy is built up and the typelist consumed.

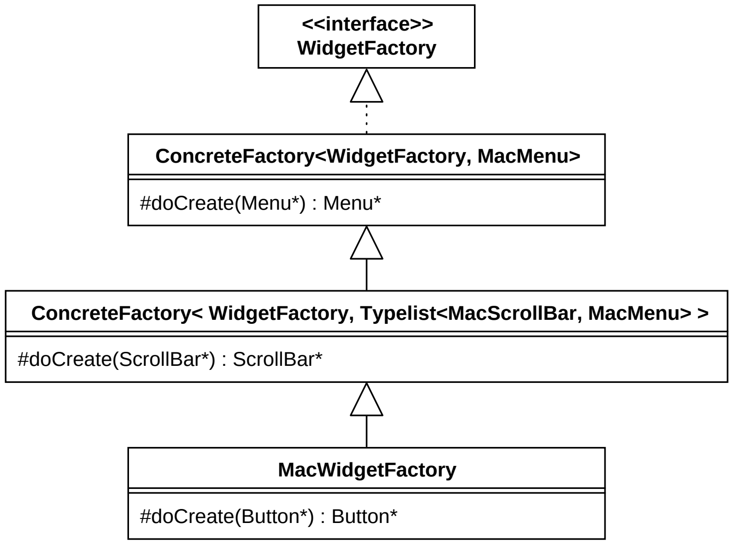

Figure 2: `MacWidgetFactory` *implementation hierarchy*

All we need now is a decent interface for clients, one that hides `doCreate` behind something akin to the factory methods in a standard AbstractFactory. Why isn't `doCreate` decent to begin with? Apart from its intentionally arcane name, it's saddled with an argument that's never used. The argument's sole purpose is to let the compiler differentiate between all the overloaded versions of `doCreate`. If there were no such parameter, each subclass would override the same parameterless `doCreate`. `MacWidgetFactory` would wind up with just one `doCreate` rather than one for *each* AbstractProduct (in this case `Button`, `ScrollBar`, and `Menu`).

Recall that clients use a function template to create a button,

```
Button* btn = widgetFactory->create<Button>();
```

rather than the usual hard-wired function:

```
Button* btn = widgetFactory->createButton();
```

Here's the template that does the trick:

```
template <class T> T* ConcreteFactory::create () {
    return doCreate(static_cast<T*>(0));
}
```

Template parameters as design choices

GPI employs template parameters to let you choose among a pattern's variant implementations and trade-offs. Last time you saw how `enum`s lend flexibility to the `Singleton` template:

```
enum Allocation { staticStorage, dynamicStorage };

enum Lifetime {
    stdLifetime, phoenix, varLifetime, immortal
};

enum ThreadingModel { singleThreaded, multiThreaded };
```

```
template <
    class T,
    Allocation = staticStorage,
    Lifetime = stdLifetime,
    ThreadingModel = singleThreaded
> class Singleton;
```

How are these values interpreted? Ideally there would be no run-time overhead for handling different cases, and clients could add their own choices noninvasively. That pretty much rules out a brute-force approach using conditionals:

```
template < ... > T* Singleton< ... >::instance() {
    if (threadingModel == singleThreaded){
        // single-threaded behavior

    } else {
        // multi-threaded behavior

    }
}
```

GPI has a better idea. Consider:

```
class AnyDesignChoice {};

template <long ChoiceID>
class DesignChoice : public AnyDesignChoice {};
```

A `DesignChoice` instantiated with any integral value is an `AnyDesignChoice`. `DesignChoice` instantiations with different values will be distinct types.

Now add a dash of overloading. The `Singleton` template should use different allocation strategies based on the value of the `Allocation` template parameter. GPI defines private `doCreate` helper functions[†] to encapsulate different allocation strategies at compile-time.

```
template <
    class T,
    Allocation alloc = staticStorage,
    Lifetime lifetime = stdLifetime,
    ThreadingModel threadingModel = singleThreaded

> class Singleton {
    // ...

private:
    static T* doCreate (DesignChoice<staticStorage>) {
        static T instance;
        return &instance;
    }

    static T* doCreate (AnyDesignChoice) {
        return new T;
    }
};
```

No rocket science here. If a `Singleton` operation calls `doCreate` with `DesignChoice<staticStorage>()` as a parameter, it'll get the address of a statically allocated object. Any other parameter (as long as it's type-compatible with `AnyDesignChoice`) will produce a dynamically

[†] Not to be confused with the `ConcreteFactory` templates' `doCreate` operations. Same name, different pattern.

allocated object. If an operation calls doCreate(DesignChoice<alloc>()), then what you get depends on the value of alloc, the Allocation template parameter.

Think of this as compile-time dispatch. The actual parameters are not used when the program runs; the code for creating them and passing them around will be optimized away by any reasonable compiler. They exist solely to choose between overloaded operations at compile-time, statically mapping an enumerated value to a behavior.

We've got a lot of flexibility here. An overloaded function that takes an AnyDesignChoice acts as a catch-all for default behavior. That's useful when the number of specialized behaviors is small compared to the common case. Consider Lifetime, which defines four distinct behaviors. The designer may specialize one or two of them and let the catch-all do the rest.

Singleton defines four primitive operations in support of the Allocation, Lifetime, and ThreadingModel design choices:

- doCreate tracks the Allocation strategy. Its sole purpose is to create an object. You can specialize this function to use a custom allocator or to create an object of a derived class. Here's an example:

```
typedef Singleton<Foo, ...> FooSingleton;

template <>
Foo* FooSingleton::doCreate (AnyDesignChoice) {
    // custom allocator implementation
}
```

- atomicCreate focuses on threading issues, tracking the ThreadingModel parameter. It tests to see if the singleton is allocated, employing the DOUBLE-CHECKED LOCKING pattern[4] in the multi-threaded case. After the test, atomicCreate delegates to doCreate(DesignChoice<alloc>()).
- scheduleDestruction schedules the object's destruction according to the Lifetime parameter.
- onDeadReference governs the behavior of instance() after the singleton has been destroyed. There are just two specializations: the phoenix case, which recreates the singleton, and all other cases, which throw an exception.

The instance() operation orchestrates these primitives, calling them with the appropriate DesignChoice template parameters. The compiler will instantiate and dispatch to the corresponding specializations. The result is both flexible and minimalist, extensible yet efficient, with the compiler doing most of the work.

Inheritance versus templates

One last thing before we turn you loose to try GPI yourself. If you're prepared to believe that SINGLETON is worth implementing in a reusable fashion (and shame on you if you're not), then you've got a problem. Before you can decouple SINGLETON functionality from clients that act like Singletons, you must decide how that functionality will cooperate with the clients.

C++ offers two relevant decoupling mechanisms: inheritance and templates. You can apply them in at least four ways to get a plausibly reusable implementation of SINGLETON. Let's assume a Singleton class encapsulates SINGLETON pattern functionality, and MyClass should play the Singleton role. Then the four possibilities are:

1. A straight inheritance relationship: MyClass derives from Singleton.
2. A templatized Singleton<MyClass> deriving from MyClass.

3. MyClass derives from Singleton<MyClass> (in other words, apply Coplien's CURIOUSLY RECURRING TEMPLATE pattern[5]).
4. Singleton<MyClass> is a stand-alone class with a reference to the lone MyClass instance.

All four require coding MyClass to prevent direct instantiation by clients; none of the four offers an advantage in that respect. However, all but one has a distinct disadvantage: interference with the client class hierarchy. Introducing a Singleton (template) class complicates the hierarchy, tightly coupling the pattern implementation to the client type structure.

Only one alternative avoids that outcome—number 4, which turns client classes into Singletons with templates rather than inheritance. It exemplifies favoring composition over inheritance, but not exactly as *Design Patterns* exhorts,[6] because we're composing types, not objects. Yet the rationale and benefits are much the same. Avoiding encapsulation-busting inheritance makes the pure-template approach the best-decoupled of the lot. Needless to say, GPI's Singleton is an independent template class.

Tip of the GPIceberg

There's a lot more to GPI than we've been able to present in two short columns. But we won't leave you high and dry. Andrei will take up these and other marvels of GPI now that he has a column of his own. Be sure to follow it every other month right here in *C++ Report*. Tell him John sent ya.

Acknowledgments

Thanks to Jim Coplien, Ralph Johnson, Scott Meyers, and Dirk Riehle for their sage advice.

References

[1] Vlissides, J. and A. Alexandrescu. To Code or Not to Code, Part I. *C++ Report*, March 2000, pp. ??–??.
[2] Gamma, et al. *Design Patterns*, Addison–Wesley, Reading, MA, 1995, pp. 87–95.
[3] Alexandrescu, A. *Design with C++* (tentative title), Addison–Wesley, Reading, MA, in preparation.
[4] Schmidt, D., et al. Double-Checked Locking. In *Pattern Languages of Program Design 3*, Addison–Wesley, Reading, MA, 1998, pp. 363–375.
[5] Coplien, J. Curiously Recurring Template Patterns. *C++ Report*, February 1995, pp. 24–27.
[6] *Design Patterns*, p. 20.