

## Pattern Hatching

# To Code or Not to Code, Part I

John Vlissides and Andrei Alexandrescu

*C++ Report*, March 2000

© 2000 by John Vlissides and Andrei Alexandrescu. All rights reserved.

Allow me to introduce Andrei Alexandrescu. We met after a mutual acquaintance, one Scott Meyers, had sent us e-mail that attempted to show “how patterns, classical algorithms and data structures, and [work like Andrei’s] seem to dovetail in a way that offers benefits to people working in each of those fields.” The details of that message and the ensuing thread are pretty interesting. I hope Scott sees fit to publish it someday.

What really struck me, though, was Andrei’s work,<sup>1</sup> both for its relationship to code generation (something I’ve dabbled in<sup>2,3</sup>) and for its potential impact on pattern application. Nay, I found Andrei’s stuff so intriguing and germane that I immediately felt an urge to give it a column—or two, as it turns out. But it would be impolite to share these goodies with you apart from their inventor, and I couldn’t do them justice anyway. So I am both honored and relieved to co-author these columns with Andrei. Consider them his; I’m just kibitzing.

## Talkin’ ’bout code generation

The idea of a tool producing code automatically enthralls many developers, for good reason: Why crank out code yourself when a machine can do it for you? The more code you don’t write, the easier your life gets, and the fewer bugs you’ll get blamed for. That’s not to say generated code is always perfect—generators are software too, you know. But owing to their mission as system tools, there’s incentive to lavish lots of time and effort on getting them right. Generators and the code they generate are thus far less likely to harbor flaws than code that’s fresh from your fingertips.

Still, all is not sweetness and light. Despite allegations that certain tools can reduce software development to the push of a button, in reality a tool hasn’t been devised that can write every jot and tittle of code for you, certainly not if you’re building industrial-strength software. Pre-packaged and toy examples abound, but real-world applications—those that truly add value—inevitably require the handwritten stuff.

And that’s okay, with certain caveats:

1. *You’re sure you won’t have to modify generated code by hand.* Whenever you generate code, you pave the way for the so-called *round-trip problem*: regenerating the code overwrites modifications to the previous generation. This is primarily a maintenance issue. It won’t hit you right away, and when it does, it causes difficulty only when generated and handcrafted code aren’t well decoupled. Trouble is, that’s more the rule than the exception.
2. *Generated code should outweigh handwritten code.* Like most forms of automation, code generation incurs overhead. Generated code usually isn’t as tight or fast as can be achieved by hand. It’s also not as flexible, because you don’t have as much control over it. And unless you can rely on the generator to implement 100% of the code, generation ends up adding steps to the programming process. You have to amortize such inefficiencies by generating as much useful code as possible. The lower the percentage of generated code, the harder it is to justify its costs.

Alas, these conditions aren’t always easy to meet. Decoupling is a problem in general, and many approaches have been devised specifically for code generation. These include everything from generators that pepper their output with comments about what should and shouldn’t be modified, to tools that maintain links between the generated code and the specification from which it’s generated (e.g., UML), to patterns such as

GENERATION GAP<sup>4</sup> that explain how to refashion your class structure to enforce decoupling. All have drawbacks that can make code generation more trouble than it's worth.

Then there's the problem of not having enough code to generate. It may not seem problematic, but it can actually be a showstopper, especially when you're generating code for design patterns.

Once your average bright developer discovers design patterns, it isn't long before he or she starts thinking about automating their use. "A code generator would be just the ticket," they'll conclude. But most design pattern implementations don't have much application-independent code; the patterns prescribe far less code than you'll end up writing. Take for example OBSERVER, one of the more complex GoF patterns.<sup>5</sup> Chances are you'll put more application-specific code in the `ConcreteObserver` `update` methods than in all application-independent methods (`attach`, `detach`, etc.) combined. Simpler patterns such as PROTOTYPE and TEMPLATE METHOD have implementations that are almost entirely application-dependent. There isn't much for a code generator to do but complicate things.

Generating useful code for patterns is tougher than general-purpose code generation for a couple of other reasons:

1. It's hard to generate code that's as flexible as a pattern is. The generator can be designed to cover the trade-offs and variants that are explicit in the pattern, but it can't vary far from them.
2. Generated code is often difficult to integrate with existing code, particularly when they're in a language that lacks multiple inheritance. This is a serious drawback when you're trying to implement compound patterns such as PLUGGABLE FACTORY<sup>6,7</sup> and TOOLED COMPOSITE<sup>8</sup> by combining code generated for their constituent patterns.

If implementations generally don't embody a lot of pattern-specific code, then that's all the more code to write manually. A pattern code generator hasn't helped you much—perhaps not enough to pull its weight. And if you look at independent implementations of the same pattern, they'll tend to be more alike than different. It would be nice to avoid *all* code duplication, even if you're duplicating only bits and pieces.

Indeed, fine-grained duplication often pervades software. When the repeated code is small enough, it can go unnoticed in refactoring phases, or it may be deemed too small to worry about. So the redundancy proliferates, and it's surprisingly hard to rein in. Design pattern code is like that, whether generated or not.

Wouldn't it be great if we could realize the benefits of code generation—quicker, easier development; reduced redundancy; fewer bugs—without the drawbacks? That's what Andrei's approach promises.

He calls it "GPI," short for Generic Pattern Implementations. GPI uses C++ templates and generic programming techniques to capture good pattern implementations in an easy-to-use, mixable-and-matchable form. The templates do pretty much what code generators do: produce boilerplate code for compiler consumption. The difference is they do it *within* the language, not apart from it. The result is seamless integration with application code. You can also use the full power of the language to extend, override, and otherwise manipulate GPI facilities.

## ABSTRACT FACTORY à la Andrei

Let's look at how to implement ABSTRACT FACTORY<sup>9</sup> with GPI. You declare an `AbstractFactory` class like so:

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

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

This declaration reveals an important GPI technique: **typelists**. We'll describe them in detail next time. For now, it suffices to know that typelists let you manipulate collections of types at compile-time much like you

manipulate collections of values at run-time. Typelists are crucial for design patterns that deal with families of types—ABSTRACT FACTORY and VISITOR in particular.

All we've defined so far is the AbstractFactory participant of the pattern, which in this application defines an interface for creating buttons, scrollbars, and menus without specifying concrete subclasses. Those subclasses have to come from somewhere. Here's how you would define a couple of concrete WidgetFactory subclasses:

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

typedef ConcreteFactory <
    WidgetFactory,
    TYPELIST_3(WindowsButton, WindowsScrollBar, WindowsMenu)
> WindowsWidgetFactory;
```

These aren't just declarations; they institute full *implementations* of the concrete factories as prescribed by the pattern. MacWidgetFactory and WindowsWidgetFactory work just like their conventional handwritten counterparts—with one minor difference. As usual, you declare the factory and initialize it sometime later, when your program knows which kind of look and feel to adopt:

```
extern WidgetFactory* widgetFactory;
// ...
widgetFactory = new MacWidgetFactory;
```

Now you can put the factory to work. But instead of a hard-wired operation for each product class, e.g.,

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

the GPI implementation uses a function template to specify the type of product. A client specifies a type from the typelist supplied to the WidgetFactory declaration:

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

The compiler will complain bitterly if you ask for anything outside the typelist, as it should. To create a scrollbar or menu, just say so with the corresponding template parameter:

```
ScrollBar* sb = widgetFactory->create<ScrollBar>();
Menu* menu = widgetFactory->create<Menu>();
```

## Infusing SINGLETON

A global widgetFactory is rather gauche. We might well prefer to make WidgetFactory a Singleton.

GPI lets you declare a Singleton of type Foo by supplying Foo as a parameter to a Singleton template:

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

You instantiate Foo through a static FooSingleton::instance member function:\*

```
Foo& f = FooSingleton::instance();
```

The Singleton template provides a vanilla implementation of the pattern by default. Foo will be instantiated just once, the first time instance gets called, and instance will return that same instance forever

---

\* This isn't quite in keeping with the SINGLETON pattern, since FooSingleton::instance returns a Foo& as opposed to a FooSingleton\*. The latter is inappropriate because FooSingleton is a *container* of type Foo, not a base or derived class thereof. While establishing an inheritance relationship between Foo and FooSingleton is certainly possible with GPI, it offers no significant benefits over this pure-container approach, and it's more invasive and less flexible to boot. We'll delve further into this issue next time.

after. The template shields you from the implementation details of `SINGLETON` and makes the pattern explicit in the code.

There's one thing the template doesn't do, however: it does nothing to prevent clients from instantiating `Foo` directly. If you want to preclude that, then you have to be proactive about it when you're defining `Foo`—protect its constructors, and make `FooSingleton` its friend.<sup>†</sup>

Declaring the `WidgetFactory` class a `Singleton` is a little more involved because it's an abstract class. The declaration itself is what you'd expect:

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

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

But you can't stop there, because you haven't specified the concrete class to instantiate. You do that with member function specialization:

```
template <T> WidgetFactory::Type& WidgetFactory::instance () {
    if (WidgetFactory::_instance == 0) {
        // decide which concrete subclass to instantiate
        // and assign an instance thereof to _instance
    }
    return *_instance;
}
```

`Type` is a typedef that the `Singleton` template defines as a shorthand for its argument, which in this case is `AbstractFactory<TYPELIST_3(...)>`.

Once you've defined how `instance` works, your `WidgetFactory` is ready to use:

```
Button* btn = WidgetFactory::instance().create<Button>();
```

Specializing `instance` lets us tweak `SINGLETON`'s implementation in unforeseeable ways. In fact, the `Singleton` class template defines and documents several primitive operations, any of which can be redefined through specialization. Clients thus have considerable control over a pattern's implementation.

We're getting ahead of ourselves, though. Call us lazy, but we think it's a pain to resort to such specialization for common variants of `SINGLETON`, including

- where to allocate the `Singleton` instance (static versus free store allocation)<sup>10</sup>
- when to destroy it<sup>11</sup>
- how to report an error in case the singleton gets accessed after it's been destroyed (the *dead reference* issue, largely ignored by most `SINGLETON` implementations)
- whether the singleton should be thread-safe<sup>12</sup>

The easier it is to implement these design choices, and the more clearly we convey them in the code, the better the implementation.

Template parameters are the knobs that adjust GPI for different needs. Each template parameter controls a *degree of freedom* in which an implementation can vary. A degree of freedom embodies a design decision, a constraint, or a trade-off in the pattern that may vary independently of others. Such orthogonality promotes

<sup>†</sup> Dirk Riehle recommends against using `friend`. Instead, the `Singleton` template could define a trivial subclass of `Foo` as a nested class, for the sole purpose of making the subclass constructors public. That gives the template exclusive rights to instantiating the subclass without resorting to friendship. All a client must do is protect `Foo`'s constructors. Andrei's implementation doesn't do this—yet.

expressiveness: a small amount of template code can support many implementation variants in a consistent, extensible, bug-resistant, and maintainable fashion.

Suppose `WidgetFactory` should be allocated on the free store, should never be deleted, and should work reliably (albeit with overhead) in a multithreaded application. You can say all that with template parameters:

```
typedef Singleton <
    AbstractFactory<TYPELIST_3(Button, ScrollBar, Menu)>,
    dynamicStorage,          // use dynamic allocation
    immortal,                // will never be destroyed
    multiThreaded           // supports multithreading
> WidgetFactory;
```

Where do the parameters come from? Here's an excerpt from the `GPI SINGLETON` implementation:

```
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;
```

Controlling a pattern's degrees of freedom through template parameters is key to GPI's utility. Chosen well, these parameters let you handle a combinatorial number of situations with a linear amount of code. A programmer implements just the parts that are unique to the problem at hand; template arguments govern the common design trade-offs and variations.

That's essentially what we wanted from the code generation approach: "Code the boring parts for me so I can focus on the interesting stuff." But GPI does it under the aegis of the programming language, not through added tooling. You reap the benefits of code generation without the drawbacks—although to be fair, GPI has drawbacks of its own:

- *Its templates are complex, breaking many template-challenged compilers.* Vendors are constantly improving their compilers, however, so this is a short-term problem.
- *Be prepared for unenlightening error messages.* Compilers are at a real disadvantage when they come across a mistake in GPI code. They know nothing of the pattern behind a template, nor can we reasonably expect them to. So don't be surprised by error messages that are incomprehensible at first.
- *Your mileage may vary.* Because template parameters cover only variations their author knows about, you may have to do more than your share of specialization, subclassing, and overloading. If so, please let Andrei know. He's always looking for common variants he's missed.
- *It takes getting used to.* Like STL, GPI puts a new spin on your code, and it might induce vertigo. Getting over it takes time, but it's time well spent if GPI helps you reduce redundancy, leverage proven implementations, and make the patterns in your code more explicit.

## Stay tuned

The next installment will offer a more detailed view of the generic programming techniques used with GPI. In particular, we'll take a look under the hood of the `AbstractFactory` and `Singleton` templates. We'll see more ways to specialize their behavior and to compound them with other patterns.

## Corrections

The November/December '99 installment had a couple bugs:

1. Allusions to `Visitor::accept` and `AppVisitor::accept` in the paragraph just before the “Back to the Drawing Board” section should of course be `Element::accept` and `AppElement::accept`.
2. Please ignore spurious voids in the return values of manipulate functions.

A thousand sorries!

## Acknowledgments

Dirk Riehle and Scott Meyers provided excellent feedback.

## References

- <sup>1</sup> Alexandrescu, A. *Design with C++* (tentative title), Addison–Wesley, Reading, MA, 2001.
- <sup>2</sup> Vlissides, J., et al. A Unidraw-based user interface builder. *Proceedings of the ACM SIGGRAPH Fourth Annual Symposium on User Interface Software and Technology*, Hilton Head, SC, November 1991, pp. 201–210.
- <sup>3</sup> Budinsky, F., et al. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996. <http://www.almaden.ibm.com/journal/sj/budin/budinsky.html>.
- <sup>4</sup> Vlissides, J. *Pattern Hatching*, Addison–Wesley, Reading, MA, 1998, pp. 85–101.
- <sup>5</sup> Gamma, E., et al. *Design Patterns*, Addison–Wesley, Reading, MA, 1995.
- <sup>6</sup> Vlissides, J. PLUGGABLE FACTORY, part I. *C++ Report*, November/December 1998, pp. 52–56, 68.
- <sup>7</sup> Vlissides, J. PLUGGABLE FACTORY, part II. *C++ Report*, February 1999, pp. 51–57.
- <sup>8</sup> Vlissides, J. TOOLED COMPOSITE. *C++ Report*, September 1999, pp. 43–47.
- <sup>9</sup> *Design Patterns*, pp. 87–95.
- <sup>10</sup> Meyers, S. *More Effective C++*, Addison–Wesley, Reading, MA, 1996, pp. 130–145.
- <sup>11</sup> *Pattern Hatching*, pp. 61–69.
- <sup>12</sup> Schmidt, D., et al. Double-Checked Locking. In *Pattern Languages of Program Design 3*, Addison–Wesley, Reading, MA, 1998, pp. 363–375.