IEEE Software
www.computer.org/software

Designing Extensible Classes

Rebecca J. Wirfs-Brock

Vol. 24, No. 5
September/October 2007

Editor: Rebecca J. Wirfs-Brock ■ Wirfs-Brock Associates ■ rebecca@wirfs-brock.com

Designing Extensible Classes

Rebecca J. Wirfs-Brock

When you have great songs that are going to live longer than the composers, everything you can do to bring those different elements and nuances out serves the song. —Michael Bolton

I know a designer who doesn't appreciate his colleagues' long-winded debates about correct use of interfaces and inheritance. Enough debate and hair splitting! He seeks design rules that guarantee well-formed class hierarchies, appropriate interfaces, and easily extensible classes. Unfortunately, the detailed design of extensible class hierarchies is filled with nuanced reasoning.

Nuanced reasoning

When I teach how to design class hierarchies, I illustrate how abstract classes can define default behaviors and implement configurable algorithms. I show how to use the Template Method pattern to define a configurable algorithm with replaceable steps. The mental trick is to decompose an operation into a sequence of substeps implemented by helper methods. At your discretion, you can provide default implementations for particular substeps and fixed (unchangeable) implementations for others. Most static, object-oriented languages—including C++ and Java—offer designers several choices for defining a method: Do you want the method to be visible to clients and to subclasses? Should it provide an implementation? If so, can subclasses redefine the definition?

Building clean abstractions with clearly defined extension points is satisfying, but the best design choice isn't always obvious. How much access should you give a subclass to a class's inner workings? How much freedom should you give a subclass designer to "bend" inherited behaviors to make a new abstraction fit in or to extend an existing one? These decisions involve nuanced reasoning. The contract between a class and its subclasses requires thoughtful design, experimentation, and careful specification.

Constraints: Avoid extremes

Kryzsztof Cwalina and Brad Adams, two designers of the .NET framework, advise, "Do not make members virtual [overridable] unless you have a good reason to do so and you are aware of all the costs related to designing, testing, and maintaining virtual members" (*Framework Design Guidelines*, Addison-Wesley, 2005). They further propose "limiting extensibility to only that absolutely necessary through use of the template method." So you shouldn't provide overridable methods unless a template method invokes them, and you shouldn't let subclass designers override template methods. So, in most cases, public methods can't be overridden.

As a former framework designer, I understand this reasoning—abstract algorithms embodied in a template method are hard to get right. Once

0740-7459/07/$25.00 © 2007 IEEE

you've taken the time to specify a configurable algorithm, you should only let subclass designers change prescribed substeps in a tightly controlled manner. But Cwalina and Adams go one step further: they suggest precluding any public method from being overridden.

This seems overly restrictive. I realize that limiting access to a class's methods lets class designers conceal implementation details, so they can change them with little or no impact on the class's users. However, taking this to an extreme can be a problem for the subclass designer. It isn't always clear whether something is an implementation detail or an important, albeit nuanced, behavior that a subclass designer needs to slightly adjust to make his or her class work.

When extending a class, I might need to add on explicit behavior to the inherited behavior. When you expose any interior design detail of a class, you let subclass designers make subtle shifts in implemented behavior (whether or not these methods are public). These subtle adjustments let the subclass designer exploit behaviors you've provided and tune them to their specific context.

Admittedly, a subclass designer must seriously study and experiment with a class before he or she can extend it in this way. It's easier—and perhaps safer—if the designer declares only the barest amount of behavior that a subclass designer can alter.

The need for freedom

As a framework consumer, I'm often unhappy with such rigid class definitions. I prefer having the freedom to tinker over promises of safe-to-use but immutable definitions. There are times when I might need to change default (inherited) behaviors. When you lock a class' implementation to prevent mistakes (so I can't break your pretty abstraction), you also preclude me from making any necessary changes. If you know you don't want me readjusting the ordering of steps in a template method, by all means declare that order immutable. But if you're merely suggesting a default ordering, or providing reasonable default behavior in a method (whether it's part of an abstract algorithm or not), let me change and adjust those definitions.

I prefer having the freedom to tinker over promises of safe-to-use but immutable definitions.

Otherwise, I have to resort to workarounds. I might have to reimplement my own equivalent classes or wrap your immutable ones in mine and write mindless code that delegates to your classes. The end result is always more complicated. I've also had to create classes that aren't extensions to yours. Unless you've specified an interface that my classes can also implement, my classes can't readily plug in to your preexisting frameworks. This might protect the framework designer in a business or legal sense, but it isn't helpful to me as a sophisticated framework consumer.

It seems natural to allow subclass designers to change inherited behaviors—at their own peril, of course, but also to suit their own purpose. This bias toward openness comes from my Smalltalk programming roots. In Smalltalk, as well as other dynamically typed languages such as Ruby, class hierarchies are open, extensible, and thus, to some extent, fragile.

Challenges

In "Issues in the Design and Specification of Class Libraries," Gregor Kiczales and John Lamping wrote about the challenges framework designers face (*Proc. OOPSLA 92*, ACM Press, 1992). Inherited code can break when a subclass implements an inherited method that violates the implicit constraints a superclass requires. Indeed, a class offers two contracts—one for users of its publicly defined interface and another for subclasses that can modify and configure the class's operations and data members. The latter contract is especially difficult to define programmatically, so extending a class correctly can be surprisingly difficult.

Kiczales and Lamping introduced terminology that lets framework designers distinguish between changeable and fixed class and method specifications. A *specified definition* for a class or method is listed in a library specification. An *implementation-specific definition* is present but doesn't appear in the specification. As these are susceptible to change and revision, they don't recommend extending or using them. A framework user defines a portable implementation if he or she relies only on framework classes and methods that are in the library specification.

Kiczales and Lamping also propose rules for letting subclass designers make safe extensions. Their paper was filled with nuanced discussions. For example, what happens when a set of specified methods that are defined in a library interact with each other? Should you let subclass implementers modify these methods individually or only as a group? Kiczales and Lamping assert that when designing methods that work in con-

cert, rules for overriding need to also be explicitly documented. In their opinion, you should force subclass designers to provide new implementations for all interacting methods in an interaction set.

According to Kiczales and Lamping,

> *As part of learning how to specify extensible class libraries, we must learn a new sense of the distinction between "implementation details" and crucial parts of the specification.... We still want to hide things that truly are implementation specific... it is just that we need to say more about the internal structure than we used to.*

If you open up a class's interior design, you must specify details about method interdependencies, invariants to be preserved between calls, and state changes that must be made at particular points in an algorithm. Otherwise, subclass designers might violate these constraints. Unfortunately, most framework designers don't specify these details because it requires a lot of work. Furthermore, modern programming languages don't let you clearly specify every nuance. So framework designers, if they bother, must document design specifications through a variety of means: method declarations, comments, assertions, and working examples of correct extensions.

But most of us don't invent frameworks or class libraries for unknowing consumers. Our classes are part of a subsystem, component, or application for which we know the context of its use. Consumers of our classes are likely to be fellow developers working in our same company, if not in the same physical location. And we're most likely building classes whose evolution and extension we'll continue to have some control over. This should make our job easier, but we still need to take the time to understand what constitutes a well-defined inheritance hierarchy and specify intended extension points.

Open to change

Most designers don't need to be as restrictive as framework designers when it comes to limiting extensions. For most classes, it's reasonable to make certain methods extensible, but you shouldn't doggedly follow the same conventions everywhere. Some classes should rarely change. If they aren't intended to be open, don't confuse your teammates by leaving everything in those classes public, visible, and open for extension.

Other classes, more naturally, are designed for extension. Sorting out the nuances of classes at an inheritance hierarchy's base requires experimentation and time. Consider letting subclasses make extensive changes. But don't let them modify something you view as fixed and immutable. But be open to change. As you (or others) create new subclasses, you might discover ways to improve your initial abstractions and are likely to refactor your initial design as you incorporate what you've learned.

Some developers declare every method visible and changeable to avoid making any decision that they might later need to reverse. This is okay when working in a small, tight team, where everyone is intimately familiar with each detail. But too much openness can be frustrating. If you don't state how you intend a class to be used or extended, a newcomer must make educated guesses or ask an expert. My first choice is always to ask the initial designer about his or her intentions. But I usually also have to perform experiments and diligently read others' code, hoping that their code illustrates good use. Without any concern for encapsulating implementation details, the details can easily leak out and become tangled with other code. Too much freedom isn't good either.

A responsible class designer must strike a balance between openness, clarity, safety, and ease of making extensions, which isn't always easy the first time around. Yesterday's decision might need rethinking to meet tomorrow's needs. The stability of a class's public interface might increase with use and extension, but its interior design details are likely to require rework. This doesn't mean we should throw up our hands and declare a free-for-all or preclude any extension. Anything we can do to make evident our understanding of our classes' inner workings will only enhance subclass designers' ability to grow and evolve our designs and make their own nuanced decisions.

If you don't state how you intend a class to be used or extended, a newcomer must make educated guesses or ask an expert.

Rebecca J. Wirfs-Brock is president of Wirfs-Brock Associates. Contact her at rebecca@wirfs-brock.com; www.wirfs-brock.com.