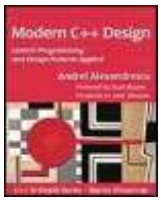

Modern C++ Design: Generic Programming and Design Patterns Applied

By Andrei Alexandrescu

Publisher: Addison Wesley
Pub Date: February 01, 2001
ISBN: 0-201-70431-5
Pages: 352

Table of Contents

Modern C++ Design is an important book. Fundamentally, it demonstrates 'generic patterns' or 'pattern templates' as a powerful new way of creating extensible designs in C++--a new way to combine templates and patterns that you may never have dreamt was possible, but is. If your work involves C++ design and coding, you should read this book. Highly recommended.-Herb Sutter

What's left to say about C++ that hasn't already been said? Plenty, it turns out.-From the Foreword by John Vlissides

In *Modern C++ Design*, Andrei Alexandrescu opens new vistas for C++ programmers. Displaying extraordinary creativity and programming virtuosity, Alexandrescu offers a cutting-edge approach to design that unites design patterns, generic programming, and C++, enabling programmers to achieve expressive, flexible, and highly reusable code.

This book introduces the concept of generic components-reusable design templates that produce boilerplate code for compiler consumption-all within C++. Generic components enable an easier and more seamless transition from design to application code, generate code that better expresses the original design intention, and support the reuse of design structures with minimal recoding.

The author describes the specific C++ techniques and features that are used in building generic components and goes on to implement industrial strength generic components for real-world applications. Recurring issues that C++ developers face in their day-to-day activity are discussed in depth and implemented in a generic way. These include:

- Policy-based design for flexibility
- Partial template specialization
- Typelists-powerful type manipulation structures
- Patterns such as Visitor, Singleton, Command, and Factories
- Multi-method engines

For each generic component, the book presents the fundamental problems and design options, and finally implements a generic solution.

In addition, an accompanying Web site, http://www.awl.com/cseng/titles/0-201-70431-5, makes the code implementations available for the generic components in the book and provides a free, downloadable C++ library, called Loki, created by the author. Loki provides out-of-the-box functionality for virtually any C++ project.

Table of Content

Copyright

Library of Congress Cataloging-in-Publication Data

Alexandrescu, Andrei.

Modern C++ design : generic programming and design patterns applied / Andrei Alexandrescu.

p. cm. — (C++ in depth series)

Includes bibliographical references and index.

ISBN 0-201-70431-5

1. C++ (Computer program language) 2. Generic programming (Computer science) I. Title. II. Series.

QA76.73.C153 A42 2001

005.13'3—dc21 00-049596

2 3 4 5 6 7 8 9 10—DOC—0504030201

Second printing, June 2001

Foreword

by Scott Meyers

In 1991, I wrote the first edition of *Effective C++*. The book contained almost no discussions of templates, because templates were such a recent addition to the language, I knew almost nothing about them. What little template code I included, I had verified by e-mailing it to other people, because none of the compilers to which I had access offered support for templates.

In 1995, I wrote *More Effective C++*. Again I wrote almost nothing about templates. What stopped me this time was neither a lack of knowledge of templates (my initial outline for the book included an entire chapter on the topic) nor shortcomings on the part of my compilers. Instead, it was a suspicion that the C++ community's understanding of templates was about to undergo such dramatic change, anything I had to say about them would soon be considered trite, superficial, or just plain wrong.

There were two reasons for that suspicion. The first was a column by John Barton and Lee Nackman in the January 1995 *C++ Report* that described how templates could be used to perform typesafe dimensional analysis with zero runtime cost. This was a problem I'd spent some time on myself, and I knew that many had searched for a solution, but none had succeeded. Barton and Nackman's revolutionary approach made me realize that templates were good for a lot more than just creating containers of `T`.

As an example of their design, consider this code for multiplying two physical quantities of arbitrary dimensional type:

```
template<int m1, int l1, int t1, int m2, int l2, int t2>
Physical<m1+m2, l1+l2, t1+t2> operator*(Physical<m1, l1, t1> lhs,
                                        Physical<m2, l2, t2> rhs)
{
   return Physical<m1+m2, l1+l2, t1+t2>::unit*lhs.value()*rhs.value();
}
```

Even without the context of the column to clarify this code, it's clear that this function template takes six parameters, none of which represents a type! This use of templates was such a revelation to me, I was positively giddy.

Shortly thereafter, I started reading about the STL. Alexander Stepanov's elegant li brary design, where containers know nothing about algorithms; algorithms know nothing about containers; iterators act like pointers (but may be objects instead); containers and algorithms accept function pointers and function objects with equal aplomb; and library clients may extend the library without having to inherit from any base classes or redefine any virtual functions, made me feel—as I had when I read Barton and Nackman's work—like I knew almost *nothing* about templates.

So I wrote almost nothing about them in *More Effective C++*. How could I? My understanding of templates was still at the containers-of-`T` stage, while Barton, Nackman, Stepanov, and others were demonstrating that such uses barely scratched the surface of what templates could do.

In 1998, Andrei Alexandrescu and I began an e-mail correspondence, and it was not long before I recognized that I was again about to modify my thinking about templates. Where Barton, Nackman, and

Stepanov had stunned me with what templates could *do,* however, Andrei's work initially made more of an impression on me for *how* it did what it did.

One of the simplest things he helped popularize continues to be the example I use when introducing people to his work. It's the `CTAssert` template, analogous in use to the `assert` macro, but applied to conditions that can be evaluated during compilation. Here it is:

```
template<bool> struct CTAssert;
template<> struct CTAssert<true> {};
```

That's it. Notice how the general template, `CTAssert,` is never defined. Notice how there is a specialization for `true,` but not for `false.` In this design, what's *missing* is at least as important as what's present. It makes you look at template code in a new way, because large portions of the "source code" are deliberately omitted. That's a very different way of thinking from the one most of us are used to. (In this book, Andrei discusses the more sophisticated `CompileTimeChecker` template instead of `CTAssert`.)

Eventually, Andrei turned his attention to the development of template-based implementations of popular language idioms and design patterns, especially the GoF[*] patterns. This led to a brief skirmish with the Patterns community, because one of their fundamental tenets is that patterns cannot be represented in code. Once it became clear that Andrei was automating the generation of pattern *implementations* rather than trying to encode patterns themselves, that objection was removed, and I was pleased to see Andrei and one of the GoF (John Vlissides) collaborate on two columns in the *C++ Report* focusing on Andrei's work.

[*] "GoF" stands for "Gang of Four" and refers to Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, authors of the definitive book on patterns, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).

In the course of developing the templates to generate idiom and pattern implementations, Andrei was forced to confront the variety of design decisions that all implementers face. Should the code be thread safe? Should auxiliary memory come from the heap, from the stack, or from a static pool? Should smart pointers be checked for nullness prior to dereferencing? What should happen during program shutdown if one Singleton's destructor tries to use another Singleton that's already been destroyed? Andrei's goal was to offer his clients all possible design choices while mandating none.

His solution was to encapsulate such decisions in the form of *policy classes,* to allow clients to pass policy classes as template parameters, and to provide reasonable default values for such classes so that most clients could ignore them. The results can be astonishing. For example, the Smart Pointer template in this book takes only 4 policy parameters, but it can generate over 300 different smart pointer types, each with unique behavioral characteristics! Programmers who are content with the default smart pointer behavior, however, can ignore the policy parameters, specify only the type of object pointed to by the smart pointer, and reap the benefits of a finely crafted smart pointer class with virtually no effort.

In the end, this book tells three different technical stories, each compelling in its own way. First, it offers new insights into the power and flexibility of C++ templates. (If the material on typelists doesn't knock your socks off, it's got to be because you're already barefoot.) Second, it identifies orthogonal dimensions along which idiom and pattern implementations may differ. This is critical information for template designers and pattern implementers, but you're unlikely to find this kind of analysis in most idiom or pattern descriptions. Finally, the source code to Loki (the template library described in this book) is available for free download, so you can study Andrei's implementation of the templates corresponding to the idioms and patterns he discusses. Aside from providing a nice stress test for your compilers' support for templates, this source code serves as an invaluable starting point for templates of your own design. Of course, it's also perfectly respectable (and completely legal) to use Andrei's code right out of the box. I know he'd want you to take advantage of his efforts.

From what I can tell, the template landscape is changing almost as quickly now as it was in 1995 when I decided to avoid writing about it. At the rate things continue to develop, I may *never* write about templates. Fortunately for all of us, some people are braver than I am. Andrei is one such pioneer. I think you'll get a lot out of his book. I did.

Scott Meyers
September 2000

Foreword

by John Vlissides

What's left to say about C++ that hasn't already been said? Plenty, it turns out. This book documents a convergence of programming techniques—generic programming, template meta programming, object-oriented programming, and design patterns—that are well understood in isolation but whose synergies are only beginning to be appreciated. These synergies have opened up whole new vistas for C++, not just for programming but for software design itself, with profound implications for software analysis and architecture as well.

Andrei's generic components raise the level of abstraction high enough to make C++ begin to look and feel like a design specification language. Unlike dedicated design languages, however, you retain the full expressiveness and familiarity of C++. Andrei shows you how to program in terms of design concepts: singletons, visitors, proxies, abstract factories, and more. You can even vary implementation trade-offs through template parameters, with positively no runtime overhead. And you don't have to blow big bucks on new development tools or learn reams of methodological mumbo jumbo. All you need is a trusty, late-model C++ compiler—and this book.

Code generators have held comparable promise for years, but my own research and practical experience have convinced me that, in the end, code generation doesn't compare. You have the round-trip problem, the not-enough-code-worth-generating problem, the inflexible-generator problem, the inscrutable-generated-code problem, and of course the I-can't-integrate-the-bloody-generated-code-with-my-own-code problem. Any one of these problems may be a showstopper; together, they make code generation an unlikely solution for most programming challenges.

Wouldn't it be great if we could realize the theoretical benefits of code generation—quicker, easier development, reduced redundancy, fewer bugs—without the drawbacks? That's what Andrei's approach promises. Generic components implement good designs in easy-to-use, mixable-and-matchable templates. They do pretty much what code generators do: produce boilerplate code for compiler consumption. The difference is that they do it within C++, 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 tweak the designs to suit your needs.

Some of the techniques herein are admittedly tricky to grasp, especially the template metaprogramming in Chapter 3. Once you've mastered that, however, you'll have a solid foundation for the edifice of generic componentry, which almost builds itself in the ensuing chapters. In fact, I would argue that the metaprogramming material of Chapter 3 alone is worth the book's price—and there are ten other chapters full of insights to profit from. "Ten" represents an order of magnitude. Even so, the return on your investment will be far greater.

John Vlissides
IBM T.J. Watson Research
September 2000

Preface

You might be holding this book in a bookstore, asking yourself whether you should buy it. Or maybe you are in your employer's library, wondering whether you should invest time in reading it. I know you don't have time, so I'll cut to the chase. If you have ever asked yourself how to write higher-level programs in C++, how to cope with the avalanche of irrelevant details that plague even the cleanest design, or how to build reusable components that you don't have to hack into each time you take them to your next application, then this book is for you.

Imagine the following scenario. You come from a design meeting with a couple of printed diagrams, scribbled with your annotations. Okay, the event type passed between these objects is not `char` anymore; it's `int`. You change one line of code. The smart pointers to `Widget` are too slow; they should go unchecked. You change one line of code. The object factory needs to support the new `Gadget` class just added by another department. You change one line of code.

You have changed the design. Compile. Link. Done.

Well, there is something wrong with this scenario, isn't there? A much more likely scenario is this: You come from the meeting in a hurry because you have a pile of work to do. You fire a global search. You perform surgery on code. You add code. You introduce bugs. You remove the bugs . . . that's the way a programmer's job is, right? Although this book cannot possibly promise you the first scenario, it is nonetheless a resolute step in that direction. It tries to present C++ as a newly discovered language for software architects.

Traditionally, code is the most detailed and intricate aspect of a software system. Historically, in spite of various levels of language support for design methodologies (such as object orientation), a significant gap has persisted between the blueprints of a program and its code because the code must take care of the ultimate details of the implementation and of many ancillary tasks. The intent of the design is, more often than not, dissolved in a sea of quirks.

This book presents a collection of reusable design artifacts, called *generic components,* together with the techniques that make them possible. These generic components bring their users the well-known benefits of libraries, but in the broader space of system architecture. The coding techniques and the implementations provided focus on tasks and issues that traditionally fall in the area of design, activities usually done *before* coding. Because of their high level, generic components make it possible to map intricate architectures to code in unusually expressive, terse, and easy-to-maintain ways.

Three elements are reunited here: design patterns, generic programming, and C++. These elements are combined to achieve a very high rate of reuse, both horizontally and vertically. On the horizontal dimension, a small amount of library code implements a combinatorial—and essentially open-ended—number of structures and behaviors. On the vertical dimension, the generality of these components makes them applicable to a vast range of programs.

This book owes much to design patterns, powerful solutions to ever-recurring problems in object-oriented development. Design patterns are distilled pieces of good design—recipes for sound, reusable solutions to problems that can be encountered in many contexts. Design patterns concentrate on providing a suggestive lexicon for designs to be conveyed. They describe the problem, a time-proven solution with its variants, and the consequences of choosing each variant of that solution. Design patterns go above and beyond anything a programming language, no matter how advanced, could possibly express. By following and combining certain design patterns, the components presented in this book tend to address a large category of concrete problems.

Generic programming is a paradigm that focuses on abstracting types to a narrow collection of functional requirements and on implementing algorithms in terms of these requirements. Because algorithms define a strict and narrow interface to the types they operate on, the same algorithm can be used against a wide collection of types. The implementations in this book use generic programming techniques to achieve a minimal commitment to specificity, extraordinary terseness, and efficiency that rivals carefully hand crafted code.

C++ is the only implementation tool used in this book. You will not find in this book code that implements nifty windowing systems, complex networking libraries, or clever logging mechanisms. Instead, you will find the fundamental components that make it easy to implement all of the above, and much more. C++ has the breadth necessary to make this possible. Its underlying C memory model ensures raw performance, its support for polymorphism enables object-oriented techniques, and its templates unleash an incredible code generation machine. Templates pervade all the code in the book because they allow close cooperation between the user and the library. The user of the library literally controls the way code is generated, in ways constrained by the library. The role of a generic component library is to allow user-specified types and behaviors to be combined with generic components in a sound design. Because of the static nature of the techniques used, errors in mixing and matching the appropriate pieces are usually caught during compile time.

This book's manifest intent is to create generic components—preimplemented pieces of design whose main characteristics are flexibility, versatility, and ease of use. Generic components do not form a framework. In fact, their approach is complementary—whereas a framework defines interdependent classes to foster a specific object model, generic components are lightweight design artifacts that are independent of each other, yet can be mixed and matched freely. They can be of great help in *implementing* frameworks.

Audience

The intended audience of this book falls into two main categories. The first category is that of experienced C++ programmers who want to master the most modern library writing techniques. The book presents new, powerful C++ idioms that have surprising capabilities, some of which weren't even thought possible. These idioms are of great help in writing high-level libraries. Intermediate C++ programmers who want to go a step further will certainly find the book useful, too, especially if they invest a bit of perseverance. Although pretty hard-core C++ code is sometimes presented, it is thoroughly explained.

The second category consists of busy programmers who need to get the job done without undergoing a steep learning investment. They can skim the most intricate details of implementation and concentrate on *using* the provided library. Each chapter has an introductory explanation and ends with a Quick Facts section. Programmers will find these features a useful reference in understanding and using the components. The components can be understood in isolation, are very powerful yet safe, and are a joy to use.

You need to have a solid working experience with C++ and, above all, the desire to learn more. A degree of familiarity with templates and the Standard Template Library (STL) is desirable.

Having an acquaintance with design patterns (Gamma et al. 1995) is recommended but not mandatory. The patterns and idioms applied in the book are described in detail. However, this book is not a pattern book—it does not attempt to treat patterns in full generality. Because patterns are presented from the pragmatic standpoint of a library writer, even readers interested mostly in patterns may find the perspective refreshing, if constrained.

Loki

The book describes an actual C++ library called Loki. Loki is the god of wit and mischief in Norse mythology, and the author's hope is that the library's originality and flexibility will remind readers of the playful Norse god. All the elements of the library live in the namespace `Loki`. The namespace is not mentioned in the coding examples because it would have unnecessarily increased indentation and the size of the examples. Loki is freely available; you can download it from http://www.awl.com/cseng/titles/0-201-70431-5.

Except for its threading part, Loki is written exclusively in standard C++. This, alas, means that many current compilers cannot cope with parts of it. I implemented and tested Loki using Metrowerks' CodeWarrior Pro 6.0 and Comeau C++ 4.2.38, both on Windows. It is likely that KAI C++ wouldn't have any problem with the code, either. As vendors release new, better compiler versions, you will be able to exploit everything Loki has to offer.

Loki's code and the code samples presented throughout the book use a popular coding standard originated by Herb Sutter. I'm sure you will pick it up easily. In a nutshell,

- Classes, functions, and enumerated types look `LikeThis`.
- Variables and enumerated values look `likeThis`.
- Member variables look `likeThis_`.
- Template parameters are declared with `class` if they can be only a user-defined type, and with `typename` if they can also be a primitive type.

Organization

The book consists of two major parts: techniques and components. Part I (Chapters 1 to 4) describes the C++ techniques that are used in generic programming and in particular in building generic components. A host of C++-specific features and techniques are presented: policy-based design, partial template specialization, typelists, local classes, and more. You may want to read this part sequentially and return to specific sections for reference.

Part II builds on the foundation established in Part I by implementing a number of generic components. These are not toy examples; they are industrial-strength components used in real-world applications. Recurring issues that C++ developers face in their day-to-day activity, such as smart pointers, object factories, and functor objects, are discussed in depth and implemented in a generic way. The text presents implementations that address basic needs and solve fundamental problems. Instead of explaining what a body of code does, the approach of the book is to discuss problems, take design decisions, and implement those decisions gradually.

Chapter 1 presents policies—a C++ idiom that helps in creating flexible designs.

Chapter 2 discusses general C++ techniques related to generic programming.

Chapter 3 implements typelists, which are powerful type manipulation structures.

Chapter 4 introduces an important ancillary tool: a small-object allocator.

Chapter 5 introduces the concept of generalized functors, useful in designs that use the Command design pattern.

Chapter 6 describes Singleton objects.

Chapter 7 discusses and implements smart pointers.

Chapter 8 describes generic object factories.

Chapter 9 treats the Abstract Factory design pattern and provides implementations of it.

Chapter 10 implements several variations of the Visitor design pattern in a generic manner.

Chapter 11 implements several multimethod engines, solutions that foster various trade-offs.

The design themes cover many important situations that C++ programmers have to cope with on a regular basis. I personally consider object factories (Chapter 8) a cornerstone of virtually any quality polymorphic design. Also, smart pointers (Chapter 7) are an important component of many C++ applications, small and large. Generalized functors (Chapter 5) have an incredibly broad range of applications. Once you have generalized functors, many complicated design problems become very simple. The other, more specialized, generic components, such as Visitor (Chapter 10) or multimethods (Chapter 11), have important niche applications and stretch the boundaries of language support.

Acknowledgments

I would like to thank my parents for diligently taking care of the longest, toughest part of them all.

It should be stressed that this book, and much of my professional development, wouldn't have existed without Scott Meyers. Since we met at the C++ World Conference in 1998, Scott has constantly helped me do more and do better. Scott was the first person who enthusiastically encouraged me to develop my early ideas. He introduced me to John Vlissides, catalyzing another fruitful cooperation; lobbied Herb Sutter to accept me as a columnist for *C++ Report*; and introduced me to Addison-Wesley, practically forcing me into starting this book, at a time when I still had trouble understanding New York sales clerks. Ultimately, Scott helped me all the way through the book with reviews and advice, sharing with me all the pains of writing, and none of the benefits.

Many thanks to John Vlissides, who, with his sharp insights, convinced me of the problems with my solutions and suggested much better ones. Chapter 9 exists because John insisted that "things could be done better."

Thanks to P. J. Plauger and Marc Briand for encouraging me to write for the *C/C++ Users Journal*, at a time when I still believed that magazine columnists were inhabitants of another planet.

I am indebted to my editor, Debbie Lafferty, for her continuous support and sagacious advice.

My colleagues at RealNetworks, especially Boris Jerkunica and Jim Knaack, helped me very much by fostering an atmosphere of freedom, emulation, and excellence. I am grateful to them all for that.

I also owe much to all participants in the comp.lang.c++.moderated and comp.std.c++ Usenet newsgroups. These people greatly and generously contributed to my understanding of C++.

I would like to address thanks to the reviewers of early drafts of the manuscript: Mihail Antonescu, Bob Archer (my most thorough reviewer of all), Allen Broadman, Ionut Burete, Mirel Chirita, Steve Clamage, James O. Coplien, Doug Hazen, Kevlin Henney, John Hickin, Howard Hinnant, Sorin Jianu, Zoltan Kormos, James Kuyper, Lisa Lippincott, Jonathan H. Lundquist, Petru Marginean, Patrick McKillen, Florin Mihaila, Sorin Oprea, John Potter, Adrian Rapiteanu, Monica Rapiteanu, Brian Stanton, Adrian Steflea, Herb Sutter, John Torjo, Florin Trofin, and Cristi Vlasceanu. They all have invested significant efforts in reading and providing input, without which this book wouldn't have been half of what it is.

Thanks to Greg Comeau for providing me with his top-notch C++ compiler for free.

Last but not least, I would like to thank all my family and friends for their continuous encouragement and support.,

Part I: Techniques

Chapter 1. Policy-Based Class Design

This chapter describes policies and policy classes, important class design techniques that enable the creation of flexible, highly reusable libraries—as Loki aims to be. In brief, policy-based class design fosters assembling a class with complex behavior out of many little classes (called *policies*), each of which takes care of only one behavioral or structural aspect. As the name suggests, a policy establishes an interface pertaining to a specific issue. You can implement policies in various ways as long as you respect the policy interface.

Because you can mix and match policies, you can achieve a combinatorial set of behaviors by using a small core of elementary components.

Policies are used in many chapters of this book. The generic `SingletonHolder` class template (Chapter 6) uses policies for managing lifetime and thread safety. `SmartPtr` (Chapter 7) is built almost entirely from policies. The double-dispatch engine in Chapter 11 uses policies for selecting various trade-offs. The generic Abstract Factory (Gamma et al. 1995) implementation in Chapter 9 uses a policy for choosing a creation method.

This chapter explains the problem that policies are intended to solve, provides details of policy-based class design, and gives advice on decomposing a class into policies.

1.1 The Multiplicity of Software Design

Software engineering, maybe more than any other engineering discipline, exhibits a rich multiplicity: You can do the same thing in many correct ways, and there are infinite nuances between right and wrong. Each path opens up a new world. Once you choose a solution, a host of possible variants appears, on and on at all levels—from the system architecture level down to the smallest coding detail. The design of a software system is a choice of solutions out of a combinatorial solution space.

Let's think of a simple, low-level design artifact: a smart pointer (Chapter 7). A smart pointer class can be single threaded or multithreaded, can use various ownership strategies, can make various trade-offs between safety and speed, and may or may not support automatic conversions to the underlying raw pointer type. All these features can be combined freely, and usually exactly one solution is best suited for a given area of your application.

The multiplicity of the design space constantly confuses apprentice designers. Given a software design problem, what's a good solution to it? Events? Objects? Observers? Callbacks? Virtuals? Templates? Up to a certain scale and level of detail, many different solutions seem to work equally well.

The most important difference between an expert software architect and a beginner is *the knowledge of what works and what doesn't*. For any given architectural problem, there are many competing ways of solving it. However, they scale differently and have distinct sets of advantages and disadvantages, which may or may not be suitable for the problem at hand. A solution that appears to be acceptable on the whiteboard might be unusable in practice.

Designing software systems is hard because it constantly asks you to *choose*. And in program design, just as in life, choice is hard.

Good, seasoned designers know what choices will lead to a good design. For a beginner, each design choice opens a door to the unknown. The experienced designer is like a good chess player: She can see

more moves ahead. This takes time to learn. Maybe this is the reason why programming genius may show at an early age, whereas software design genius tends to take more time to ripen.

In addition to being a puzzle for beginners, the combinatorial nature of design decisions is a major source of trouble for library writers. To implement a useful library of designs, the library designer must classify and accommodate many typical situations, yet still leave the library open-ended so that the application programmer can tailor it to the specific needs of a particular situation.

Indeed, how can one package flexible, sound design components in libraries? How can one let the user configure these components? How does one fight the "evil multiplicity" of design with a reasonably sized army of code? These are the questions that the remainder of this chapter, and ultimately this whole book, tries to answer.

1.2 The Failure of the Do-It-All Interface

Implementing everything under the umbrella of a do-it-all interface is not a good solution, for several reasons.

Some important negative consequences are intellectual overhead, sheer size, and inefficiency. Mammoth classes are unsuccessful because they incur a big learning overhead, tend to be unnecessarily large, and lead to code that's much slower than the equivalent handcrafted version.

But maybe the most important problem of an overly rich interface is *loss of static type safety*. One essential purpose of the architecture of a system is to enforce certain axioms "by design"—for example, you cannot create two Singleton objects (see Chapter 6) or create objects of disjoint families (see Chapter 9). Ideally, a design should enforce most constraints at compile time.

In a large, all-encompassing interface, it is very hard to enforce such constraints. Typically, once you have chosen a certain set of design constraints, only certain subsets of the large interface remain semantically valid. A gap grows between *syntactically valid* and *semantically valid* uses of the library. The programmer can write an increasing number of constructs that are syntactically valid, but semantically illegal.

For example, consider the thread-safety aspect of implementing a Singleton object. If the library fully encapsulates threading, then the user of a particular, nonportable threading system cannot use the Singleton library. If the library gives access to the unprotected primitive functions, there is the risk that the programmer will break the design by writing code that's syntactically—but not semantically—valid.

What if the library implements different design choices as different, smaller classes? Each class would represent a specific canned design solution. In the smart pointer case, for example, you would expect a battery of implementations: `SingleThreadedSmartPtr`, `MultiThreadedSmartPtr`, `RefCountedSmartPtr`, `RefLinkedSmartPtr`, and so on.

The problem that emerges with this second approach is the combinatorial explosion of the various design choices. The four classes just mentioned lead necessarily to combinations such as `SingleThreadedRefCountedSmartPtr`. Adding a third design option such as conversion support leads to exponentially more combinations, which will eventually overwhelm both the implementer and the user of the library. Clearly this is not the way to go. Never use brute force in fighting an exponential.

Not only does such a library incur an immense intellectual overhead, but it also is extremely rigid. The slightest unpredicted customization—such as trying to initialize default-constructed smart pointers with a particular value—renders all the carefully crafted library classes useless.

Designs enforce constraints; consequently, design-targeted libraries must help user-crafted designs to enforce *their own* constraints, instead of enforcing *predefined* constraints. Canned design choices would be as uncomfortable in design-targeted libraries as magic constants would be in regular code. Of course, batteries of "most popular" or "recommended" canned solutions are welcome, as long as the client programmer can change them if needed.

These issues have led to an unfortunate state of the art in the library space: Low-level general-purpose and specialized libraries abound, while libraries that directly assist the design of an application—the higher-level structures—are practically nonexistent. This situation is paradoxical because any nontrivial application has a design, so a design-targeted library would apply to most applications.

Frameworks try to fill the gap here, but they tend to lock an application into a specific design rather than help the user to *choose* and *customize* a design. If programmers need to implement an original design, they have to start from first principles—classes, functions, and so on.

1.3 Multiple Inheritance to the Rescue?

A `TemporarySecretary` class inherits both the `Secretary` and the `Temporary` classes.[1] `TemporarySecretary` has the features of both a secretary and a temporary employee, and possibly some more features of its own. This leads to the idea that multiple inheritance might help with handling the combinatorial explosion of design choices through a small number of cleverly chosen base classes. In such a setting, the user would build a multi-threaded, reference-counted smart pointer class by inheriting some `BaseSmartPtr` class and two classes: `MultiThreaded` and `RefCounted`. Any experienced class designer knows that such a naïve design does not work.

[1] This example is drawn from an old argument that Bjarne Stroustrup made in favor of multiple inheritance, in the first edition of *The C++ Programming Language*. At that time, multiple inheritance had not yet been introduced in C++.

Analyzing the reasons why multiple inheritance fails to allow the creation of flexible designs provides interesting ideas for reaching a sound solution. The problems with assembling separate features by using multiple inheritance are as follows:

1. *Mechanics.* There is no boilerplate code to assemble the inherited components in a controlled manner. The only tool that combines `BaseSmartPtr`, `MultiThreaded`, and `RefCounted` is a language mechanism called *multiple inheritance*. The language applies simple superposition in combining the base classes and establishes a set of simple rules for accessing their members. This is unacceptable except for the simplest cases. Most of the time, you need to carefully orchestrate the workings of the inherited classes to obtain the desired behavior.
2. *Type information.* The base classes do not have enough type information to carry out their tasks. For example, imagine you try to implement deep copy for your smart pointer class by deriving from a `DeepCopy` base class. What interface would `DeepCopy` have? It must create objects of a type it doesn't know yet.
3. *State manipulation.* Various behavioral aspects implemented with base classes must manipulate the same state. This means that they must use virtual inheritance to inherit a base class that holds the state. This complicates the design and makes it more rigid because the premise was that user classes inherit library classes, not vice versa.

Although combinatorial in nature, multiple inheritance by itself cannot address the multiplicity of design choices.

1.4 The Benefit of Templates

Templates are a good candidate for coping with combinatorial behaviors because they generate code at compile time based on the types provided by the user.

Class templates are customizable in ways not supported by regular classes. If you want to implement a special case, you can specialize any member functions of a class template for a specific instantiation of the class template. For example, if the template is `SmartPtr<T>`, you can specialize any member function for, say, `SmartPtr<Widget>`. This gives you good granularity in customizing behavior.

Furthermore, for class templates with multiple parameters, you can use partial template specialization (as you will see in Chapter 2). Partial template specialization gives you the ability to specialize a class template for only some of its arguments. For example, given the definition

```
template <class T, class U> class SmartPtr { ... };
```

you can specialize `SmartPtr<T, U>` for `Widget` and any other type using the following syntax:

```
template <class U> class SmartPtr<Widget, U> { ... };
```

The innate compile-time and combinatorial nature of templates makes them very attractive for creating design pieces. As soon as you try to implement such designs, you stumble upon several problems that are not self-evident:

1. *You cannot specialize structure.* Using templates alone, you cannot specialize the structure of a class (its data members). You can specialize only functions.
2. *Specialization of member functions does not scale.* You can specialize any member function of a class template with one template parameter, but you cannot specialize individual member functions for templates with multiple template parameters. For example:

```
template <class T> class Widget
{
   void Fun() { .. generic implementation ... }
};
// OK: specialization of a member function of Widget
template <> Widget<char>::Fun()
{
   ... specialized implementation ...
}
template <class T, class U> class Gadget
{
   void Fun() { .. generic implementation ... }
};
// Error! Cannot partially specialize a member class of Gadget
template <class U> void Gadget<char, U>::Fun()
{
   ... specialized implementation ...
}
```

3. *The library writer cannot provide multiple default values.* At best, a class template implementer can provide a single default implementation for each member function. You cannot provide *several* defaults for a template member function.

Now compare the list of drawbacks of multiple inheritance with the list of drawbacks of templates. Interestingly, multiple inheritance and templates foster complementary trade-offs. Multiple inheritance has scarce mechanics; templates have rich mechanics. Multiple inheritance loses type information, which

abounds in templates. Specialization of templates does not scale, but multiple inheritance scales quite nicely. You can provide only one default for a template member function, but you can write an unbounded number of base classes.

This analysis suggests that a combination of templates and multiple inheritance could engender a very flexible device, appropriate for creating libraries of design elements.

1.5 Policies and Policy Classes

Policies and policy classes help in implementing safe, efficient, and highly customizable design elements. A *policy* defines a class interface or a class template interface. The interface consists of one or all of the following: inner type definitions, member functions, and member variables.

Policies have much in common with traits (Alexandrescu 2000a) but differ in that they put less emphasis on type and more emphasis on behavior. Also, policies are reminiscent of the Strategy design pattern (Gamma et al. 1995), with the twist that policies are compile-time bound.

For example, let's define a policy for creating objects. The Creator policy prescribes a class template of type `T`. This class template must expose a member function called `Create` that takes no arguments and returns a pointer to `T`. Semantically, each call to `Create` should return a pointer to a new object of type `T`. The exact mode in which the object is created is left to the latitude of the policy implementation.

Let's define some policy classes that implement the Creator policy. One possible way is to use the `new` operator. Another way is to use `malloc` and a call to the placement `new` operator (Meyers 1998b). Yet another way would be to create new objects by cloning a prototype object. Here are examples of all three methods:

```
template <class T>
struct OpNewCreator
{
   static T* Create()
   {
      return new T;
   }
};

template <class T>
struct MallocCreator
{
   static T* Create()
   {
      void* buf = std::malloc(sizeof(T));
      if (!buf) return 0;
      return new(buf) T;
   }
};

template <class T>
struct PrototypeCreator
{
   PrototypeCreator(T* pObj = 0)
      :pPrototype_(pObj)
   {}
   T* Create()
   {
```

```
      return pPrototype_ ? pPrototype_->Clone() : 0;
   }
   T* GetPrototype() { return pPrototype_; }
   void SetPrototype(T* pObj) { pPrototype_ = pObj; }
private:
   T* pPrototype_;
};
```

For a given policy, there can be an unlimited number of implementations. The implementations of a policy are called *policy classes*.[2] Policy classes are not intended for stand-alone use; instead, they are inherited by, or contained within, other classes.

[2] This name is slightly inaccurate because, as you will see soon, policy implementations can be class *templates*.

An important aspect is that, unlike classic interfaces (collections of pure virtual functions), policies' interfaces are loosely defined. Policies are syntax oriented, not signature oriented. In other words, Creator specifies which syntactic constructs should be valid for a conforming class, rather than which exact functions that class must implement. For example, the Creator policy does not specify that `Create` must be static or virtual—the only requirement is that the class template define a `Create` member function. Also, Creator says that `Create` *should* return a pointer to a new object (as opposed to *must*). Consequently, it is acceptable that in special cases, `Create` might return zero or throw an exception.

You can implement several policy classes for a given policy. They all must respect the interface as defined by the policy. The user then chooses which policy class to use in larger structures, as you will see.

The three policy classes defined earlier have different implementations and even slightly different interfaces (for example, `PrototypeCreator` has two extra functions: `GetPrototype` and `SetPrototype`). However, they all define a function called `Create` with the required return type, so they conform to the Creator policy.

Let's see now how we can design a class that exploits the Creator policy. Such a class will either contain or inherit one of the three classes defined previously, as shown in the following:

```
// Library code
template <class CreationPolicy>
class WidgetManager : public CreationPolicy
{
   ...
};
```

The classes that use one or more policies are called *hosts* or *host classes*.[3] In the example above, `WidgetManager` is a host class with one policy. Hosts are responsible for assembling the structures and behaviors of their policies in a single complex unit.

[3] Although host classes are technically host class *templates*, let's stick to a unique definition. Both host classes and host class templates serve the same concept.

When instantiating the `WidgetManager` template, the client passes the desired policy:

```
// Application code
typedef WidgetManager< OpNewCreator<Widget> > MyWidgetMgr;
```

Let's analyze the resulting context. Whenever an object of type `MyWidgetMgr` needs to create a `Widget`, it invokes `Create()` for its `OpNewCreator<Widget>` policy subobject. How ever, it is the user of

`WidgetManager` who chooses the creation policy. Effectively, through its design, `WidgetManager` allows its users to configure a specific aspect of `WidgetManager`'s functionality.

This is the gist of policy-based class design.

1.5.1 Implementing Policy Classes with Template Template Parameters

Often, as is the case above, the policy's template argument is redundant. It is awkward that the user must pass `OpNewCreator`'s template argument explicitly. Typically, the host class already knows, or can easily deduce, the template argument of the policy class. In the example above, `WidgetManager` always manages objects of type `Widget`, so requiring the user to specify `Widget` again in the instantiation of `OpNewCreator` is redundant and potentially dangerous.

In this case, library code can use *template template parameters* for specifying policies, as shown in the following:

```
// Library code
template <template <class Created> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
   ...
};
```

In spite of appearances, the `Created` symbol does not contribute to the definition of `WidgetManager`. You cannot use `Created` inside `WidgetManager`—it is a formal argument for `CreationPolicy` (not `WidgetManager`) and simply can be omitted.

Application code now need only provide the name of the template in instantiating `WidgetManager`:

```
// Application code
typedef WidgetManager<OpNewCreator> MyWidgetMgr;
```

Using template template parameters with policy classes is not simply a matter of convenience; sometimes, it is essential that the host class have access to the template so that the host can instantiate it with a different type. For example, assume `WidgetManager` also needs to create objects of type `Gadget` using the same creation policy. Then the code would look like this:

```
// Library code
template <template <class> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
   ...
   void DoSomething()
   {
      Gadget* pW = CreationPolicy<Gadget>().Create();
      ...
   }
};
```

Does using policies give you an edge? At first sight, not a lot. For one thing, all implementations of the Creator policy are trivially simple. The author of `WidgetManager` could certainly have written the creation code inline and avoided the trouble of making `WidgetManager` a template.

But using policies gives great flexibility to `WidgetManager`. First, you can change policies *from the outside* as easily as changing a template argument when you instantiate `WidgetManager`. Second, you

can provide your own policies that are specific to your concrete application. You can use `new`, `malloc`, prototypes, or a peculiar memory allocation library that only your system uses. *It is as if* `WidgetManager` *were a little code generation engine, and you configure the ways in which it generates code.*

To ease the lives of application developers, `WidgetManager`'s author might define a battery of often-used policies and provide a default template argument for the policy that's most commonly used:

```
template <template <class> class CreationPolicy = OpNewCreator>
class WidgetManager ...
```

Note that policies are quite different from mere virtual functions. Virtual functions promise a similar effect: The implementer of a class defines higher-level functions in terms of primitive virtual functions and lets the user override the behavior of those primitives. As shown above, however, policies come with enriched type knowledge and static binding, which are essential ingredients for building designs. Aren't designs full of rules that dictate *before runtime* how types interact with each other and what you can and what you cannot do? Policies allow you to generate designs by combining simple choices in a typesafe manner. In addition, because the binding between a host class and its policies is done at compile time, the code is tight and efficient, comparable to its handcrafted equivalent.

Of course, policies' features also make them unsuitable for dynamic binding and binary interfaces, so in essence policies and classic interfaces do not compete.

1.5.2 Implementing Policy Classes with Template Member Functions

An alternative to using template template parameters is to use template member functions in conjunction with simple classes. That is, the policy implementation is a simple class (as opposed to a template class) but has one or more templated members.

For example, we can redefine the Creator policy to prescribe a regular (nontemplate) class that exposes a template function `Create<T>`. A conforming policy class looks like the following:

```
struct OpNewCreator
{
   template <class T>
   static T* Create()
   {
      return new T;
   }
};
```

This way of defining and implementing a policy has the advantage of being better supported by older compilers. On the other hand, policies defined this way are often harder to talk about, define, implement, and use.

1.6 Enriched Policies

The Creator policy prescribes only one member function, `Create`. However, `PrototypeCreator` defines two more functions: `GetPrototype` and `SetPrototype`. Let's analyze the resulting context.

Because `WidgetManager` inherits its policy class and because `GetPrototype` and `Set-Prototype` are public members of `PrototypeCreator`, the two functions propagate through `WidgetManager` and are directly accessible to clients.

However, `WidgetManager` asks only for the `Create` member function; that's all `WidgetManager` needs and uses for ensuring its own functionality. Users, however, can exploit the enriched interface.

A user who uses a prototype-based Creator policy class can write the following code:

```
typedef WidgetManager<PrototypeCreator>
   MyWidgetManager;
...
Widget* pPrototype = ...;
MyWidgetManager mgr;
mgr.SetPrototype(pPrototype);
... use mgr ...
```

If the user later decides to use a creation policy that does not support prototypes, the compiler pinpoints the spots where the prototype-specific interface was used. This is exactly what should be expected from a sound design.

The resulting context is very favorable. Clients who need enriched policies can benefit from that rich functionality, without affecting the basic functionality of the host class. Don't forget that *users*—and not the library—decide which policy class to use. Unlike regular multiple interfaces, policies give the user the ability to add functionality to a host class, in a typesafe manner.

1.7 Destructors of Policy Classes

There is an additional important detail about creating policy classes. Most often, the host class uses public inheritance to derive from its policies. For this reason, the user can automatically convert a host class to a policy and later `delete` that pointer. Unless the policy class defines a virtual destructor, applying `delete` to a pointer to the policy class has undefined behavior,[4] as shown below.

[4] In Chapter 4, Small-Object Allocation, you can find a discussion on exactly why this happens.

```
typedef WidgetManager<PrototypeCreator>
   MyWidgetManager;
...
MyWidgetManager wm;
PrototypeCreator<Widget>* pCreator = &wm; // dubious, but legal
delete pCreator;  // compiles fine, but has undefined behavior
```

Defining a virtual destructor for a policy, however, works against its static nature and hurts performance. Many policies don't have any data members, but rather are purely behavioral by nature. The first virtual function added incurs some size overhead for the objects of that class, so the virtual destructor should be avoided.

A solution is to have the host class use protected or private inheritance when deriving from the policy class. However, this would disable enriched policies as well (Section 1.6).

The lightweight, effective solution that policies should use is to define a nonvirtual protected destructor:

```
template <class T>
struct OpNewCreator
{
   static T* Create()
   {
    return new T;
```