Part I

Techniques

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 practically built out of 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 apparently 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 puzzling 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, try 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 efficiency. 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 is not able to 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 multithreaded, reference-counted smart pointer class by inheriting some `BaseSmartPtr` class

[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++.

and two classes: `MultiThreaded` and `RefCounted`. Any experienced class designer knows that such a naïve design does not work.

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 orchestrate the workings of the inherited classes carefully to obtain the desired behavior.
2. *Type information.* The base classes do not have enough type information to carry on their tasks. For example, imagine you try to implement deep copy for your smart pointer class by deriving from a `DeepCopy` base class. But 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 cannot address by itself the multiplicity of design choices.

1.4 Templates Bring Hope

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, with 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 artifacts. 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 only specialize 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 tradeoffs. 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 unbounded number of implementations. The implementations of a policy are called *policy classes*.[2] Policy classes are not intended for standalone use; instead, they are inherited by, or contained within, other classes.

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 what syntactic constructs should be valid for a conforming class, rather than what 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 what 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 complex structure and behavior.

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. However, it is the user of `WidgetManager` who chose 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.

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

[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.

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 can be simply missing.

Application code now has to provide only 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` needs also 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 one 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 instantiating `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 life 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 type-safe 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 `SetPrototype` 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. The 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 later on the user 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* decide what policy class to use, not the library. 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 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.

```
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 be-

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

havioral by nature. The first virtual function added incurs a 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:

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

Because the destructor is protected, only derived classes can destroy policy objects, so it's impossible for outsiders to apply `delete` to a pointer to a policy class. The destructor, however, is not virtual, so there is no size or speed overhead.

1.8 Optional Functionality Through Incomplete Instantiation

It gets even better. C++ contributes to the power of policies with an interesting feature. If a member function of a class template is never used, it is not even instantiated—the compiler does not look at it at all, except perhaps for syntax checking.[5]

This gives the host class a chance to specify and use optional features of a policy class. For example, let's define a `SwitchPrototype` member function for `WidgetManager`.

```
// Library code
template <template <class> class CreationPolicy>
class WidgetManager : public CreationPolicy<Widget>
{
   ...
   void SwitchPrototype(Widget* pNewPrototype)
   {
      CreationPolicy<Widget>& myPolicy = *this;
      delete myPolicy.GetPrototype();
      myPolicy.SetPrototype(pNewPrototype);
   }
};
```

The resulting context is very interesting:

- If the user instantiates `WidgetManager` with a Creator policy class that supports prototypes, he or she can use `SwitchPrototype`.

[5]According to the C++ standard, the degree of syntax checking for unused template functions is up to the implementation. The compiler does not do any semantic checking—for example, symbols are not looked up.

- If the user instantiates `WidgetManager` with a Creator policy class that does not support prototypes and tries to use `SwitchPrototype`, a compile-time error occurs.
- If the user instantiates `WidgetManager` with a Creator policy class that does not support prototypes and does not try to use `SwitchPrototype`, the program is valid.

This all means that `WidgetManager` can benefit from optional enriched interfaces but still work correctly with poorer interfaces—as long as you don't try to use certain member functions of `WidgetManager`.

The author of `WidgetManager` can define the Creator policy in the following manner:

> Creator prescribes a class template of one type `T` that exposes a member function `Create`. `Create` should return a pointer to a new object of type `T`. Optionally, the implementation can define two additional member functions, `T* GetPrototype()` and `SetPrototype(T*)`, having the semantics of getting/ setting a prototype object used for creation. In this case, `WidgetManager` exposes the `SwitchPrototype (T* pNewPrototype)` member function, which deletes the current prototype and sets it to the incoming argument.

In conjunction with policy classes, incomplete instantiation brings remarkable freedom to you as a library designer. You can implement lean host classes that are able to use additional features and degrade graciously for spartan, minimal policies.

1.9 Combining Policy Classes

The greatest usefulness of policies is apparent when you combine them. Typically, a highly configurable class uses several policies for various aspects of its workings. Then the library user selects the desired high-level behavior by combining several policy classes.

For example, consider designing a generic smart pointer class. (Chapter 7 builds a full implementation.) Say you identify two design choices that you should establish with policies: threading model and check before dereference. Then you implement a `SmartPtr` class template that uses two policies, as shown:

```
template
<
   class T,
   template <class> class CheckingPolicy,
   template <class> class ThreadingModel
>
class SmartPtr;
```

`SmartPtr` has three template parameters: the pointee type and two policies. Inside `SmartPtr`, you orchestrate the two policies into a sound implementation. `SmartPtr` becomes a coherent shell that integrates several policies, rather than a rigid, canned implementation. By designing `SmartPtr` this way, you confer on the user the ability to configure `SmartPtr` with a simple `typedef`:

```
typedef SmartPtr<Widget, NoChecking, SingleThreaded>
   WidgetPtr;
```

Inside the same application, you can define and use several smart pointer classes:

```
typedef SmartPtr<Widget, EnforceNotNull, SingleThreaded>
   SafeWidgetPtr;
```

The two policies can be defined as follows:

Checking: The `CheckingPolicy<T>` class template must expose a `Check` member function, callable with an lvalue of type `T*`. `SmartPtr` calls `Check`, passing it the pointee object before dereferencing it.

ThreadingModel: The `ThreadingModel<T>` class template must expose an inner type called `Lock`, whose constructor accepts a `T&`. For the lifetime of a `Lock` object, operations on the `T` object are serialized.

For example, here is the implementation of the `NoChecking` and `EnforceNotNull` policy classes:

```
template <class T> struct NoChecking
{
   static void Check(T*) {}
};
template <class T> struct EnforceNotNull
{
   class NullPointerException : public std::exception { ... };
   static void Check(T* ptr)
   {
      if (!ptr) throw NullPointerException();
   }
};
```

By plugging in various checking policy classes, you can implement various behaviors. You can even initialize the pointee object with a default value by accepting a reference to a pointer, as shown:

```
template <class T> struct EnsureNotNull
{
   static void Check(T*& ptr)
   {
      if (!ptr) ptr = GetDefaultValue();
   }
};
```

SmartPtr uses the Checking policy like so:

```
template
<
   class T,
   template <class> class CheckingPolicy,
   template <class> class ThreadingModel
>
class SmartPtr
   : public CheckingPolicy<T>
```

```
   , public ThreadingModel<SmartPtr>
{
   ...
   T* operator->()
   {
      typename ThreadingModel<SmartPtr>::Lock guard(*this);
      CheckingPolicy<T>::Check(pointee_);
      return pointee_;
   }
private:
   T* pointee_;
};
```

Notice the use of both the `CheckingPolicy` and `ThreadingModel` policy classes in the same function. Depending on the two template arguments, `SmartPtr::operator->` behaves differently on two orthogonal dimensions. Such is the power of combining policies.

If you manage to decompose a class in orthogonal policies, you can cover a large spectrum of behaviors with a small amount of code.

1.10 Customizing Structure with Policy Classes

One of the limitations of templates, mentioned in Section 1.4, is that you cannot use templates to customize the structure of a class—only its behavior. Policy-based designs, however, do support structural customization in a natural manner.

Suppose that you want to support nonpointer representations for `SmartPtr`. For example, on certain platforms some pointers might be represented by a handle—an integral value that you pass to a system function to obtain the actual pointer. To solve this you might "indirect" the pointer access through a policy, say, a Structure policy. The Structure policy abstracts the pointer storage. Consequently, Structure should expose types called `PointerType` (the type of the pointed-to object) and `ReferenceType` (the type to which the pointer refers) and functions such as `GetPointer` and `SetPointer`.

The fact that the pointer type is not hardcoded to `T*` has important advantages. For example, you can use `SmartPtr` with nonstandard pointer types (such as `near` and `far` pointers on segmented architectures), or you can easily implement clever solutions such as `before` and `after` functions (Stroustrup 2000a). The possibilities are extremely interesting.

The default storage of a smart pointer is a plain-vanilla pointer adorned with the Structure policy interface, as shown in the following code.

```
template <class T>
class DefaultSmartPtrStorage
{
public:
   typedef T* PointerType;
   typedef T& ReferenceType;
protected:
   PointerType GetPointer() { return ptr_; }
   void SetPointer(PointerType ptr) { ptr_ = ptr; }
```

```
private:
   PointerType pointee_;
};
```

The actual storage used is completely hidden behind Structure's interface. Now SmartPtr can use a Storage policy instead of aggregating a `T*`.

```
template
<
   class T,
   template <class> class CheckingPolicy,
   template <class> class ThreadingModel,
   template <class> class Storage = DefaultSmartPtrStorage
>
class SmartPtr;
```

Of course, `SmartPtr` must either derive from `Storage<T>` or aggregate a `Storage<T>` object in order to embed the needed structure.

1.11 Compatible and Noncompatible Policies

Suppose you create two instantiations of `SmartPtr`: `FastWidgetPtr`, a pointer without checking, and `SafeWidgetPtr`, a pointer with checking before dereference. An interesting question is: Should you be able to assign `FastWidgetPtr` objects to `SafeWidgetPtr` objects? Should you be able to assign them the other way around? If you want to allow such conversions, how can you implement that?

Starting from the reasoning that `SafeWidgetPtr` is more restrictive than `FastWidgetPtr`, it is natural to accept the conversion from `FastWidgetPtr` to `SafeWidgetPtr`. This is because C++ already supports implicit conversions that increase restrictions—namely, from non-`const` to `const` types.

On the other hand, freely converting `SafeWidgetPtr` objects to `FastWidgetPtr` objects is dangerous. This is because in an application, the majority of code would use `SafeWidgetPtr` and only a small, speed-critical core would use `FastWidgetPtr`. Allowing only explicit, controlled conversions to `FastWidgetPtr` would help keep `FastWidgetPtr`'s usage to a minimum.

The best, most scalable way to implement conversions between policies is to initialize and copy `SmartPtr` objects *policy by policy*, as shown below. (Let's simplify the code by getting back to only one policy—the Checking policy.)

```
template
<
   class T,
   template <class> class CheckingPolicy
>
class SmartPtr : public CheckingPolicy<T>
{
   ...
   template
```

```
      <
         class T1,
         template <class> class CP1,
      >
      SmartPtr(const SmartPtr<T1, CP1>& other)
         : pointee_(other.pointee_), CheckingPolicy<T>(other)
      { ... }
};
```

`SmartPtr` implements a templated copy constructor, which accepts any other instantiation of `SmartPtr`. The code in bold initializes the components of `SmartPtr` with the components of the other `SmartPtr<T1,CP1>` received as arguments.

Here's how it works. (Follow the constructor code.) Assume you have a class `ExtendedWidget`, derived from `Widget`. If you initialize a `SmartPtr<Widget, NoChecking>` with a `SmartPtr<ExtendedWidget, NoChecking>`, the compiler attempts to initialize a `Widget*` with an `ExtendedWiget*` (which works), and a `NoChecking` with a `SmartPtr<Widget, NoChecking>`. This might look suspicious, but don't forget that `SmartPtr` derives from its policy, so in essence the compiler will easily figure out that you initialize a `NoChecking` with a `NoChecking`. The whole initialization works.

Now for the interesting part. Say you initialize a `SmartPtr<Widget,EnforceNotNull>` with a `SmartPtr<ExtendedWidget, NoChecking>`. The `ExtendedWidget*` to `Widget*` conversion works just as before. Then the compiler tries to match `SmartPtr<ExtendedWidget, NoChecking>` to `EnforceNotNull`'s constructors.

If `EnforceNotNull` implements a constructor that accepts a `NoChecking` object, then the compiler matches that constructor. If `NoChecking` implements a conversion operator to `EnforceNotNull`, that conversion is invoked. In any other case, the code fails to compile.

As you can see, you have two-sided flexibility in implementing conversions between policies. You can implement a conversion constructor on the left-hand side, or you can implement a conversion operator on the right-hand side.

The assignment operator looks like an equally tricky problem, but fortunately, Sutter (2000) describes a very nifty technique that allows you to implement the assignment operator in terms of the copy constructor. (It's so nifty, you *have* to read about it. You can see the technique at work in Loki's `SmartPtr` implementation.)

Although conversions from `NoChecking` to `EnforceNotNull` and even vice versa are quite sensible, some conversions don't make any sense at all. Imagine converting a reference-counted pointer to a pointer that supports another ownership strategy, such as destructive copy (*à la* `std::auto_ptr`). Such a conversion is semantically wrong. The definition of reference counting is that all pointers to the same object are known and tracked by a unique counter. As soon as you try to confine a pointer to another ownership policy, you break the invariant that makes reference counting work.

In conclusion, conversions that change the ownership policy should not be allowed implicitly and should be treated with maximum care. At best, you can change the ownership policy of a reference-counted pointer by explicitly calling a function. That function succeeds if and only if the reference count of the source pointer is 1.

1.12 Decomposing a Class into Policies

The hardest part of creating policy-based class design is to decompose correctly the functionality of a class in policies. The rule of thumb is to identify and name the design decisions that take part in a class's behavior. Anything that can be done in more than one way should be identified and migrated from the class to a policy. Don't forget: Design constraints buried in a class's design are as bad as magic constants buried in code.

For example, consider a `WidgetManager` class. If `WidgetManager` creates new `Widget` objects internally, creation should be deferred to a policy. If `WidgetManager` stores a collection of `Widgets`, it makes sense to make that collection a storage policy, unless there is a strong preference for a specific storage mechanism.

At an extreme, a host class is totally depleted of any intrinsic policy. It delegates all design decisions and constraints to policies. Such a host class is a shell over a collection of policies and deals only with assembling the policies into a coherent behavior.

The disadvantage of an overly generic host class is the abundance of template parameters. In practice, more than four to six template parameters become awkward to work with. Still, they justify their presence if the host class offers complex, useful functionality.

Type definitions—`typedef` statements—are an essential tool in using classes that rely on policies. Using `typedef` is not only a matter of convenience. Using `typedef` ensures ordered use and easy maintenance. For example, consider the following type definition:

```
typedef SmartPtr
<
   Widget,
   RefCounted,
   NoChecked
>
WidgetPtr;
```

It would be very tedious to use the lengthy specialization of `SmartPtr` instead of `WidgetPtr` in code. But the tediousness of writing code is nothing compared with the major problems in *understanding* and *maintaining* that code. As design evolves, `WidgetPtr`'s definition might change—for example, to use a different checking policy than `NoChecked` in debug builds. It is essential that all the code use `WidgetPtr` instead of a hardcoded instantiation of `SmartPtr`. It's just like the difference between calling a function and writing the equivalent inline code: The inline code technically does the same thing but fails to build an abstraction behind it.

When decomposing a class in policies, it is very important to find an *orthogonal* decomposition. An orthogonal decomposition yields policies that are completely independent of each other. You can easily spot a nonorthogonal decomposition when various policies need to know about each other.

For example, think of an Array policy in a smart pointer. The Array policy is very simple—it dictates whether the smart pointer points to an array or not. The policy can be defined to have a member function `T& ElementAt(T* ptr, unsigned int index)`, plus a similar version for `const T`. The non-array policy simply does not define an `ElementAt`

member function, so trying to use it would yield a compile-time error. The `ElementAt` function is an optional enriched behavior as defined in Section 1.6.

The implementations of two policy classes that implement the Array policy follow.

```
template <class T>
struct IsArray
{
   T& ElementAt(T* ptr, unsigned int index)
   {
      return ptr[index];
   }
   const T& ElementAt(T* ptr, unsigned int index) const
   {
      return ptr[index];
   }
};
template <class T> struct IsNotArray {};
```

The problem is that whether the smart pointer points to an array or not has an unfortunate interaction with another policy: destruction. You must destroy pointers to objects with the `delete` operator, and destroy pointers to arrays of objects with the `delete[]` operator.

Two policies that do not interact with each other are called *orthogonal*. By this definition, the Array and the Destroy policies are not orthogonal.

If you still need to confine the quality of being an array and of destruction to separate policies, you need to establish a way for the two policies to communicate. You must have the Array policy expose a Boolean constant in addition to a function, and pass that Boolean to the Destroy policy. This complicates and somewhat constrains the design of both the Array and Destroy policies.

Nonorthogonal policies are an imperfection you should strive to avoid. They reduce compile-time type safety and complicate the design of both the host class and the policy classes.

If you must use nonorthogonal policies, you can keep dependencies to a minimum by passing a policy class as an argument to another policy class's template function. This way you can benefit from the flexibility specific to template-based interfaces. The downside remains that one policy must expose some of its implementation details to other policies. This decreases encapsulation.

1.13 Summary

Design is choice. Most often, the struggle is not that there is no way to solve a design problem, but that there are too many ways that apparently solve that problem. You must know which collection of solutions solves the problem in a satisfactory manner. The need to choose propagates from the largest architectural levels down to the smallest unit of code. Furthermore, choices can be combined, which confers on design an evil multiplicity.

To fight the multiplicity of design with a reasonably small amount of code, a design-oriented library needs to develop and use special techniques. These techniques are purposely conceived to support flexible code generation by combining a small number of

primitive devices. The library itself provides a number of such devices. Furthermore, the library exposes the *specifications* from which these devices are built, so the client can build her own. This essentially makes a policy-based design open-ended. These devices are called *policies*, and the implementations thereof are called *policy classes*.

The mechanics of policies consist of a combination of templates with multiple inheritance. A class that uses policies—a *host* class—is a template with many template parameters (often, *template template parameters*), each parameter being a policy. The host class "indirects" parts of its functionality through its policies and acts as a receptacle that combines several policies in a coherent aggregate.

Classes designed around policies support enriched behavior and graceful degradation of functionality. A policy can provide supplemental functionality that propagates through the host class due to public inheritance. Furthermore, the host class can implement enriched functionality that uses optional functionality of a policy. If the optional functionality is not present, the host class still compiles successfully, provided the enriched functionality is not used.

The power of policies comes from their ability to mix and match. A policy-based class can accommodate very many behaviors by combining the simpler behaviors that its policies implement. This effectively makes policies a good weapon for fighting against the evil multiplicity of design.

You can customize not only behavior but also structure with policy classes. This important feature takes policy-based design beyond the simple type genericity that's specific to container classes.

Policy-based classes support flexibility when it comes about conversions. If you use policy-by-policy copying, each policy can control what other policies it accepts, or converts to, by providing the appropriate conversion constructors, conversion operators, or both.

In breaking a class into policies, you should follow two important guidelines. One is to localize, name, and isolate design decisions in your class—things that are subject to a trade-off or could be sensibly implemented in other ways. The other guideline is to look for orthogonal policies, that is, policies that don't need to interact with each other and that can be changed independently.