The Design Is In The Code
Enhanced Reuse Techniques in C++

Abstract

Extreme Programming emphasizes the coding activity in all its aspects. It would be useful, then, if new coding techniques could render the code higher-level, more compact, more reusable, and easier to change.

This paper describes *policy classes* in C++ - a new approach that combines generic programming and object-oriented techniques. The end is to make it easier to express and convey design entities directly in code.

Using policy classes, library writers can make it possible to better achieve the "write once and only once" goal - provide high-level, powerful, extensible libraries. Leveraging design patterns [1] and language-specific idioms as recipes for successful solutions, generic libraries using policy classes truly democratize good designs. The example used throughout this paper is defining a truly generic, flexible, portable smart pointer - a popular C++ idiom and an incarnation of the Proxy design pattern [2].

Pre-requirements

Good knowledge of C++ and OOP design. Acquaintance with design patterns.

Keywords

Reuse, policy classes, traits, generic programming, object-oriented design, design patterns, C++, templates, template specialization.

Introduction

Extreme Programming puts a significant emphasis on coding. XP includes pair programming and continuous refactoring as essential components of the development process.

Old-fashioned processes render design modifications the exception; in XP, change is the rule. New code structures, then, ought to be developed to back up this tendency. They should be high-level, concise, expressive, easy to understand, and easy to change.

Traditionally, coding is seen as the process that takes a design to its ultimate detail. For this reason, sometimes the ideas underlying a piece of code, like a design pattern [1], get lost in the avalanche of details, context-related idiosyncrasies, and tweaks that the code has to provide to ensure proper functionality. There is an explanation for each line of code, but as a whole, the code blurs the design. Usually, developers help themselves with comments: "This class implements an Observer for objects of type `Widget`, which generate synchronous events of type `WidgetChanged`." Or: "Class `App` is a Singleton that supports multithreaded access." Short of analyzing the code of `App`, there is no simple way to figure out whether `App` is a multithreaded Singleton or not, and, for that matter, whether it is correctly and efficiently implemented. Seasoned designers know what a multithreaded Singleton is; the problem is that this information resides in a chunk of code that must be dug out, instead of a clear declarative statement.

Also, consider the innumerable ways in which the Singleton object can be initialized. Moreover, specialized techniques and recipes, like making a Singleton thread-safe [3] have limited portability. All these issues effectively limit one's ability to define a truly portable library that provides typical pattern implementations, thus missing an important reuse opportunity.

Switching between two well-known variants of a design pattern is a nontrivial process, because pattern variants don't map to code in a straight manner. For instance, changing the identifier type in a parameterized Factory Method [4] incurs cascading changes to the code and the data structures in the implementation space. This makes changes that are very simple and natural at design level to become unacceptably clumsy at coding level. Design and code evolve separately, and as the code dictates the actual behavior, usually the design is doomed to obsolescence. Hence the ironic adage: "The code is the design".

New generic programming techniques [5] render coding with design patterns and advanced C++ idioms simpler, and change more affordable. They make it possible to express some common design patterns and idioms in as little as a couple of clear declarative statements. If the default behavior is not satisfactory, you don't have to restart from scratch - you can punctually override the defaults to support an open-bounded range of behaviors.

These techniques map design much more directly to code, transforming the adage above into the more desirable: "The design is *in* the code".

The Multiplicity of Design

Much of the difficulty in implementing a software system is to choose between various competing solutions for each architectural issue, at all levels. The solutions are similar in the sense that they all ultimately solve (or promise to solve) the problem at hand. Yet, they sport different costs and tradeoffs and have distinct sets of advantages and disadvantages. In turn, each solution might have a large number of variants, and this multiplicity manifests itself at all levels of a design problem - from the highest to the lowest.

Design patterns come with a systematic way of discovering and documenting sound design solutions. Idioms do the practically the same in the narrower context of a specific programming language. However, programmers, although they might use these higher-level structures, must implement them in most cases starting from first principles.

This problem exists because of the combinatorial nature of design. A design is a deliberate choice of a set of tradeoffs, out of a combinatorial space. For instance, a Singleton object can be single-threaded or multithreaded; allocated statically, on the free store, or in some implementation-specific memory space; and constructed with various numbers and types of parameters. All these features can be combined freely. In the presence of such open-ended options, it is hard to provide a library Singleton that's not too rigid. A flexible implementation should leave the user full freedom for tweaking any of its aspects, in addition to providing a good set of defaults.

For implementing design structures and for working with design patterns, a library should help in the following ways.

- Cope with the combinatorial nature of design with a reasonably small code base;

- Allow the user to combine tradeoffs and design decisions in any ways that make sense;
- Validate the chosen set of tradeoffs at compile time;
- Make the resulting code reasonably efficient;
- Do not incur a penalty in space or runtime for options that are not used;
- Make the resulting implementation small, terse, and easy to explain to peers;
- Make it easy to change the design options after the fact.

No built-in feature or idiom of traditional procedural, functional, or object-oriented programming supports these requirements.

Procedural programming combines behaviors by using pointers to functions. Functional programming uses function objects. Object-oriented programs use inheritance and containment in various forms.

Each of these forms of coping with combinatorial behaviors has its own advantages and disadvantages. They all share the disadvantage of postponing to runtime things that should be performed at compile time. Most design decisions - like the threading model of a class - are immutable at runtime. Unnecessary dynamism wastes essential checking and optimization opportunities.

Generic programming techniques, implemented herein with C++ templates, can provide combinatorial behaviors with a linear amount of code. The mixing and matching is checked at compile time. In addition, possible behaviors are open-ended, thus reducing the need to start a design implementation from scratch whenever a special circumstance occurs.

Template Parameters as Design Constraints

Originally, parameterized types were introduced in C++ to allow creation of generic type-safe containers. Needs such as creating fixed generic arrays led to the addition of non-type (integral and address) template parameters. Over time, to accommodate more and more powerful generic programming idioms, the template engine of the compiler evolved into an intricate pattern-matching engine, combined with the integral arithmetic calculator that was already available.

Templates work at a meta-linguistic level; they form a little metalanguage on top of the rest of C++. Template code can be seen as guidelines to the compiler to generate actual code. The generated code is in non-templated C++.

This viewpoint leads to the idea that templates can be used to help various tasks that fall in the contingency of compile time, like design itself.

The elements controlling code generation are template parameters. Each template parameter is one degree of freedom on which generated code can vary. By fixing one of those parameters, you fix a dimension of variability, while the others can still control code generation on other dimensions.

A description of this fertile view of templates can be found in [6].

Link this concept with perusing a design pattern that offers many design choices. Combined, the choices lead to a plethora of variants, making traditional reusable design impractical, complicated, and hard to optimize.

However, if design choices are mapped to the template parameters of a template class, we can achieve combinatorial effect with a linear amount of well-chosen primi-

tives. The compiler generates and combines the appropriate primitives as requested at template instantiation time, and ignores the unused ones.

Below is presented with examples a C++ idiom that helps in building flexible libraries of typical design implementations.

Policy Classes

Policy classes are implementations of punctual design choices. They are not intended for standalone use; instead, they are inherited from, or contained within, other classes.

A policy class defines a C++-specific interface. The interface consists of inner type definitions, member functions, and possibly member data definitions. In this respect, policy classes resemble traits classes [7]. Unlike most traits classes, policy classes can be either templated or not templated. They also are typically behavior-richer than traits classes.

A policy class not only defines an interface; it also *implements* that interface. This sets an important distinction between policy classes and interfaces, without putting them to competition. Interfaces are a communication device; policy classes are an implementation device. In particular, a policy class can implement an interface.

For example, each of the three policy classes below implements a locking policy that corresponds to a specific threading model. Each locking policy class defines an inner type called `Lock`. The policy states that for the duration of a `Lock` object, operations on its host policy object are guaranteed to be atomic. This defines a simple, yet lucrative, threading model.

```
class SingleThreaded
{
public:
    class Lock
    {
    public:
        Lock(SingleThreaded&) {}
    };
};

class ClassLevelLockable
{
public:
    class Lock
    {
    public:
        Lock(ClassLevelLockable&) { mutex_.Acquire(); }
        ~Lock() { mutex_.Release(); }
    };
private:
    static Mutex mutex_;
};
```

```
class ObjectLevelLockable
{
public:
    class Lock
    {
    public:
        Lock(ObjectLevelLockable& obj) : m_(obj.mutex_)
        { m_.Acquire(); }
        ~Lock() { m_.Release(); }
    private:
        Mutex& m_;
    };
private:
    Mutex mutex_;
    friend class Lock;
};
```

The three policy classes defined above provide different threading models under the same interface. A class that wants to take advantage of locking inherits one of the policies. The actual policy class chosen depends on what kind of locking is needed, as shown below:

```
template <class Pointee>
class SmartPtr : public ClassLevelLockable
{
    ...
    SmartPtr& operator=(const SmartPtr& other)
    {
        Lock guard(other);
        ... perform operation ...
    }
private:
    Pointee* pointee_;
};
```

The question arises, what advantage does `ClassLevelLockable` give us? For one thing, the parameter passed to `Lock`'s constructor is unused, and `SmartPtr` could have used a static `Mutex` directly - a standard, easy to understand locking strategy.

However, if `SmartPtr` used a locking strategy directly, changing that strategy would have incurred changes to several `SmartPtr` member functions. The quality of locking (for instance, correctly pairing the `Acquire`/`Release` calls in the presence of early returns and exceptions) would have depended largely on `SmartPtr`'s implementer. Moreover, to figure out the actual locking strategy used, a reviewer must analyze the `SmartPtr` implementation.

The approach using a locking policy class has important advantages in flexibility and clarity:

- The locking strategy of `SmartPtr` can be figured by simply looking at the `SmartPtr` base class list.
- All locking strategies have the same interface, which means that you can later change the locking model only by changing `SmartPtr`'s base class and recompiling `SmartPtr`.
- The three locking policies are highly reusable classes that distill the threading aspect in defining a class, without interfering with other aspects. Therefore, locking policies can be carefully implemented, documented, and put in a library.

Compilers commonly optimize out unused arguments and empty base classes, leading to a `SmartPtr` implementation that's as efficient as a handcrafted one.

However, in the setting above, the user cannot use a single-threaded `SmartPtr` and a multithreaded one in the same application. If at least one `SmartPtr` is multithreaded, all `SmartPtr` instantiations will pay the locking price. To solve this problem, we must make the locking policy a *template parameter* of `SmartPtr`.

```
template <class Pointee,
    class LockingPolicy = SingleThreaded>
class SmartPtr : public LockingPolicy
{
    ...
};
```

The required interface of a locking policy is an inner class `Lock`. The semantics of `Lock` is that it makes operations on an object atomic for the lifetime of a `Lock` object. Any conforming implementation of the locking policy can be plugged in `SmartPtr`. The three classes presented provide default, often used, locking policies.

Policy Classes With Generic Behavior

The threading policy class defined above has semantics independent of the `SmartPtr` or pointee type. In general, however, policy classes have generic behavior.

For instance, imagine defining a *null checking policy* for our `SmartPtr`. Depending on the speed and the safety needed by the application, smart pointers might sport various checking levels. A fast `SmartPtr` might implement no checking at all, while in some applications a null check before each dereference is desirable.

A possible interface for a null checking policy class would consist of a unique function, `Check`. Because `Check` might need the type and the value of the pointee object, the null checking policy is a template class (as opposed to a simple class like the threading policy is). The policy below throws a standard error object if the pointer passed to `Check` is null. The text of the exception thrown contains the name of the pointee type, which makes it necessary to know the pointee type (`Pointee`) in `Check`.

```
template <class T>
class FullCheckingPolicy
{
public:
```

```
    static void Check(const T* p)
    {
        if (p) return;
        throw std::runtime_error(
            std::string("Null pointer of type ") +
            typeid(T).name() + " detected");
    }
};
```

Generic (templated) policy classes have considerably broader flexibility than simple policy classes. In practice, only the simplest policies are non-templated. Most policy classes either are templates or have template member functions.

If a class needs to enforce a policy class to be template, it can do this by requiring a template template parameter, as shown below.

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

This setting is particularly useful when `SmartPtr` needs to use the checking policy with two types instead of only with `Pointee`. Template template arguments also avoid the redundant repetition of the `Pointee` type, as in the slightly uncomfortable `SmartPtr<Widget, FullCheckingPolicy<Widget> >`.

Combining Multiple Policy Classes

In isolation, policy classes provide the known advantages of a modular design and the potential of increased reuse.

However, the true power of policy classes comes from their ability to combine freely. The client of a template class designed around policies can combine policies either by mixing and matching predefined policies, or by adding new ones. By combining several policy classes in a template class with multiple parameters, one achieves combinatorial behaviors with a linear amount of code. In addition to increasing the amount of reuse, this property of policy classes makes them suitable as building blocks in higher level libraries.

Let's combine a locking policy and a checking policy in the `SmartPtr` class template.

```
template
<
    class Pointee,
```

```
    class LockingPolicy = SingleThreaded,
    template <class U> class CheckingPolicy =
        FullCheckingPolicy
>
class SmartPtr
    : public LockingPolicy
    , public CheckingPolicy<Pointee>
{
    ...
    SmartPtr& operator=(const SmartPtr& other)
    {
        Lock guard1(*this);
        Lock guard2(other);
        ... perform copy operation ...
    }
    Pointee& operator*()
    {
        return *operator->();
    }
    Pointee* operator->()
    {
        CheckingPolicy<Pointee>::Check(pointee_);
        return pointee_;
    }
private:
    Pointee* pointee_;
};
```

The copying operation in `operator=` can be - and should be - defined by yet another policy class.

As suggested by the incomplete implementation above, in an implementation built around policy classes, the `SmartPtr` template class itself becomes syntactic glue that dovetails together several policy classes. Each of these policy classes implements a specific aspect of the smart pointer behavior.

Suppose we define three policies for locking (threading model) and four policies for checking. We already have twelve possible behaviors of `SmartPtr`. These behaviors are selected by the user of `SmartPtr` with a single type definition. For instance, the type definition below defines a pointer to `Widget` objects that support class level locking semantics and null checking with the `assert` macro (the `AssertChecked` policy class, not shown, is trivial to implement).

```
typedef SmartPtr
<
    Widget,
    ClassLevelLockable,
    AssertChecked
>
```

```
WidgetPtr;
```

Because class templates using policies - such as `SmartPtr` - are likely to have many template parameters, almost any reasonable use thereof should be through a `typedef`. Type definitions are not only a convenience but also an abstraction in itself. In the space of designing with policies, type definitions are equivalent to function definitions of traditional implementation. A practical consequence of concentrating policy selections in type definitions is that the resulting type definitions provide unique points of maintenance.

As multiple policies are defined and used with a class, the advantages of a policy-based approach become more and more evident and even spectacular. The `SmartPtr` class template described in [5] uses six policies - for ownership, error handling, implicit conversion, array handling, threading, and storage. Although each policy is easy to implement and needs little code, the policies combine to provide about 160 different behaviors, easily selectable by feeding appropriate template arguments to `SmartPtr`. It is very hard to deal with such a multitude of behaviors with traditional means.

A policy-based class thoroughly documents the syntactic and semantic requirements for each of its policies. This way, users can develop and use their own policies, which add to the pre-built ones. This makes a policy-based approach very flexible and suitable even in the most particular applications.

Because each policy in `SmartPtr` implements a well-defined decision or constraint in the smart pointer design space, `SmartPtr` users deal with high-level concepts such as error handling strategies or ownership strategies. In contrast, when developing a smart pointer starting from scratch, a programmer has to deal with all smart pointer design issues, plus a plethora of subtle syntactical issues. A handcrafted, more specialized, smart pointer is likely to be more rigid and less resilient to design changes than an instantiation of a policy-based smart pointer.

Policy-based implementations reach many of the goals stated in the introduction of this paper. Their use fosters a more natural mapping of design choices and constraints to implementation artifacts. Policies cope with the combinatorial nature of design with linear effort in an economic, organized manner. A policy-based class combines little selectable structural and behavioral entities into larger structures.

Conversions Between Policies

An application can use the same policy-based class template (`SmartPtr` in our example) instantiated with various design decisions. For instance, most smart pointers are checked upon each dereference, while some performance-critical code might use unchecked smart pointers.

From a compiler's perspective, two different instantiations of the same class template are completely different types. However, for the program, certain conversions between smart pointers are sensible. For instance, an unchecked smart pointer should be convertible to a smart pointer with dereference checking. On the other hand, converting a multithreaded smart pointer to a single-threaded one is an error that should be signaled at compile time.

Policy libraries can solve conversions in a simple and elegant way by initializing and assigning objects *on a per policy basis*.

For example, in addition to the copy constructor, `SmartPtr` gets added a conversion constructor that accepts a `SmartPtr` instantiation with different template arguments.

```
// Inside SmartPtr's class definition
template <class P, class L,
    template <class U> class C>
SmartPtr(const SmartPtr<P, L, C>& other)
: pointee_(other.pointee_)
, LockingPolicy(other)
, CheckingPolicy<Pointee>(other)
{
}
```

The code above initializes `SmartPtr` policy by policy, passing `other` to each policy constructor. One of three things might happen:

- The source policy is the same as the target policy. This is the case of a simple copy construction.
- The source policy is incompatible with the target policy. In this case, the initialization is a compile-time error.
- The source policy is convertible to the target policy. (For example, the source policy is derived from the target policy.) In this case, the initialization is legal.

The third case leaves the policy developer the option on which other policies to consider compatible, and which to reject. For example, if `FullCheckingPolicy` has a conversion constructor that accepts a `NonCheckingPolicy`, the `SmartPtr` user will be able to construct a checked pointer from an unchecked one.

Caveat

Designing with policy classes is expressive and productive, but inherently compile-time bound. Policy classes should be used for those aspects of a design that are fixed at runtime. Overusing policies might lead to excessive recompilation, code bloating, and rigid architectures.

Also, using policies is fun, but decomposing classes and choosing the right interface for each policy is hard. The onus of choosing the most orthogonal policy set falls on the component library developer.

A typical effect of choosing a non-orthogonal decomposition is the apparition of policies that depend on each other. A `SmartPtr`-related example: the storage policy - which deals with memory allocation and deallocation - is tied in unfortunate ways with the array policy, which deals with aspects like `operator[]`. Depending on the array policy, the storage policy must issue either a `delete` or a `delete[]` call.

Fortunately, library development is usually performed by a minority of experienced designers. From this perspective, policy-based libraries democratize good design practices.

Conclusion

Implementing code based upon advanced idioms and design patterns is a tough endeavor when starting from first principles. The higher-level design structures do not map to similar code structures naturally, and ultimately the implementation blurs the simplicity and the terseness of the design.

The *Policy Class* C++ idiom fosters defining high-level classes in terms of other classes, each implementing one specific aspect - design choice, constraint - of a design. The high-level class becomes a template class, accepting each of the design constraints as a template parameter.

The resulting setting allows library developers to define highly configurable high-level classes without sacrificing performance or flexibility. Well-designed policy-based classes support many different behaviors by combining at compile time a small set of core policies.

Bibliography

[1] Gamma, E., et al. *Design Patterns*, Addison–Wesley, Reading, MA, 1995.

[2] *Design Patterns*, pp. 207-217

[3] Schmidt, D., et al. Double-Checked Locking. In *Pattern Languages of Program Design 3*, Addison–Wesley, Reading, MA, 1998, pp. 363–375.

[4] *Design Patterns*, pp. 110

[5] Alexandrescu, A. *Design with C++* (tentative title), Addison–Wesley, Reading, MA, 2001 (in press).

[6] Coplien, James O. *Multi-Paradigm Design for C++*, Addison–Wesley, Reading, MA, 1999, pp. 39-43.

[7] Alexandrescu, A. Traits: The else-if-then of Types, *C++ Report*, April 2000.