WIKIPEDIA

Policy-based design

Policy-based design, also known as **policy-based class design** or **policy-based programming**, is a computer programming paradigm based on an idiom for C++ known as **policies**. It has been described as a compile-time variant of the strategy pattern, and has connections with C++ template metaprogramming. It was first popularized by Andrei Alexandrescu with his 2001 book *Modern C++ Design* and his column *Generic<Programming>* in the *C/C++ Users Journal*.

Although the technique could theoretically be applied to other languages, it is currently closely associated with C++ and D, as it requires a compiler with highly robust support for templates, which wasn't common before about 2003.

Contents

Overview

The central idiom in policy-based design is a class template (called the *host* class), taking several type parameters as input, which are instantiated with types selected by the user (called *policy classes*), each implementing a particular implicit interface (called a *policy*), and encapsulating some orthogonal (or mostly orthogonal) aspect of the behavior of the instantiated host class. By supplying a host class combined with a set of different, canned implementations for each policy, a library or module can support an exponential number of different behavior combinations, resolved at compile time, and selected by mixing and matching the different supplied policy classes in the instantiation of the host class template. Additionally, by writing a custom implementation of a given policy, a policy-based library can be used in situations requiring behaviors unforeseen by the library implementor. Even in cases where no more than one implementation of each policy will ever be used, decomposing a class into policies can aid the design process, by increasing modularity and highlighting exactly where orthogonal design decisions have been made.

While assembling software components out of interchangeable modules is a far from new concept, policy-based design represents an innovation in the way it applies that concept at the (relatively low) level of defining the behavior of an individual class. Policy classes have some similarity to callbacks, but differ in that, rather than consisting of a single function, a policy class will typically contain several related functions (methods), often combined with state variables or other facilities such as nested types. A policy-based host class can be thought of as a type of metafunction, taking a set of behaviors represented by types as input, and returning as output a type representing the result of combining those behaviors into a functioning whole. (Unlike MPL metafunctions, however, the output is usually represented by the instantiated host class itself, rather than a nested output type.)

A key feature of the *policy* idiom is that, usually (though it is not strictly necessary), the host class will derive from (make itself a child class of) each of its policy classes using (public) multiple inheritance. (Alternatives are for the host class to merely contain a member variable of each policy class type, or else to inherit the policy classes privately; however inheriting the policy classes publicly has the major advantage that a policy class can add new methods, inherited by the instantiated host class and accessible to its users, which the host class itself need not even know about.) A notable feature of this aspect of the policy idiom is that, relative to object-oriented programming, policies invert the relationship between base class and derived class - whereas in OOP interfaces are traditionally represented by (abstract) base classes and implementations of interfaces by derived classes, in policy-based design the derived (host) class represents the interfaces and the base (policy) classes implement them. It should also be noted that in the case of policies, the public inheritance does not represent an is-a relationship between the host and the policy classes. While this would traditionally be considered evidence of a design defect in OOP contexts, this doesn't apply in the context of the policy idiom.

A disadvantage of policies in their current incarnation is that the policy interface doesn't have a direct, explicit representation in code, but rather is defined implicitly, via duck typing, and must be documented separately and manually, in comments. The main idea is to use commonality-variability analysis to divide the type into the fixed implementation and interface, the policy-based class, and the different policies. The trick is to know what goes into the main class, and what policies should one create. The article mentioned above gives the following answer: wherever we would need to make a possible limiting design decision, we should postpone that decision, we should delegate it to an appropriately named policy.

Policy classes can contain implementation, type definitions and so forth. Basically, the designer of the main template class will define what the policy classes should provide, what customization points they need to implement.

It may be a delicate task to create a good set of policies, just the right number (e.g., the minimum necessary). The different customization points, which belong together, should go into one policy argument, such as storage policy, validation policy and so forth. Graphic designers are able to give a name to their policies, which represent concepts, and not those which represent operations or minor implementation details.

Policy-based design may incorporate other useful techniques. For example, the template method pattern can be reinterpreted for compile time, so that a main class has a skeleton algorithm, which — at customization points — calls the appropriate functions of some of the policies.

This will be achieved dynamically by concepts [1] in future versions of C++.

Simple example

Presented below is a simple (contrived) example of a C++ hello world program, where the text to be printed and the method of printing it are decomposed using policies. In this example, *HelloWorld* is a host class where it takes two policies, one for specifying how a message should be shown and the other for the actual message being printed. Note that the generic implementation is in *run()* and therefore the code is unable to be compiled unless both policies (print and message) are provided.

```
#include <iostream>
#include <string>

template <typename OutputPolicy, typename LanguagePolicy>
class HelloWorld : private OutputPolicy, private LanguagePolicy
{
    using OutputPolicy::print;
    using LanguagePolicy::message;

public:
    // Behaviour method
    void run() const
    {
        // Two policy methods
        print(message());
    }
};

class OutputPolicyWriteToCout
{
protected:
    template<typename MessageType>
    void print(MessageType const &message) const
    {
        std::cout << message << std::endl;
    }
};

class LanguagePolicyEnglish
{
protected:
    std::string message() const
    {
        return "Hello, World!";
    }
};

class LanguagePolicyGerman
{
protected:
    std::string message() const
    {
        return "Hallo Welt!";
    }
};

int main()
{
    /* Example 1 */
    typedef HelloWorld<OutputPolicyWriteToCout, LanguagePolicyEnglish> HelloWorldEnglish;
    HelloWorldEnglish hello_world;
    hello_world.run(); // prints "Hello, World!"

    /* Example 2
     * Does the same, but uses another language policy */
    typedef HelloWorld<OutputPolicyWriteToCout, LanguagePolicyGerman> HelloWorldGerman;

    HelloWorldGerman hello_world2;
    hello_world2.run(); // prints "Hallo Welt!"
}
```

Designers can easily write more OutputPolicies by adding new classes with the member function print() and take those as new OutputPolicies.

See also

- Mixin

References

1. http://www.stroustrup.com/good_concepts.pdf