

Polacastify

This document aims to give a description of the open source polacastify project that is currently under development as part of the polacanthus training material. Where C++ is known to have its limitations with respect to memory safety issues, that make it unpopular amongst many security aware developers, C++ is a language that offers 3 important constructs that are very useful in demonstrating decomposition in least authority style programming. Polacastify builds on 3 of these C++ constructs, templates, functors and cast operators, in order to build a convenient layer of syntactic sugar for least authority de/re-composition in a C++ environment.

Templates, functors and cast operators

The 3 C++ constructs that polacastify builds on are functors, templates and cast operators.

Templates are the C++ approach to generic programming, they allow types as parameters to for example class definitions, thus allowing large sets of classes to be defined with a single template, given that the number of number of type parameters is set.

Functors or function objects are classes that overload the () operator. The actual C++ construct being used for functors is thus 'operator overloading', of the () operator.

Basics

Polacastify consists of two distinct parts. The first part is the polacast template library, that defines all functor interfaces, and a basic set of simple functor based patterns that are usefull for decomposition and attenuation purposes. This library is the same for all projects using polacastify.

Next to the base template library, that works only on functors, the polacastify program is used on a per project basis. If a project has a number of header files that define (trees of) abstract classes (these are called interfaces by Java people), than the polacastify program will process these files and will define a set of template implementations for these interfaces that help the project to de/re compose where possible in a syntactically simple manner.

Implemetation

Polacastify makes use of two programs to do its job. The gccxml program is used to create an XML representation of a project header file. This XML representation is used as input for xsltproc. The xsltproc program is used to transform the XML into an (additional) header file, using a transformation sheet. A simple perl script is used to integrate the two existing programs and the transformation sheet into a usable tool.

Polacastify output

For each abstract class found in a projects header files, polacastify will create a set of 3 template implementations, so that these templates could be used with all defined abstract classes and their implementations.

decomposable_base templates

This template implementation is used by the other two below, but next to this can be used within the project. If an implementation class is defined as:

```
class FoolImpl : public FooPowerfull {  
    ...  
};
```

This can be replaced by:

```
class FoolImpl: public polacast::decomposable_base<FooPowerfull> {  
    ..  
};
```

Doing this would make FoolImpl decomposable.

Lets say that FooPowerfull itself has an abstract baseclasses FooBasic. We could give a reference to an instance of FooPowerfull to something expecting a FooBasic, but given that the receiver could dynamic_cast to a FooPowerfull, this would not actually have any use from an authority point of view.

The decomposable however defines an inner subclass of FooBasic and a cast operator to this inner class type. We could this use this as follows:

```
FoolImpl foo(...);  
..  
polacast::decomposable_base<FooPowerfull>::FooBasicFacet &foobasic=foo;  
Bob.somemethod(foobasic);
```

Next to baseclass based proxies, the decompose_base will also contain cast operators to functors, either as defined in the abstract class, or in any of its abstract baseclasses. Lets say that FooBasic defines a method 'int bar(char *,size_t)', and FooPowerfull defines a method 'void killme(void)'. This example would imply that we could get at functors for both these methods as follows:

```
polacast::decomposable_base<FooPowerfull>::FooBasicFacet::barFunctor  
    &mybar=foo;  
polacast::decomposable_base<FooPowerfull>::killmeFunctor &mykill=foo;
```

Please note that the thus defined mybar is an implementation of polacast::functor<int,char *,size_t> and mykill an implementation of polacast::functor<void,void>. It is this inheritance hierarchies that makes the combination with the functor template library (described later) so usefull.

decomposable_proxy templates

Next to adding `decomposable_base` to the inheritance hierarchy, it is also possible to use a proxy instead. Where the `decomposable_base` leaves the implementation of the methods to a derived class, the proxy implements them as forwarders to a target object with the same baseclass.

recomposition_proxy templates

It may in many cases be desirable to minimize refactoring of existing code by (re)composing an interface using functors. A simple way to allow for least authority usage without interface changes, is to decompose an interface into functors, create assertions and filters (as defined in the template library described next) for these functors, and create a recomposition from these filters and assertions.

```
polacast::recomposition<FooPowerfull>  
    recomp(fooFunctor,polacast::throwfunctor<void,void>());
```

Polacast template library

The polacast template library defines and works on the functors that are created by the polacastify template implementation described above. It does this up to a maximum number of arguments. Given that large amounts of arguments on a single method are considered bad practice, and each number of arguments requires separate template definitions, the maximum number is defined relatively low at 8 arguments for a method.

abstract functor baseclasses

The base functor template definition defines an abstract class that finds its implementation either by the polacastify templates, in the template library, or in the target project. An example template definition for 3 argument functors would be:

```
template <class R,T1,T2,T3>  
class functor {  
    public:  
        virtual R operator()(T1 arg1,T2 arg2,T3 arg3)=0;  
};
```

nullfunctor

The nullfunctor template defines a functor that does nothing, and returns a predefined value. This functor is meant to be used in recompositions to implement 'silently discard' for specific method invocations.

An example definition would be:

```
template <class R,T1,T2,T3>
class nullfunctor: public functor<R,T1,T2,T3> {
    R mRval;
public:
    nullfunctor(R rval):mRval(rval){}
    R operator()(T1,T2,T3) {
        return mRval;
    }
};
```

throwfunctor

The throwfunctor is similar to the nullfunctor, but other than 'silently discard' and returning a value, it throws an exception on invocation.

attenuation_proxy

The attenuation_proxy implements the basic pattern of attenuation for functors. Please note that with this proxy, a generic solution was chosen over a solution that would be preferable from a purely syntactic sugar approach.

The constructor of this proxy takes as arguments a reference to the target object, a reference to an invocation assertion functor, and for each argument of the target class, a filter/assertion functor reference. On invocation of the attenuation_proxy as functor, first the invocation assertion functor will be invoked. If no exception is thrown during this assertion invocation, then the target object will be invoked, but each argument will be replaced by the result of the appropriate filter/assertion functor invoked with the original argument.

```
template <class R,T1,T2,T3>
class attenuation_proxy: public functor<R,T1,T2,T3> {
    functor<R,T1,T2,T3> &mTarget;
    functor<void,void> &mBaseAssertion;
    functor<T1,T1> &mFilter1;
    functor<T2,T2> &mFilter2;
    functor<T3,T3> &mFilter3;
public:
    attenuation_proxy(functor<R,T1,T2,T3> &target,functor<void,void>
        &assertion,functor<T1,T1> &filt1,functor<T2,T2> &filt2,
        functor<T1,T1> &filt1): mTarget(target),
        mBaseAssertion(assertion),mFilter1(filt1),mFilter2(filt2),
        mFilter3(filt3){}
    R operator()(T1 arg1,T2 arg2,T3 arg3) {
        mBaseAssertion();
        return mTarget(mFilter1(arg1),mFilter2(arg2),mFilter3(arg3));
    }
};
```

Attenuation invocation assertions

The second argument of the `attenuation_proxy` is the invocation assertion. This is a functor<void,void> reference that will get invoked prior to any invocation of the target. The assertion functor can either return or throw an exception.

succeed_assertion

In situations where the attenuation has no argument independent precondition, the `succeed_assertion` is usefull. It simply returns directly.

invoke_quota_assertion

The `invoke_quota_assertion` can be used in order to make an attenuation (or a interdependent set of attenuations) that can be invoked until an invocation count reaches a pre-set maximum value.

```
class invoke_quota_assertion: public functor<void,void> {
    int    mMaxcount;
    int    mCount;
public:
    invoke_quota_assertion(int max):mMaxCount(max),mCount(0){}
    void operator()() {
        if (mCount >= mMaxCount)
            throw std::exception("invokation quota reached");
        mCount++;
        return;
    };
};
```

caretaker_assertion

The caretaker assertion is a boolean state class. On creation its behavior is to return, but after its 'revoke' method is invoked, any invocation after that will result in an exception being thrown.

```
class caretaker_assertion: public functor<void,void> {
    bool mRevoked;
public:
    caretaker_assertion():mRevoked(false){}
    void revoke(){mRevoked=true;}
    void operator()(){
        throw std::exception("Invocation of revoked functor");
        return;
    }
};
```

Attenuation argument filters and assertions

Where the invocation assertion argument can only return or throw an exception, the argument filters and assertions can choose to implement similar assertions with given arguments, and/or can choose to act as a filter on these arguments, changing the argument value.

null_filter

The null filter simply returns its argument.

```
template <class T>
class null_filter: public functor<T,T> {
    public:
        T operator()(T arg) {return arg;}
};
```

assertion_filter_inverter

The `assertion_filter_inverter` is a proxy for an other `assertion_filter*` functor. It implements a try/catch around invocation of the other functor, and throws an exception if the wrapper functor does not.

```
template <class T>
class assertion_filter_inverter: public functor<T,T> {
    functor<T,T> &mTarget;
    public:
        assertion_filter_inverter(functor<T,T> &target):mTarget(target) {}
        T operator(T arg) {
            try {
                mTarget(arg);
            } catch(std::exception &e) {
                return arg;
            }
            throw std::exception("No exception from target");
        }
};
```

range_filter_assertion

This functor throws an exception if the argument does not meet two '<' comparisons indicating a range.

```
template <class T>
class range_filter_assertion: public functor<T,T> {
    T mMin;
    T mMax;
    public:
        range_filter_assertion(T arg1,T arg2):mMin(arg1),mMax(arg2) {}
        T operator()(T arg) {
            if ((arg<mMin)|| (mMax<arg)) throw std::exception("out of range");
            return arg;
        }
};
```

range_filter

The `range_filter` is similar to the `range_filter_assertion`, but instead of throwing an exception, it will set the return value to either the max or the minimal value if it is out of range.

```
template <class T>
class range_filter: public functor<T,T> {
    T    mMin;
    T    mMax;
public:
    range_filter(T arg1,T arg2):mMin(arg1),mMax(arg2){}
    T operator()(T arg){
        if (arg < mMin) return mMin;
        if (mMax < arg) return mMax;
        return arg;
    }
};
```

equal_filter_assertion

The `equal_filter_assertion` asserts that the argument is equal (using `==`) to a constant defined on creation of the functor.

```
template <class T>
class equal_filter_assertion: public functor<T,T> {
    T    mConst;
public:
    range_filter_assertion(T arg1):mConst(arg1){}
    T operator()(T arg){
        if (arg == mConst) return arg;
        throw std::exception("invalid argument value");
    }
};
```

const_filter

The `const filter` simply ignores the argument and always return the same value defined at creation.

```
template <class T>
class const_filter: public functor<T,T> {
    T    mConst;
public:
    const_filter(T arg1,T arg2):mConst(arg1){}
    T operator()(T){
        return mConst;
    }
};
```

Status

This document is currently still in a state of flux and template definitions in this document have not yet been tested. This document is purely meant to describe the current (12-19-2007) state and direction of the project.

If you have any feedback you would like to give on this document that may be useful in the design or implementation, please contact me:

rmeijer <at> polacanthus <dot> net