Patterns for Coping with Copying in C++

Clone Alone

Kevlin Henney
kevlin@acm.org

Patterns and Pattern Languages

A pattern identifies a general approach to solving a problem, typically capturing a solution practice or collaborative structure. It identifies the general context of the problem, the nature of the problem, the interplay of conflicting forces, the configuration that resolves the problem, and the resulting context of applying such a solution. In short: where, what and how the problem arises and is solved. Rather than living in the abstract, a pattern needs to be illustrated by something concrete, such as a specific example with code and diagrams.

There are many forms for documenting patterns [Coplien1996], ranging from the highly structured heading-based template form used by the Gang of Four [Gamma+1995], to the more narrative form used by Alexander [Alexander+1977, Alexander1979]. In each form we should be able to find the essential elements I have just described. For this article I intend to use a relatively low-ceremony form structured around *problem*, *solution* and *discussion* sections.

Patterns can be grouped together and collected in a catalogue to provide a useful knowledge source, a simple software engineering handbook if you like. However, the full power of patterns is realised by connecting them together in a narrative structure called a pattern language. A pattern language is focused on the construction of a particular type or facet of a system. It should represent a reasonable set of practices and decisions that need to be taken to resolve the issues in this broader context.

Patterns do not belong exclusively to a single pattern language, but in including them within a language it is often best to document them in terms of the domain of that language, illustrating them with examples directly relevant to the language, rather than in purely general and abstract terms. The connections from one pattern to its possible successors and predecessors are found within the pattern [Alexander1979][1].

[1] There are some differing interpretations of the relationship between the context of a pattern and a pattern language. A literal reading of Alexander [Alexander1979] suggests that the knowledge of possible contexts for a pattern is contained within the pattern. There is also a case to be made — brought out in a recent discussion with Jim Coplien — for saying that the pattern is context free and it is a pattern language that provides all of the context. My own personal view on this is that a freestanding pattern comes with a context — enough to make it an intentional structure — but that this is incomplete; a pattern language provides the completion, weaving together short vignettes into a broader narrative.

*First published 1999 (*Overload 33*), revised 2003*

Copying

Understanding a programming language goes beyond by-rote knowledge of syntax and semantics; it is about use, idioms, expressing oneself in that language with intention rather than by accident or by dogma. It might be overstating it a little, but perhaps not too much, to say that there only are few concepts in C++ that if understood imply a total understanding of working in and with C++ itself, as opposed to knowledge of the workings of C++. By *understanding* I mean *grok* [Raymond1991], i.e. a deep understanding that takes in both the mechanism and the purpose. I consider `const`, and by implication `const_cast` and `mutable`, to be one such concept. Copying and conversions are another. It is copying — or rather one aspect of copying — on which I intend to focus in this article.

Copying provide us with our basic problem context. What does it mean to copy one object from another? It is most easily understood in terms of value objects versus referential objects. An object is characterised by identity, state and behaviour [Booch1994]. We can generally categorise elements of a software system in these terms [Henney1999a]: different profiles of identity, state and behaviour give rise to different styles of programming and tradeoffs.

Of the many profiles, we can characterise a value object as having insignificant identity, significant state and behaviour related directly to state. An example of this is a string where our focus is on its content and its manipulation, but not on its address in memory: comparison of the content of two strings is of interest, but comparison of their identities[2] is less useful. One value is substitutable for another with the same state. Thus in C++, value-based programming relies heavily on `const`, aliasing through references, the ability to copy by construction and assignment, and typically little or no involvement with subtyping hierarchies.

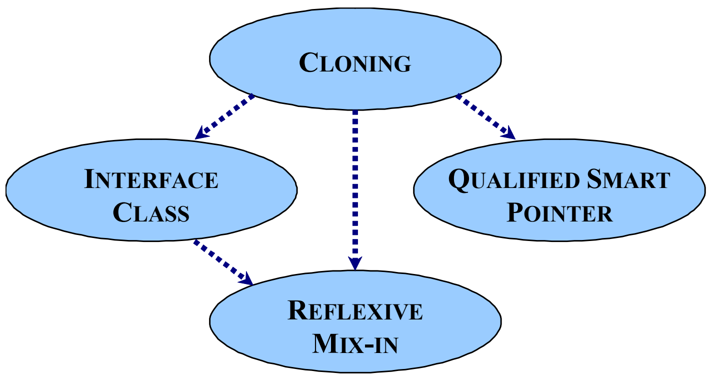
For reference-based objects — by which I mean anything that we would consider having significant identity as opposed to the C++ language feature — we find that dynamic allocation and usage by pointer is the norm, as well as involvement in subtyping hierarchies. We find that copying is less meaningful: What does it mean to copy something for which identity is significant? What is the relationship between the copy and the original in terms of identity? What would be a meaningful copy constructor or assignment operator for a person, for instance? In such cases, we normally unask the question and restrict copying behaviour, i.e. by making the copy constructor and assignment operator private.

These classifications serve us well for most cases, but in the territory between these crisp definitions lie some shades of grey. What mechanisms are open to us if we have referential objects in a class hierarchy for which copying is meaningful? Alternatively, what if we have a classification hierarchy for value-based objects that require copying?

This article presents what should be considered a small fragment of a first-cut pattern language focused on copying issues in C++. It balances patterns documented herein and those found in other sources. Note that the style of the code is at times affected by the need for brevity in an article, e.g. use of inlined functions, and does not necessarily stand as a recommendation. Pattern names are presented in small capitals. For instance, the patterns described here are CLONING, INTERFACE CLASS, REFLEXIVE MIX-IN and QUALIFIED SMART POINTER.

The relationship between these patterns can be shown as follows, where the dependency indicates a successor pattern:

[2] Strictly speaking, we might say that the state *is* the identity for a value. This is an appropriate aerial view from ten thousand metres. However, we are interpreting this in the context of C++, which has its own mechanisms and idioms for realising the concepts.



The pattern relationships not shown include specialisation [Henney1999b], such as QUALIFIED SMART POINTER from SMART POINTER, and minor inclusion, such as COPY BEFORE RELEASE by QUALIFIED SMART POINTER.

CLONING

How can objects in a class hierarchy be copied if their runtime class is not known?

Problem

A fruitful source of examples for exploring the composition and use of class hierarchies is a graphical editor that allows primitive and complex shapes to be drawn:

```
class graphic {...};
class ellipse : public graphic {...};
class rectangle : public graphic {...};
...
```

This hierarchy is based on classification: it is abstract at the root and concrete at the leaves. When we add a new ellipse to our picture, the code knows at the point of creation (somewhere) what the actual creation type of the object is. However, beyond that its usage is entirely through the base:

```
class picture
{
    ...
private:
    ...
    std::list<graphic *> graphics;
};
```

One feature we expect in a fully functioning graphics editor is the ability to copy and paste shapes. Given a graphic we need a new copy of it with the same state (although we may offset it by a couple of grid lines so that it does not mask the original) and behaviour. The following will not work:

```
class picture
{
    ...
    void copy(const graphic *original)
    {
        graphic *copy = new graphic(*original); // won't compile
        ...
    }
    ...
};
```

It is possible to do this in some languages, such as Ada 95, but not in C++: `graphic` is an abstract base class. The specific request we have here is to create a new instance of something that is incomplete — you will typically be greeted by a compiler message that is similarly incomplete in its helpfulness.

We can make an attempt at explicit type laundering as follows:

```
class picture
{
    ...
    void copy(const graphic *original)
    {
        graphic *copy = 0;
        if(typeid(*original) == typeid(rectangle))
            copy = new rectangle(static_cast<const rectangle &>(*original));
        else if(typeid(*original) == typeid(ellipse))
```

```
                copy = new ellipse(static_cast<const ellipse &>(*original));
            ...
        }
        ...
    };
```

Such a microarchitecture is brittle in the face of change — what if we add a new class to the hierarchy? — and clunky in its expression — large cascading control structures seem to suggest a joy of typing rather than of elegance, economy and extensibility. In essence, this solution is best described as a hack; it is a classic misuse of runtime type information (RTTI).

Solution

The location of the knowledge of the exact type of the source object for copying is within that object. Therefore we should turn the problem around and ask it for a copy of itself. Provide the entry point for this capability in the base class and realise its implementation in the concrete derived classes, each of which knows how to make a copy of itself:

```
class graphic
{
public:
    virtual graphic *clone() const = 0;
    ...
};

class rectangle : public graphic
{
public:
    virtual graphic *clone() const;
    ...
};

class ellipse : public graphic
{
public:
    virtual graphic *clone() const;
    ...
};

class picture
{
public:
    ...
    void copy(const graphic *original)
    {
        graphic *copy = original->clone();
        ...
    }
    ...
};
```

This tidy solution uses the natural form of RTTI found within object-oriented systems: objects encapsulate knowledge of not only of their state but also of their runtime type, which is decoupled from usage through inheritance and polymorphism.

Therefore, the system is more loosely coupled, is more open to change by extension and is expressed at a more abstract level. It is possible to refactor cloning capability more cohesively into a separate INTERFACE CLASS. Any new class in the hierarchy must implement the protocol for cloning by defining an appropriate `clone` member function, which places greater responsibility on the derived class implementer; implementing derived classes with a REFLEXIVE MIX-IN can simplify this, bringing the responsibility for some of the mechanism back into the framework. With an established protocol, copying and object lifetime management of cloned objects can be automated relatively transparently with a QUALIFIED SMART POINTER.

Discussion

CLONING resolves the problems deep type copying when only a shallow type is known, i.e. a copy of the specialised type rather than the generalised type is required, but objects are manipulated through the generalised type. This distinction between profound copying versus superficial copying with respect to type marks the distinction between CLONING and use of an ordinary copy constructor to create copies[3].

Another name for this technique is VIRTUAL COPY CONSTRUCTOR, as it is a special case of VIRTUAL CONSTRUCTOR [Coplien1992, Gamma+1995]. The specialisation is that it is a FACTORY METHOD [Gamma+1995] where the product and producer are instances of the same class. Note, however, that this is not the same as the PROTOTYPE pattern [Gamma+1995], or exemplar-based programming [Coplien1992]. The distinction can be found in PROTOTYPE's intent:

> *Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.*

In a C++ implementation PROTOTYPE will typically use CLONING, but a collaboration that uses CLONING is not necessarily an implementation of PROTOTYPE. In the graphics editor we could use PROTOTYPE to simplify the initial creation of new graphical objects [Gamma+1995], and this would build on our use of CLONING to give a PROTOTYPE collaboration.

[3] It is actually possible, using a couple of template techniques, to make this distinction largely transparent. Alan Griffiths and I have been discussing this recently based on an idea Alan has for an article [unpublished]. I will leave Alan to present his thoughts on this as it is definitely beyond the scope of this article.

INTERFACE CLASS

How can we represent the protocol for class usage without also expressing any implementation?

Problem

Abstract classes, such as `graphic`, are often considered partial implementations. As such they represent a focus for adding features to a hierarchy. In the case of copying this means that `graphic` must offer CLONING as part of its public interface, although it can offer no meaningful default implementation.

However, CLONING is a cohesive concept in its own right. If we are building a large framework we may find it recurring across root abstract classes. There is also the possibility that in such a framework, any multiple inheritance will lead to repeated inheritance of a `clone` member function. Assuming signature compatibility, i.e. both called `clone` and both qualified as `const`, a single `clone` using a covariant return type resolves the basic conflicts, ensuring that cloning of an object results in cloning the whole object and not its parts. However, if this is not followed the concept of cloning will be stated (and implemented) more than once in the interface of the derived class, reducing its cohesion.

Solution

Define a separate INTERFACE CLASS to express the common capability of derived classes, i.e. a class to represent the contract and many derived classes to fulfil it and express the implementation detail. This class has no data and the only ordinary member functions are declared `public` and pure `virtual`:

```
class cloneable
{
public:
    virtual ~cloneable();
    virtual cloneable *clone() const = 0;
};
```

If it is intended that objects can be deleted via a pointer to the INTERFACE CLASS the destructor should be declared `public` and `virtual`. It is a matter of personal taste as to whether or not the destructor is also declared pure `virtual` as it must be defined regardless. Personal preference is not to declare it as pure `virtual` as the occurrence of this usage — i.e. pure `virtual` declaration but with definition — is sufficiently rare that the majority of C++ developers are not familiar with it, and automatically take pure `virtual` to be equivalent to "has no definition".

On the other hand, if objects should not be deleted via the INTERFACE CLASS it should not offer this ability, i.e. the destructor should be made `protected` and non-`virtual` thus deferring full responsibility for destruction policy to derived classes[4]:

```
class cloneable
{
public:
    virtual cloneable *clone() const = 0;
```

[4] This practice is something I came up when dealing with an object ownership problem (for "object ownership problem" you can read "memory bug") a few years ago. Such a constraint clarifies code, and has also been included as a useful guideline elsewhere — as a reviewer, I recommended its inclusion in *Industrial Strength C++* [Henricson+1997].

```
protected:
    ~cloneable();
};
```

In either case, the body of the destructor should be empty. For cloning it seems to make sense to support public deletion, and so we will go with the first option.

Value-based polymorphic assignment makes little sense, causes many problems (both practical [Meyers1996] and philosophical) and should not be supported. Deferring policy responsibility to derived classes again resolves this. To prevent any default assignment generation (and the same applies for preventing default copy construction), the copy assignment operator should be declared `private` but left undefined:

```
class cloneable
{
    ...
private:
    cloneable &operator=(const cloneable &);
};
```

Otherwise assignment is best defined as `protected`.

In addition to resolving the deferred decisions, derived classes implementing the INTERFACE CLASS should consider using `virtual` inheritance:

```
class graphic : public virtual cloneable {...};
```

Interfaces express capabilities of objects, and where the object's class multiply inherits from other classes that ultimately inherit from the same interface, the object still only expresses these capabilities once. The default inheritance mechanism leads to the idea that any repeated inheritance represents multiple instances of the repeated class, with multiple instances of the data; this is not meaningful in the case of an INTERFACE CLASS. Nonetheless, tied to this is the idea that functions are still associated with separate subobjects and that the derived class is substitutable for one of two occurrences of the base class – an ambiguity that must be resolved by the programmer – rather than being directly substitutable for the interface, which is after all the point of defining one! Therefore, an INTERFACE CLASS that represents a mix-in capability should typically be a `virtual` base class to resolve this tension.

The result of expressing usage protocol as an INTERFACE CLASS is a clean separation between interface and implementation, i.e. contract and fulfilment, allowing derived classes to focus on the implementation detail. Codifying a common and cohesive public interface as an INTERFACE CLASS encourages a coherent framework design. Where the code structure of the implementation is similar for each derived class, varying only in its dependence on the type name of the derived class – as it will be in this case – consider using a REFLEXIVE MIX-IN to factor out the details completely.

Discussion

The explicit and physical separation of interface from implementation is key to modern component-based development (CBD) as typified by COM and CORBA. Interfaces as language elements distinct from implementation classes are an old idea that can be found in non-mainstream languages such as Emerald, Guide, Portlandish and Sather, and more recently (and popularly) in Java. Interfaces are considered to be separate modelling elements in UML [Booch+1999], although the issue is a little muddled by introducing both a type–class separation *and* a fashionable interpretation of interface. In C++, INTERFACE CLASS is the idiomatic practice that expresses such ideas [Henney1998b].

A distinction that I have found useful is that an INTERFACE CLASS represents the *usage type*, or some aspect of the usage, of an object, whereas its actual runtime type is only ever used for construction, i.e. a *creation type* [Barton+1994]. This decoupling allows us to view and discuss interfaces as the seams in our system [Booch+1999].

Note that by implication, the concrete classes derived from `graphic` specialise the return type of `clone` from `cloneable`. Covariant return types may be declared in C++ for `virtual` functions returning pointers or references. As inheritance implies specialisation, this makes perfect sense: when we clone a `graphic`, we do not simply get a `cloneable` back; we know we get some kind of `graphic`. Covariant return types allow us to express this constraint more precisely, and sidestep suspicious casts often found in code when you know something more about the type of an object being copied than just its base class. Contrast this with cloning in Java which normally requires a downcast of the clone to be of any use.

However, using covariant return types is often a game of challenge your compiler: it is typically the case that your compiler loses such a contest at present. Although handling covariant return types was the first extension considered during the standardisation of C++ and was adopted in 1992 [Stroustrup1994], it would be fair to say that compiler writers have not so much been slow to catch up as to not even get to the starting blocks with this feature.

REFLEXIVE MIX-IN

How can common behaviour and similar code structure be factored into a base class if there are details of the implementation that depend on the type name of the derived class?

Problem

Classes that implement a common capability expressed at the top of a hierarchy, such as with an INTERFACE CLASS, may end up with structurally very similar code. For CLONING, the implementation of `clone` is a matter of returning a newly allocated copy of the current instance:

```
class rectangle : public graphic
{
public:
    virtual rectangle *clone() const
    {
        return new rectangle(*this);
    }
    ...
};

class ellipse : public graphic
{
public:
    virtual ellipse *clone() const
    {
        return new ellipse(*this);
    }
    ...
};
```

This is potentially tedious, and is prone to cut and paste errors. At an aesthetic level it is also not pleasing: repetition implies that there is an abstraction missing. However, the dependency on the type of the derived class implies that a conventional mix-in class that provides common partial implementation cannot be used.

Solution

Define a template class that is intended for use as a base, inserted into the inheritance tree between the concrete implementing classes and the class hierarchy above them that expresses their common capability. It defines the common code in terms of the varying part [Coplien1999], i.e. the derived class, which is supplied as a template parameter, i.e. a self-parameterised base class:

```
template<typename derived, typename base>
class cloner : public base
{
public:
    virtual base *clone() const;
    ...
};
```

Passing in the `base` as a parameter allows the `cloner` instantiation to take on the role of deriving from it and allows it to know what type to return from `clone`: the class is not tied in to assuming that `cloneable` is an ancestor. Knowing the exact type of the derived class allows `cloner<derived, base>` to use `this` as an instance of `derived` to create a copy of the correct type:

```
template<typename derived, typename base>
base *cloner<derived, base>::clone() const
{
    return new derived(static_cast<const derived &>(*this));
}
```

To take advantage of this facility requires that a derived class derive from the template, using itself and its original base class as the template parameters:

```
class rectangle : public cloner<rectangle, graphic>
{
public:
    rectangle(const rectangle &);
    ...
};

class ellipse : public cloner<ellipse, graphic>
{
public:
    ellipse(const ellipse &);
    ...
};
```

With the exception that each derived class must provide a meaningful copy constructor, no further implementation work is needed: all of the implementation comes from the REFLEXIVE MIX-IN. The constraints on the relationship between the derived class and the template parameters are sufficiently well enforced by the type system, and limitations on the use of `static_cast` as opposed to traditional casts, that any dangerous misuse is caught at compile time.

Discussion

The `cloner` template is effectively abstract even though it apparently provides a complete implementation of `cloneable`. With the exception of some corner cases, it is in effect incomplete because it cannot be used on arbitrary freestanding types: it requires suitably related parameters to give it meaning. Further compile-time constraints can reduce the number of such corner cases [Henney1996].

REFLEXIVE MIX-IN is a particular kind of mix-in, both in that the code has an explicit but decoupled relationship with its derived classes, and in the sense that it is there to provide only implementation and not interface. We might differentiate this latter role by referring to it as a 'mix-imp'.

QUALIFIED SMART POINTER

How can we manage the access to an object used through a pointer, whilst also treating it much like a value object in terms of lifetime management and the effect of `const` qualification?

Problem

All of the creation and deletion of `graphic` objects in our editor example is handled explicitly by some controlling object, i.e. a `picture`. The same will be true of any other structure that contains and owns `graphic` objects. Consider the case where we allow users of the editor to create new compound shapes out of other graphic shapes and add them to a library. The basic structure of this is a COMPOSITE[5] and the means for supporting the creation and extension of a library of such shapes is with a PROTOTYPE [Gamma+1995]. Compound objects strictly own their contents. Assuming that the copy constructor is correctly defined, we can still take advantage of the code automation of the `cloner` REFLEXIVE MIX-IN. However, we must explicitly manage the details of creation and destruction to ensure correct ownership:

```
class compound : public cloner<compound, graphic>
{
public:
    compound(const compound &other)
    {
        typedef std::list<graphics *>::const_iterator iterator;
        for(iterator current = other.graphics.begin();
            current != other.graphics.end();
            ++current)
        {
            graphics.push_back((*current)->clone());
        }
    }
    virtual ~compound()
    {
        for(; !graphics.empty(); graphics.pop_back())
        {
            delete graphics.back();
        }
    }
    ...
private:
    ...
    std::list<graphic *> graphics;
};
```

Here the ownership concepts, i.e. creation and deletion, must be managed by explicit control flow and, naturally, careful programming. This code also does not fully reflect our understanding of the constraints: although `other.graphics` is `const` in the body of the copy constructor, this constraint does not carry through to its indirectly held contents, i.e. we could still perform non-`const` operations to change the state of any of `other`'s held `graphic` objects if we wished.

[5] COMPOSITE is perhaps better named RECURSIVE WHOLE-PART which is more explicit in terms of describing structure and in disambiguating between the many uses of the word composite and composition in OO. It also emphasises the specific relationship with the WHOLE–PART pattern [Buschmann+1996].

Solution

We introduce a QUALIFIED SMART POINTER that in use appears, almost transparently, like a conventional pointer. It retains the level of indirection required to allow its target object to take advantage of inheritance and polymorphism, but asserts ownership and access semantics that make it value-like, including support for assignment.

The constructors, destructor and copy assignment operator have been defined to `clone` and `delete` their contents as appropriate, ensuring that the lifetimes of the objects and pointer bindings coincide. The member access operators, `operator*` and `operator->`, have been overloaded with respect to `const`-ness so that their results follow the `const` qualification:

```
template<typename cloneable_type>
class cloning_ptr
{
public:
    cloning_ptr(const cloneable_type *other = 0)
      : body(other ? other->clone() : 0)
    {
    }
    cloning_ptr(const cloning_ptr &other)
      : body(other.body ? other.body->clone() : 0)
    {
    }
    ~cloning_ptr()
    {
        delete body;
    }
    cloning_ptr &operator=(const cloning_ptr &rhs)
    {
        cloneable_type *old_body = body;
        body = rhs.body ? rhs.body->clone() : 0;
        delete old_body;
        return *this;
    }
    cloneable_type &operator*()
    {
        return *body;
    }
    const cloneable_type &operator*() const
    {
        return *body;
    }
    cloneable_type *operator->()
    {
        return body;
    }
    const cloneable_type *operator->() const
    {
        return body;
    }
    ...
private:
    cloneable_type *body;
};
```

In use this drastically simplifies our `compound` class:

```
class compound : public cloner<compound, graphic>
{
public:
    compound(const compound &other)
      : graphics(other.graphics)
```

```
        {
        }
        virtual ~compound()
        {
        }
        ...
    private:
        ...
        std::list< cloning_ptr<graphic> > graphics;
    };
```

Discussion

The QUALIFIED SMART POINTER is implemented as a template class[6], allowing any class that supports CLONING through a `clone` member function to be used. This generalisation clearly accommodates, but does not restrict us to, examples built on the `cloneable` INTERFACE CLASS. In standard/STL style we can set out the requirements on `cloneable_type` as follows, assuming that `ptr` is of type `cloneable_type *`:

Expression	*Return type*	*Semantics and notes*
`ptr->clone()`	convertible to `cloneable_type`	*pre:* `ptr != 0` *post*: copy of `ptr` returned
`delete ptr`	`void`	`*ptr` no longer usable

Because there is no exchange of ownership when `cloning_ptr` is initialised from a `cloneable_type *`, there are no subtleties with silent conversions from a `cloneable_type *` to a `cloning_ptr` and so this constructor can be a converting constructor, i.e. it is not declared `explicit`. It is a matter of taste as to whether or not you also add user-defined conversion operators to support an implicit conversion from a `cloning_ptr` to a `cloneable_type *`. More conventionally one would provide `get` member functions, following the style of `std::auto_ptr`. Either way, such queries would be overloaded with respect to `const`-ness with their result type following the same qualification.

The case for `get` is simple to make: there is a good chance that not all the world works with your framework, but you should still be able to get your hands on the owned contents. The case for `release` is a little less compelling as the aim of `cloning_ptr` is to provide an owning pointer and not simply an incidental wrapper for an object, i.e. it won't take ownership of anything passed into it: it will clone it. Consistency suggests that if you want your own pointer to the content you use `get`, and if you want your own version of it you take an explicit clone yourself. You might wish to introduce `release`, and perhaps an `acquire`, if you intend to implement a general grouping construct into which you can add and remove specific `graphic` objects whilst retaining their identity (classic COMPOSITE). However, beware of mixing ownership models. More generally you can also provide a `reset` or `clear` member function.

Further generalisation is also possible using member templates to allow `cloning_ptr` types that are templated on different parts of a hierarchy to be mixed. This applies to the copy constructor and the assignment operator.

Note that the assignment operator implements COPY BEFORE RELEASE [Henney1997a, Henney1997b, Henney1998a] to preserve the integrity of the object in the event of `clone` exceptions or self-assignment.

[6] The basic motivation, as a design preservation tool to reflect `const` constraints in a whole-part structure, is something that Jon Jagger and I have discussed in some detail. It is also possible to further generalise this template in terms of an adaptor so that it can be combined with other smart pointer concepts. This, however, is beyond the scope of the current article.

QUALIFIED SMART POINTER represents a convergence of many patterns, including SMART POINTER, and therefore PROXY [Gamma+1995] of which it is a specialised form, RESULT TYPE FOLLOWS QUALIFICATION and COPY BEFORE RELEASE. It also represents a reasonable conclusion to this exploration of polymorphic copying.

References

[Alexander+1977] Christopher Alexander, Sara Ishikawa and Murray Silverstein with Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

[Alexander1979] Christopher Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.

[Barton+1994] John J Barton and Lee R Nackman, *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, Addison-Wesley, 1994.

[Booch1994] Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings, 1994.

[Booch+1999] Grady Booch, James Rumbaugh and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[Buschmann+1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.

[Coplien1992] James O Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992.

[Coplien1996] James O Coplien, *Software Patterns*, SIGS, 1996.

[Coplien1999] James O Coplien, *Multi-Paradigm Design for C++*, Addison-Wesley, 1999.

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Henney1996] Kevlin Henney, "Constraining template parameter types", *Overload* 12, February 1996.

[Henney1997a] Kevlin Henney, "Self Assignment? No Problem!", *Overload* 20, June/July 1997.

[Henney1997b] Kevlin Henney, "Safe Assignment? No Problem!", *Overload* 21, August 1997.

[Henney1998a] Kevlin Henney, "Creating Stable Assignments", *C++ Report* 10(6), June 1998.

[Henney1998b] Kevlin Henney, "Idioms: Breaking the Language Barrier", presented at the ACCU's *C and C++ European Developers Forum*, September 1998.

[Henney1999a] Kevlin Henney, "The Road to Software Architecture", presented at the *Software Architecture '99* conference, February 1999.

[Henney1999b] Kevlin Henney, "Patterns Inside Out: Understanding Relationships Between Patterns", presented at the *Application Development '99* conference, July 1999.

[Henricson+1997] Mats Henricson and Eric Nyquist, *Industrial Strength C++: Rules and Recommendations*, Prentice Hall, 1997, `http://hem.fyristorg.com/erny/industrial/`.

[Meyers1996] Scott Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.

[Raymond1991] Eric Raymond, *The New Hacker's Dictionary*, MIT Press, 1991.

[Stroustrup1994] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.