C++ Patterns
Executing Around Sequences

Kevlin Henney
May 2000

kevlin@acm.org
kevlin@curbralan.com

Abstract

Sequential control flow forms the backbone of any procedural program. Just as common function, data, and structure in a C++ program can be factored out using functions, classes, and templates, common control flow schemas can also be named and abstracted. One such recurring schema, or programming cliché, is the embracing of a sequence by a pair of actions, such as resource acquisition and release operations executed around the actual resource usage code.

The patterns in this paper document idiomatic practices for abstracting such control flow in C++. They are connected into a language and further explored through a narrative example.

Pre and Post Sequence Actions

Paired actions – where a function is called before some statement sequence and a corresponding function afterwards – are commonly associated with resource acquisition and release. Pre- and post-sequence actions are a common feature of block-scoped resource management – e.g. allocate memory, use it, deallocate it – and this programming cliché can be found time and time again across many programs.

Such repetition suggests a missing abstraction, and is both error-prone (from the programming perspective) and prone in the presence of error (not exception safe). What is needed is a way to capture this control flow form encapsulating it so that it is simple and exception-safe.

Exceptions, Complexity, and Abstraction

Idioms are context-dependent; as with multi-threading, the presence of exceptions invalidates many assumptions and practices, blowing working code out of the water and into the deep abyss of debugging. Exceptions make a handful of long-standing and well-respected C++ idioms inappropriate for modern C++ [ISO1998]. For instance, the basic rules for safe assignment have changed from the ORTHODOX CANONICAL CLASS FORM [Coplien1992] to post-standard C++ [Henney1998, Sutter2000], such that the original form is no longer considered safe. Adopting old styles with new rules tends to lead to disgraceful rather than graceful failure, and exceptions are an inevitable feature of modern C++ programs.

Perhaps the greatest upset to any well-laid plan of action is something out of the ordinary: The least favorite chore for most programmers is handling exceptions! The number of ways something can go wrong seems to outnumber the ways it can go right, leading to exception code that dwarfs the core program logic. Standard C++ provides a general exception-handling model that addresses the control and information flow issues. However, as it is non-local in effect it also affects the way the conventional control flow is – or should be – expressed. In many modern programs exceptions are not something that can be ignored safely, although there are cases when they can be avoided. For instance, errors that are expected rather than exceptional should rely on more traditional schemes of communication, such as returning error codes, e.g. failure to open a file.

The presence of exceptions in a program can have a significant impact. It is possible, if care is not taken, for an object to be left in an indeterminate state, thereby leaving the whole program in an unstable state. Safety – and hence stability – can be classed at one of three levels [Sutter2000]:

- The *basic guarantee* states that in the presence of exceptions, the throwing object will not leak resources. The object will be in a stable and usable, albeit not necessarily predictable, state.
- The *strong guarantee* states that the program's state will remain unchanged in the presence of exceptions, that is, commit-or-rollback semantics.
- The *nothrow guarantee* promises that exceptions are never thrown.

In code, exception safety can be achieved in one of two ways:

- Code may be scaffolded explicitly with `try`, `catch`, and `throw` to ensure explicit and graceful recovery.
- Exception neutral code is code that works in the presence of exceptions without requiring explicit exception handling code. Such code can also be described as exception-agnostic.

A sign of weakness in a class design is that it forces its user to remember lots of particular and subtle conditions of use: "function `a` must be called before `b`, unless condition `c` is true, in which case `d` must follow `e`, etc.". Such interfaces suggest that the design is incomplete, the user of the class is doing work that a good design would not require: Their code is verbose and repetitive, and writing it is error prone.

The Patterns

Table 1 lists the patterns described here in thumbnail form, providing a summary of the problem and solution for each. Other key patterns used in this paper are described toward the end in thumbnail form in Table 2.

Name	*Problem*	*Solution*
EXECUTE AROUND OBJECT	How do you abstract the execution of a pair of actions that surround a sequence of statements, or a single action that follows a sequence?	Provide a helper class for declaring a local object before the sequence. Its constructor is provided with any arguments necessary to perform the *before* action, and any that must be remembered for the destructor to perform the *after* action.
EXECUTE AROUND PROXY	How can you apply EXECUTE AROUND OBJECT conveniently for individual calls on an object?	Provide a PROXY that forwards function calls to the target object, with each call wrapped by an EXECUTE AROUND OBJECT.
EXECUTE AROUND POINTER	How do you define an EXECUTE AROUND PROXY when the actions performed on a target object are the same for all functions?	Define a SMART POINTER whose `operator->` returns a temporary SMART POINTER. The temporary's `operator->` performs the *before* action and its destructor performs the *after* action.
EXECUTE AROUND FUNCTION	How do you safely group and execute a sequence of statements that must be enclosed by a pair of actions, or followed by a single action?	Provide a function or object that holds the sequence of statements and pass it to another function for execution. The called function also executes the *before* and *after* actions as necessary.

Table 1.Thumbnails for patterns documented in this paper, listed in order.

The patterns may be organized into a pattern language, as shown in Figure 1. The natural entry points into the language are the EXECUTE AROUND OBJECT and EXECUTE AROUND FUNCTION patterns, and the arrows show subsequent flow through the language.

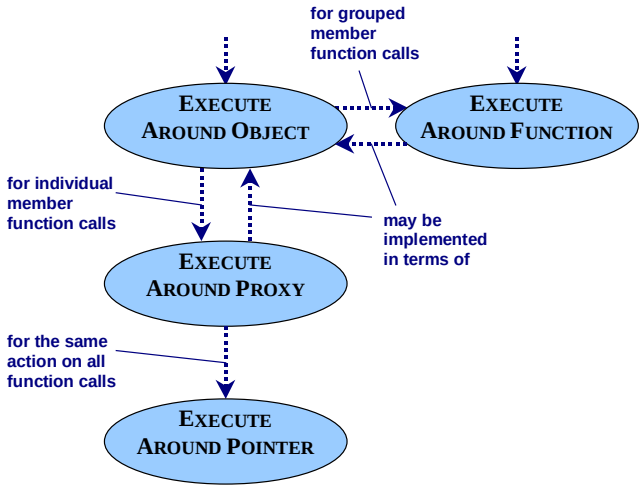

Figure 1. Patterns and their successors for abstracting execution around sequences in C++.

The patterns are presented using a brief, low-ceremony pattern form: pattern name; a problem–solution thumbnail; one or more examples; a more detailed description of the problem, identifying the forces; a more detailed description of the solution, identifying the consequences. The collection of patterns is followed by a pattern story that combines many of the patterns in an evolving worked example.

EXECUTE AROUND OBJECT

Define a helper object that executes actions before and after a sequence of statements in its constructor(s) and destructor.

Example: Critical Sections

Synchronization of data access is vital in the presence of multiple threads. Synchronization primitives, such as mutexes and semaphores, provide locking and unlocking operations to brace a critical section of code. At most one thread is permitted to execute in a critical section; its code is effectively atomic with respect to other threads. Operating system primitives are better wrapped up in classes than used in their raw API form:

```
class mutex
{
public:
    void lock();
    void unlock();
    ....
};
```

A mutex object is used typically to guard another shared resource object of some kind:

```
resource shared;
mutex    guard;
```

The following code is typical of its use:

```
{
    guard.lock();
    .... // use shared
    guard.unlock();
}
```

However, this is unsafe. What if an exception is generated before `unlock` is called? The `guard` object will remain locked, blocking further access to the `shared` object. The following code addresses these exception-safety concerns directly:

```
{
    guard.lock();
    try
    {
        .... // use shared
    }
    catch(...)
    {
        guard.unlock();
        throw;
    }
    guard.unlock();
}
```

Yes, this code is exception-safe; no, it is neither elegant nor brief. A solution that is both economical and safe places the responsibility for acquisition and release into a separate object, using the invocation of the destructor at scope termination – exceptional or otherwise – as the call stack unwinds:

```
template<typename lockee>
class locker
{
public:
    locker(lockee &to_lock)
      : target(to_lock)
    {
        target.lock();
    }
```

```
    ~locker()
    {
        target.unlock();
    }
private:
    lockee &target;
};
```

In practice this solution is more robust and briefer than the previous versions, the unsafe version included:

```
{
    locker<mutex> critical(guard);
    .... // use shared
} // destructor for critical called
```

The critical region is now tied directly to a scope. The decision to make `locker` a template class allows it to be used with any resource acquiring and releasing interface that supports `lock` and `unlock` member functions.

Example: Scoped Dynamic Memory Management

Objects that are created and destroyed within a local scope are normally variables whose lifetime is tied to the scope and automatically managed. Large objects, objects for which the relevant constructor can only be determined at the point of creation, and polymorphic objects whose type is similarly determined late, must be created and destroyed explicitly by the programmer with `new` and `delete` expressions:

```
{
    resource *ptr = new resource;
    ....
    ptr->update();
    ....
    delete ptr;
}
```

Exceptions thrown before the `delete` expression will cause a memory leak. In the following (rather cumbersome) code, `try`, `catch` and a re-`throw` are employed to cure the amnesia:

```
{
    resource *ptr = new resource;
    try
    {
        ....
        ptr->update();
        ....
    }
    catch(...)
    {
        delete ptr;
        throw;
    }
    delete ptr;
}
```

The standard C++ library provides a template class, `std::auto_ptr`, to abstract the management and safety of such scope-bound allocations. In addition to its destructor's release semantics, `std::auto_ptr` offers a familiar working syntax by also being a SMART POINTER:

```
{
    auto_ptr<resource> ptr(new resource);
    ....
    ptr->update();
    ....
}
```

This idiom applies generally to predictable acquisition-and-release control flow, not simply block-scoped `new` and `delete`. In the following class a device is created in the constructor and destroyed in the destructor:

```
class log
{
public:
    log()
      : out(new null_device)
    {
    }
    log(const std::string &name)
      : out(new file(name))
    {
    }
    ~log()
    {
        delete out;
    }
    ....
private:
    device *out;
};
```

The scope of the owning object is still said to own the allocated object, although not in the conventional sense of block scope. `std::auto_ptr` can automate such housekeeping tasks:

```
class log
{
public:
    log()
      : out(new null_device)
    {
    }
    log(const std::string &name)
      : out(new file(name))
    {
    }
    ~log()
    {
    } // destruction of out is automatic
    ....
private:
    auto_ptr<device> out;
};
```

`std::auto_ptr` simplifies the coding for two cases: a heap object whose lifetime should be bound safely to an enclosing scope, and an exception-safe transfer of object ownership through arguments or return values. However, its design has evolved through many unsatisfactory compromises to the point that its change-of-ownership semantics and its full interface are less than intuitive. It is standard, however, and the `const auto_ptr` idiom [Sutter2000] allows it to be used simply and in a manner closer to its original intent. There are many alternative implementations of this idiom for local memory management that are more direct and less surprising, including the Boost library's [Boost] `scoped_ptr` and `scoped_array`, and the practices described by Cargill for object ownership [Cargill1996].

Problem

How do you abstract the execution of a pair of actions that surround a sequence of statements, or a single action that follows a sequence? Such code is as common as it is error-prone: It is all too easy to forget the action following the sequence – unlocking a lock, releasing a resource, deleting an object. The duplication of such code suggests that there be some way of wrapping up the control flow.

The throwing of exceptions, which in their eagerness to short-circuit the rest of the sequence also bypass the post-sequence action, further complicates the matter. An exception will propagate along the call chain, clearly notifying someone along it of the bad news, but in doing so will create bad news of its own: A case of the messenger shooting the sender and, by dint of relationship, the receiver.

Unlike Java or Win32 extensions to C and C++, standard C++ does not support a `finally` block which is executed regardless of the outcome of its associated `try` block. Direct solutions, fashioned from `try`, `catch`, and `throw`, trade clarity for safety. It is not immediately obvious by inspection whether or not they are in fact correct. Presented with such complexity it is perhaps understandable that many

programmers adopt the ostrich posture when confronted with exception-safety. They would as soon plunge their head in the sand as consider the intricacies of such code, hoping the issue will either go away or that somehow it does not apply to their code, and if it does it is not their problem.

However, exception-safety cannot be ignored in any system that uses exceptions, which is, almost without exception (sic), any standard C++ program. Direct solutions convert small and simple pieces of code into longer and more complex ones. For a trained eye, code is just about manageable at that scale. Much production code is already complex and scarred by the ravages of continued development; tracing normal control flow becomes a feat in itself, without the additional worry of exceptional code.

Any of the conventional forms of discontinuous control flow – `break`, `continue`, `return` (and `goto`) – can give rise to problems similar to those with `throw`, leaving a sequence prematurely and unresolved. It is a common coding guideline to caution against jumps and multiple exit-points from loops and functions: It is a common reluctance on the part of programmers to follow such recommendations, exacerbating the problems of premature ejection.

Solution

Provide a helper class that should be used to define an object before the sequence of statements. Its constructor is provided with the necessary arguments to perform the pre-sequence action and for the destructor to later perform the post-sequence action.

In C++, a constructor is called on creation of an object for the sole purpose of initializing it; that is, the constructor describes the "boot sequence" for an object. Conversely, a destructor is automatically called at the end of an object's life to finalize or clean it up, that is, to shut it down in an orderly fashion. The calling of a destructor is deterministic: For stack variables lifetime is tied to the enclosing scope.

It is this determinism that allows constructor and destructor calls to straddle a sequence. A helper object can take advantage of this, executing the pre-sequence action in the constructor and the post-sequence action in the destructor. Where the helper object must execute against another object, the object and any additional state must be passed into its constructor.

The relationship between helper object lifetime and the destructor is crucial: This idiom addresses not only control-flow abstraction but also exception safety. Destructors are called for stack objects as the stack is unwound on leaving a scope. Stack unwinding occurs as a result of either normal flow out of the scope or exceptional flow from a `throw`. Therefore clean-up actions occur independently of outcome: after sequential flow to the end of a block; following a jump statement, such as `return`; the result of a thrown exception. Thus, destructors in C++ play the role of `finally` blocks in other languages.

If the post-sequence action taken depends on whether or not an exception has been thrown, the `std::uncaught_exception` function can be used to determine the reason the destructor is being called. However, caveat programmer: The result of `std::uncaught_exception` is worthless if the stack was already being unwound when the helper object was created.

The EXECUTE AROUND OBJECT pattern is found at the heart of the misnamed RESOURCE ACQUISITION IS INITIALIZATION idiom [Stroustrup1997] – misnamed because what makes this pattern tick is what happens in the destructor and not in the constructor. Resource acquisition often occurs independently of the EXECUTE AROUND OBJECT, as in the case of memory acquisition. *RESOURCE RELEASE IS FINALIZATION* would perhaps be a better name for resource-based applications of EXECUTE AROUND OBJECT. Resource acquisition and release is only one application of this control flow abstraction. Other uses include the restoration of state. Here the helper object plays the role of a 'Jeeves class' [Glassborow2000] by cleaning up and restoring particular things to the way they were before, e.g. a handler or a cursor replaced only for the duration of a particular task.

The elegance and brevity of the EXECUTE AROUND OBJECT pattern counterbalances the ugliness and wordiness of alternative, less-safe solutions. It requires the definition and comprehension of an additional class, but once written and understood it may be applied repeatedly. EXECUTE AROUND OBJECT may be used to address sequence of actions outside a block, so long as it is tied to a meaningful lifetime, e.g. within another object where the pre- and post-sequence actions lie at each end of the enclosing object's own life.

Where the EXECUTE AROUND OBJECT encloses a single member function call on a target object that is the subject of the EXECUTE AROUND OBJECT, EXECUTE AROUND PROXY and EXECUTE AROUND POINTER offer alternative, briefer approaches. EXECUTE AROUND FUNCTION is an alternative to EXECUTE AROUND OBJECT that focuses on encapsulating named sequences of use.

EXECUTE AROUND PROXY

Provide a PROXY object to execute actions before and after each function call on an object.

Example: Loading and Saving State

Consider a class for representing lookup table objects in some kind of efficient persistent store. The state of the object can be loaded from and saved back to the store (or an intermediate caching layer, as appropriate), and access to the store can be locked for reading – permitting multiple simultaneous readers – or for writing – excluding all but the current writer from access. The `const`-ness of each member function reflects the view of the persistent store:

```
class table
{
public:
    void lock() const;   // acquire lock for reading
    void lock();         // acquire lock for writing
    void unlock() const; // release read lock
    void unlock();       // release write lock
    void load() const;
    void save();
    value_type query(key_type) const;
    void modify(key_type, value_type);
    ....
};
```

An EXECUTE AROUND OBJECT can be introduced to address exception-safety concerns. In this case, the `locker` template class introduced in a previous example can be reused directly. The following fragments show read and write locks being applied:

```
value_type reading(const table &data, key_type key)
{
    locker<const table> critical(data);
    data.load();
    return data.query(key);
}
```

```
void writing(table &data, key_type key, value_type value)
{
    locker<table> critical(data);
    data.load();
    data.modify(key, value);
    date.save();
}
```

In the writing example, changes are saved only if all is successful, that is, there are no exceptions.

Although the basic locking has been taken care of, the manipulation of the target has been obscured by details of persistence management. For simple calls to individual functions – `query` or `modify` – this detail dominates the code. Employing a PROXY to encapsulate the executing-around allows further simplification of client code, hiding both the exception-safe locking and the persistence management. Users need only call the function they require, and the appropriate locking, loading, etc. will happen:

```
class table_view
{
public:
    table_view(table *to_view)
      : target(to_view)
    {
    }
    value_type query(key_type key) const
    {
        locker<const table> critical(*target);
        target->load();
        return target->query(key);
```

```
    }
    void modify(key_type key, value_type value)
    {
        locker<table> critical(*target);
        target->load();
        target->modify(key, value);
        target->save();
    }
    ....
private:
    table *target;
};
```

The `table_view` provides a subset of the features defined in `table`, making its interface source compatible with `table`. Factoring out the commonality between the two classes into a common interface class would be the next refinement, allowing genuine runtime substitutability.

Example: Setting and Restoring State

Actions that block a user interface for a few seconds are normally accompanied by a change in cursor, from standard pointer to hourglass. In the absence of exceptions, the following code demonstrates a typical style for an API and its use:

```
{
    cursor *old_cursor = view->set_cursor(cursor::wait);
    .... // perform long task
    view->set_cursor(old_cursor);
}
```

This exchange can be made exception-safe and more convenient by with an EXECUTE AROUND OBJECT:

```
{
    cursor_exchange scoped(view, cursor::wait);
    .... // perform long task
}
```

Rather than acquiring and releasing a resource, the `cursor_exchange` class installs and restores it:

```
class cursor_exchange
{
public:
    cursor_exchange(component *view, cursor *new_cursor)
      : exchanged_view(view),
        old_cursor(view->set_cursor(new_cursor))
    {
    }
    ~cursor_exchange()
    {
        exchanged_view->set_cursor(old_cursor);
    }
private:
    component *const exchanged_view;
    cursor    *const old_cursor;
};
```

A data object, such as a document, that is viewed by a separate windowing component will possess only a few operations in need of a wait cursor. It is tempting to set and reset the cursor in the document object itself. However, the document object is often decoupled from the windowing API, a common layering that supports parallel development, unit testing, and reuse. The alternative is to provide the wrapping at the point of call:

```
class document
{
public:
    virtual void cut(selection *) = 0;
    virtual void repaginate() = 0;
    ....
};
```

```
void repaginate(document *target, component *view)
{
    cursor_exchange scoped(view, cursor::wait);
    target->repaginate();
}
```

Not all operations require a wait cursor and they may be handled directly:

```
void cut(document *target, selection *selected)
{
    target->cut(selected);
}
```

The choice of which functions should or should not use a wait cursor will be hardwired into the calling code, and it is likely that such assumptions will be duplicated (consistently or otherwise) across a program. Another drawback is for any task described by single function, e.g. `repaginate`, more effort is required in code to set up the call than to execute it. This can also lead to acquisition scopes that are too long. In the following code only the repagination needs a wait cursor, but all statements following it also fall in its shadow:

```
void repaginate(document *target, component *view)
{
    .... // preceding tasks
    cursor_exchange scoped(view, cursor::wait);
    target->repaginate();
    .... // follow-on tasks, also "in waiting"
}
```

Programmers resort to various tricks to narrow the acquisition scope:

```
void repaginate(document *target, component *view)
{
    .... // preceding tasks
    {
        cursor_exchange scoped(view, cursor::wait);
        target->repaginate();
    }
    .... // follow-on tasks, not "in waiting"
}
```

```
void repaginate(document *target, component *view)
{
    .... // preceding tasks
    cursor_exchange(view, cursor::wait), target->repaginate();
    .... // follow-on tasks, not "in waiting"
}
```

The first trick introduces a compound statement whose scope encloses precisely the required lifetime. The second is subtler, relying on the sequential execution of the comma operator and the creation of a temporary `cursor_exchange` object destroyed at the end of the statement.

A PROXY can be introduced to implement the same interface as the document, applying the execute-around actions as necessary:

```
class viewed_document : public document
{
public:
    virtual void cut(selection *selected)
    {
        target->cut(selected);
    }
    virtual void repaginate()
    {
        cursor_exchange scoped(view, cursor::wait);
        target->repaginate();
    }
    ....
private:
    document  *target;
    component *view;
```

```
};
```

This leads to a single point of change and simplified, transparent use:

```
void repaginate(document *target)
{
    target->repaginate();
}
```

Problem

How can you encapsulate executing-around individual function calls on an object conveniently, safely, and transparently? The case of a single member function call on a single object enclosed by pre- and post-call actions is sufficiently common that it demands greater convenience than is perhaps offered by an EXECUTE AROUND OBJECT.

An EXECUTE AROUND OBJECT provides a simple and sure way of wrapping a block of code between pre- and post-sequence actions. However, no compile-time error occurs if the programmer forgets to declare an EXECUTE AROUND OBJECT, so although the risk has been reduced there is still some scope for omission and runtime error. This is more often the case when the actions being executed-around comprise not so much a sequence as a single statement. The preparation to make such a statement safe often outweighs the detail of the single statement.

Although not necessarily a danger, a common pessimizing problem is for the lifetime of the EXECUTE AROUND OBJECT to be longer than necessary, stretching beyond the single statement requiring its attention to the end of the enclosing scope a few statements away. Programmers often resort to tricks to reduce the enclosing lifetime, inserting compound statements or creating temporary objects at the beginning of comma separated expressions.

When the single statement being enveloped by the lifetime of the EXECUTE AROUND OBJECT is a member function call on an object, itself the subject of the EXECUTE AROUND OBJECT's attention, the temptation is to perform the executing-around on the inside of the function called rather than on the outside. However, such a rearrangement may be inappropriate or impossible:

- Although individual function calls would be catered for well, sequences of member function calls would be excluded.
- Different contexts of use may require different paired actions or none at all.
- The source code for the target class may not be available for modification.

However, in favor of such an approach, the executing-around actions can be tailored to each member function as necessary.

Solution

Provide a PROXY that forwards function calls to the target object, enclosing each call between the necessary pre- and post-call actions. The PROXY offers the same core interface as the target object, either by inheritance or mimicry.

The EXECUTE AROUND PROXY introduces another level of indirection and an additional object to allow both the usage code and the target code to be independent of the executing-around actions. This allows actions to be changed without affecting either client or target, or the use of different kinds of EXECUTE AROUND PROXY according to need. As a PROXY, an EXECUTE AROUND PROXY can also take on other roles, such as lazy object loading or reference counting.

An EXECUTE AROUND OBJECT may be used within each PROXY function to simplify implementation and factor out common code. Each forwarding function in the PROXY can take into account the exact pre- and post-call action requirements of functions on the target.

However, the cost of specialization is often generality. Where pre- and post-call actions between many of the forwarding functions is similar it must be repeated. If the actions for each function forwarded are the same, all the functions will follow the same structure, differing only in the target function called. Each EXECUTE AROUND PROXY is written specifically for a single target class or class hierarchy, template based genericity is not supported. If genericity is a genuine issue, and function forwarding is similar for all functions, EXECUTE AROUND POINTER provides an alternative to EXECUTE AROUND PROXY.

EXECUTE AROUND POINTER

Provide a SMART POINTER object that executes actions before and after each function call on an object, given that the actions performed are the same for all functions.

Example: Loading and Saving State

Revisiting a previous example, consider a slightly different interface to the `table` class:

```
class table
{
public:
    void lock() const;   // acquire lock for reading
    void lock();         // acquire lock for writing
    void unlock() const; // release read lock
    void unlock();       // release write lock
    void load() const;
    void save();
    bool changed() const;
    value_type query(key_type) const;
    void modify(key_type, value_type);
    void reset(key_type);
    ....
};
```

The locking functions are as before, but now one more modifier function, `reset`, has been shown and a query function, `changed`, has been added. `changed` returns `true` if the state of the object has changed since it was loaded. The result of manipulating the object, through either `modify` or `reset`, may be the same state as before, in which case a call to `changed` should return `false`. The `table_view` class can be modified to avoid unnecessary saves:

```
class table_view
{
public:
    table_view(table *to_view)
      : target(to_view)
    {
    }
    value_type query(key_type key) const
    {
        locker<const table> critical(*target);
        target->load();
        return target->query(key);
    }
    void modify(key_type key, value_type value)
    {
        locker<table> critical(*target);
        target->load();
        target->modify(key, value);
        if(target->changed())
            target->save();
    }
    void reset(key_type key)
    {
        locker<table> critical(*target);
        target->load();
        target->reset(key);
        if(target->changed())
            target->save();
    }
    ....
private:
    table *target;
};
```

The *lock–load–change–save-if-changed–unlock* code structure will be repeated for each non-`const` function, and the *lock–load–query–unlock* structure for each `const` function. Such code repetition suggests refactoring is necessary, but the only standard refactoring possible in `table_view` is to collapse the *save-if-changed* code into a single function, reducing two lines of code to one. The basic repetition survives.

Another issue that may arise is the coupling of the code in the PROXY class, `table_view`, to the target class, `table`. With the exception of the forwarded functions the mechanical details are independent of the specific target type. Whether or not this matters to the programmer depends on how commonly such a loading/saving framework is used and reused. However, the question of genericity contains the seeds of a solution to factoring out common code. A SMART POINTER is a specialized form of PROXY that presents a familiar syntax to programmers and can be used to factor out common access code:

```
value_type reading(loading_ptr<const table> data, key_type key)
{
    return data->query(key);
}
```

```
void writing(loading_ptr<table> data, key_type key)
{
    data->reset(key);
}
```

The `loading_ptr` can overload `operator->` to perform the locking and loading as necessary. Alas, there is a problem with saving and unlocking: Who performs these actions? They must take place after the target function, e.g. `reset`, has been called, which is after the result of `operator->` has been returned but before the end of the statement.

The lifetime of a temporary object is until the end of the full expression in which it is created, which conventionally means it is destroyed at the end of the statement. If `operator->` returns a temporary object instead of a raw pointer the destruction of the temporary can perform the necessary clean-up actions after the function call. The temporary can also provide its own `operator->` to perform the actions before hand. `operator->` chains, meaning that the `operator->` on the temporary will be called automatically when the `operator->` on the SMART POINTER is called:

```
template<typename type>
class loading_ptr
{
public:
    loading_ptr(type *to_load)
      : target(to_load)
    {
    }
    tmp_loader<type> operator->() const
    {
        return tmp_loader<type>(target);
    }
    ....
private:
    type *target;
};
```

The `loading_ptr` class delegates responsibility for executing-around to a temporary object whose existence is transparent to the user:

```
template<typename type>
class tmp_loader
{
public:
    explicit tmp_loader(type *to_load)
      : target(to_load), is_locked(false)
    {
    }
    ~tmp_loader()
    {
        if(is_locked)
        {
            save_if_changed(target);
```

```
            target->unlock();
        }
    }
    type *operator->()
    {
        target->lock();
        is_locked = true;
        target->load();
        return target;
    }
private:
    type *target;
    bool is_locked;
};
```

Overloading can be used to differentiate between `const` target objects, which need not be saved, and non-`const` target objects, which need to be checked before being saved:

```
template<typename type>
void save_if_changed(type *target)
{
    if(target->changed())
        target->save();
}
template<typename type>
void save_if_changed(const type *)
{
}
```

Problem

How do you define an EXECUTE AROUND PROXY when the execute-around actions performed on a target object are the same for all functions? An EXECUTE AROUND PROXY simplifies the safe use of individual member functions on an object that must have an EXECUTE AROUND OBJECT applied to it for correct use.

If the pre- and post-call action code for each forwarding function is the same or similar (within the reach of similarity defined for overloading), the only difference between functions is the forward function call itself, which differs in name and arguments. Such code repetition is tedious and suspicious, easily falling prey to the familiar problems of copy-and-paste coding.

Making the code generic in some way is attractive, but an EXECUTE AROUND PROXY class is tied to the target class or class hierarchy it is written for. It cannot be made generic with respect to the target class because each function in the target must have a corresponding forwarding function in the PROXY.

The only PROXY realization that supports genericity is the SMART POINTER. SMART POINTERS provide a natural working idiom for programmers, with familiar syntax and tradeoffs. Alas, one of the tradeoffs is that the only common action that can be factored out in the execution of `operator->` is one that executes on dereferencing the target object for use. `operator->` cannot execute actions after it returns.

Solution

Define a SMART POINTER whose `operator->` returns a temporary SMART POINTER [Henney1996, Stroustrup2000]. The temporary's `operator->` performs the action before dereferencing for any member function call, and its destructor performs the action after. The temporary lives until the end of the full expression in which it is created. This encloses the member function call, but may wrap anything executed in the same expression after it.

An EXECUTE AROUND POINTER works in part because calls to `operator->` are chained automatically by the compiler until a raw pointer type is returned. One consequence of the language and this design is that `operator*` cannot be supported with the same semantics as it does not chain, and there is no way of overloading `operator.`. However, common mechanism is factored out for all calls, and this can be made generic.

An EXECUTE AROUND POINTER is a specialized form of EXECUTE AROUND PROXY, and therefore shares many of the same consequences. For instance, as a SMART POINTER other roles may be added to it, such as reference counting for object lifetime management.

EXECUTE AROUND FUNCTION

Define a function that executes actions before and after the execution of a sequence of statements passed in as an object or function.

Example: Recoverable Objects

A transaction is a set of actions taken together that have either a successful outcome, changing the state of the system in an atomic, durable, and predictable fashion, or an unsuccessful outcome, in which case the system's state is as it was before the transaction. An object whose changes are either committed or rolled back as a result of a transaction is said to be a *recoverable object* [COSS]. Interpreted in the context of C++, with thrown exceptions indicating failure, a recoverable object offers the *strong guarantee* of exception safety. A general interface for recoverable types could be captured as

```
class recoverable
{
public:
    virtual void begin() = 0;
    virtual void commit() = 0;
    virtual void rollback() = 0;
    ....
};
```

Classes for recoverable objects can implement this interface by inheritance along with the relevant interface for using the object. Here is an example of an arbitrary resource class with these semantics:

```
class recoverable_resource
  : public resource, public recoverable
{
public:
    virtual void begin()
    {
        old_self = new body(*self);
    }
    virtual void commit()
    {
        delete old_self;
        old_self = 0;
    }
    virtual void rollback()
    {
        delete self;
        self = old_self;
        old_self = 0;
    }
    .... // functions for use between begin and commit/rollback
private:
    struct body
    {
        ....
    };
    body *self, *old_self;
};
```

The HANDLE/BODY representation is used to support the commit-or-rollback semantics. Code to check constraints has been omitted for brevity, for instance whether or not the object is already within a transaction, or to ensure that a transaction is not initiated when the stack is unwinding from an existing exception. A `std::stack` could be used to support transaction nesting rather than the flat model supported above with a single pointer, `old_self`. A thrown exception is taken to mean failure, and hence a need to rollback:

```
void transaction(recoverable_resource *target)
{
    target->begin();
```

```
    try
    {
        .... // sequence of actions involving target
    }
    catch(...)
    {
        target->rollback();
        throw;
    }
    target->commit();
}
```

This code fully takes exceptions into account, branching on the outcome. The user is required to call the pre- and post-sequence actions correctly otherwise the state of the object, and possibly the system, can become corrupted.

An EXECUTE AROUND OBJECT encapsulates the actions and decisions in the code above:

```
class recoverer
{
public:
    recoverer(recoverable *to_recover)
      : target(to_recover)
    {
        target->begin();
    }
    ~recoverer()
    {
        if(std::uncaught_exception())
            target->rollback();
        else
            target->commit();
    }
private:
    recoverable *const target;
};
```

The revised sample code for using a recoverable object safely becomes

```
void transaction(recoverable_resource *target)
{
    recoverer scoped(target);
    .... // sequence of actions involving target
}
```

This is a more memorable and safer model for the programmer. It still has the disadvantage that that there is still nothing to stop the programmer forgetting to use an EXECUTE AROUND OBJECT to keep their resource safe, or to prevent the programmer misusing the `begin`, `commit`, and `rollback` functions in the public interface.

The simplest way to prevent a public user accessing a feature is to make it private. This applies to both the `recoverable` and `resource` interface classes:

```
class recoverable_resource
  : private resource, private recoverable
{
public:
    ....
private:
    .... // all functions previously public are now private
};
```

At first sight this perhaps seems a little drastic: All features of interest are now private. Being prevented from any form of use at all prevents users from misusing features! The current object can use the private functions, as can any function to which it passes the `this` pointer. Thus, when a hook for callbacks is introduced the solution is more liberal than it appears initially:

```
class recoverable_resource ....
{
public:
```

```
    template<typename unary_function>
    void apply(unary_function callback)
    {
        recoverer scoped(this);
        callback(this);
    }
    ....
};
```

A function or FUNCTION OBJECT may be defined to execute against a resource object:

```
void transaction(resource *target)
{
    .... // sequence of actions involving target
}
```

Given a pointer to a `recoverable_resource` object, `target`, executing the `transaction` function is simply a matter of setting up a callback:

```
target->apply(transaction);
```

Problem

How do you safely group and execute a given sequence of statements that must be enclosed by a pair of actions, or followed by a single action? An EXECUTE AROUND OBJECT can perform pre- and post-sequence actions in the general case, but it cannot treat statements as a group in their own right, i.e. a block of code to be manipulated and passed around.

Furthermore, there is little that can be checked at compile-time to ensure a programmer has remembered to set up an EXECUTE AROUND OBJECT in order to run a particular sequence. It is easy to forget, until the bugs start appearing at runtime.

EXECUTE AROUND PROXY and EXECUTE AROUND POINTER provide methods for wrapping member function calls, which may themselves be groups of actions. However, these wrap only existing functions on a target object. To extend the kinds of action that may be performed conveniently requires the definition of several, incompatible, extended EXECUTE AROUND PROXY classes.

Solution

Provide a function or object that holds the sequence of statements and pass it to another function for execution. The called function executes the pre- and post-sequence actions as necessary around a callback, ensuring the atomicity of grouped operations. Any context necessary for the sequence to execute must be passed in as part of the callback.

As an object, the sequence of statements can be encapsulated in a COMMAND class. A more idiomatic variation of the COMMAND pattern in this context is the FUNCTION OBJECT pattern, where an object supports function call syntax by overloading `operator()`. Operator overloading makes the distinction between FUNCTION OBJECTs and function pointers transparent. This transparency supports generic programming and the use of templates, so that the EXECUTE AROUND FUNCTION itself may be templated with respect to whatever functional abstraction it is passed.

The EXECUTE AROUND FUNCTION may be either a member or a global function, depending on whether or not a target object is involved as the subject of the execute-around actions and the sequence in between. EXECUTE AROUND METHOD [Beck1997] documented the idiom originally for Smalltalk in terms of a distinguished target object and a block of code passed in as an object. A similar technique can be used in Java taking advantage of its inner class feature.

EXECUTE AROUND FUNCTION applies in many of the same circumstances as EXECUTE AROUND OBJECT. The responsibilities are similar, but the control-flow and object structures are effectively the inverse of one another, and the tradeoffs are different. EXECUTE AROUND FUNCTION relies on the naming and encapsulating of usage sequences, often at the expense of class or function proliferation. EXECUTE AROUND OBJECT is more flexible and often lighter in use, but it does not encapsulate the sequence as fully or succinctly. An EXECUTE AROUND FUNCTION works well when other member functions on an object already perform their own executing-around and a hook for more general functions is required. An EXECUTE AROUND OBJECT may be used in the implementation of an EXECUTE AROUND FUNCTION.

The Patterns in Practice

Concurrency, in the form of multiple threads or multiple processes, introduces a design context unfamiliar to many developers, and one fraught with subtleties. If the consequences of concurrency are not fully appreciated (i.e. the developer must genuinely grok them rather than pay lip service to them), this subtle design context becomes a subtle debugging context.

The worked example presented here explores the issues involved in creating a thread-safe queue class. The purpose of the queue is to buffer tasks supplied by one or more producers to consumers that then handle the tasks. All the producers and consumers are executing in different threads, requiring thread-safe access of the queue. The control model is push–pull, i.e. the producer pushes tasks into the queue and the consumer pulls them from the queue.

The queue is implemented as a concrete class, `queue`. This should not be confused with `std::queue` in the standard library, which may be used as part of the underlying implementation. Here is the initial thread-unsafe implementation:

```
class queue
{
public:
    void enqueue(task *new_task)
    {
        if(new_task)
            contained.push(new_task);
    }
    task *dequeue()
    {
        task *result = 0;
        if(!contained.empty())
        {
            result = contained.front();
            contained.pop();
        }
        return result;
    }
    size_t size() const
    {
        return contained.size();
    }
    ....
private:
    std::queue<task *> contained;
};
```

The worked example evolves to demonstrate how different context and usage affect the design decisions taken, drawing the presented patterns together. It is not intended to demonstrate a linear sequence of refinement, i.e. there is no one correct solution.

Detached Locking

Associating each shared `queue` object with a mutex is the simplest approach to working with the class in a threaded environment. Borrowing the `mutex` class defined in an earlier example:

```
queue tasks;
mutex tasks_guard;
```

For global `queue` objects the associated `mutex` object should also be visible. In a realistic application, objects will be passed around discretely and purposefully as arguments, rather than permissively and casually sitting in global or namespace scope:

```
void enqueue(queue *tasks, mutex *guard, task *new_task)
{
    guard->lock();
    tasks->enqueue(new_task);
```

```
    guard->unlock();
}
```

Introducing an EXECUTE AROUND OBJECT simplifies and secures the example code:

```
void enqueue(queue *tasks, mutex *guard, task *new_task)
{
    locker<mutex> critical(*guard);
    tasks->enqueue(new_task);
}
```

Attached Locking

The detached model has the apparent benefit of not over-committing the implementation and the interface of the `queue` class. The code is not tied to any multi-threading classes. However, the class is intended for use in a multi-threaded environment, so this decoupling is not as beneficial as it first appears. To use a `queue` object correctly and safely a `mutex` object must always be passed around with it. The programmer must manage more objects – more objects to be created and destroyed, more objects to be remembered, more objects to passed around, more scope for error. This association between queue and synchronization is not coincidental and presents a good target for encapsulation:

```
class queue
{
public:
    void lock() const
    {
        guard.lock();
    }
    void unlock() const
    {
        guard.unlock();
    }
    .... // other functions as before
private:
    .... // other representation as before
    mutable mutex guard;
};
```

The resulting class is more cohesive, containing its own synchronization mechanism and offering a lockable interface that may be used with the existing `locker` class:

```
void enqueue(queue *tasks, task *new_task)
{
    locker<queue> guarded(*tasks);
    tasks->enqueue(new_task);
}
```

The encapsulation of synchronization detail offers the `queue` client independence from representation and affords its supplier freedom to change. Locking and unlocking do not affect the object's logical state, hence the use of `const` member functions and `mutable` representation.

Proxied Locking

If calling a single operation at a time is how a `queue` object is commonly used, declaring named locking objects for each call becomes tedious and error-prone. It is too easy to forget that the queue needs locking:

```
void enqueue(queue *tasks, task *new_task)
{
    tasks->enqueue(new_task); // unlocked and unsafe
}
```

The action of some functions may already be atomic, requiring no further synchronization. For instance, let us assume that on the code generated on the target platforms for `queue::size` is not subject to race conditions that would lead to thread contention. The class user must be aware of when such an assumption is (or is not) valid so as to forego (or include) synchronization code:

```
size_t size(const queue *tasks)
{
    return tasks->size(); // unlocked and safe
}
```

An EXECUTE AROUND PROXY can hide such details:

```
class queue_proxy
{
public:
    queue_proxy(queue *to_proxy)
      : target(to_proxy)
    {
    }
    void enqueue(task *new_task)
    {
        locker<queue> critical(*target);
        target->enqueue(new_task);
    }
    size_t size() const
    {
        return target->size();
    }
    ....
private:
    queue *target;
};
```

This class presents the user with a simpler, safer, more consistent view of queue usage:

```
void enqueue(queue_proxy *tasks, task *new_task)
{
    tasks->enqueue(new_task);
}
```

```
size_t size(const queue_proxy *tasks)
{
    return tasks->size();
}
```

An interface base class can be factored out between the proxy and actual queue class to make the substitutability between them more transparent. Once a proxy has been introduced, it can acquire other responsibilities, such as object lifetime management through reference counting.

Pointer Locking

An alternative approach to single-call locking is to wrap each use of a `queue` pointer in a temporary object that handles the executing-around. The appearance is that of a function call and access to the now-locked object is granted by having the locker act as a SMART POINTER:

```
void enqueue(queue *tasks, task *new_task)
{
    lock(tasks)->enqueue(new_task);
}
```

The benefit this has over an EXECUTE AROUND PROXY is that it may be made generic. The type doing the real work can be templated and a `typedef` provided for syntax sugaring:

```
typedef tmp_locker<queue> lock;
```

The underlying code will work for any other class supporting `lock` and `unlock` functions:

```
template<typename lockee>
class tmp_locker
{
public:
    explicit tmp_locker(lockee *to_lock)
      : target(to_lock), is_locked(false)
    {
```

```
    }
    ~tmp_locker()
    {
        if(is_locked)
            target->unlock();
    }
    lockee *operator->()
    {
        target->lock();
        is_locked = true;
        return target;
    }
private:
    lockee *target;
    bool   is_locked;
};
```

Whilst this direct use has a certain charm and generality, it is not as transparent as an EXECUTE AROUND PROXY. An EXECUTE AROUND POINTER retains many benefits from each approach:

```
template<typename lockee>
class locking_ptr
{
public:
    tmp_locker<lockee> operator->() const
    {
        return tmp_locker<lockee>(target);
    }
    ....
private:
    lockee *target;
};
```

Programmers should be careful about attempting to access the same object twice in a statement using `locking_ptr`s: This will cause deadlock if the synchronization mechanism does not allow recursive locking, i.e. a thread reacquiring a lock it already owns. A potential efficiency issue is that there is no way of selecting individual function calls for special treatment, e.g. omitting locks for certain queries.

Internal Locking

All the synchronization so far has been external to the `queue` object. An alternative approach is to make the queue self-locking, i.e. make it behave like a monitor:

```
class queue
{
public:
    void enqueue(task *new_task)
    {
        locker<mutex> critical(guard);
        contained.push(new_task);
    }
    ....
private:
    std::queue<task *> contained;
    mutable mutex      guard;
};
```

The class and its public member functions encapsulate the locking mechanics, in a similar way to `synchronized` methods in Java. This radically simplifies the programmer's view of the class:

```
void enqueue(queue *tasks, task *new_task)
{
    tasks->enqueue(new_task);
}
```

The safety of a `queue` object's behavior is now entirely its own concern. The class author must be careful not to call other synchronized member functions from within the object if the synchronization object is not recursive. Similarly, the class author must respect the fact that C++ supports class-level rather than object-

level encapsulation; it is possible for an object to access the private members of another object of the same class and accidentally bypass the synchronized interface ordinary class users would use.

Self-locking works transparently for single isolated calls, but how can multiple tasks be enqueued or dequeued without interruption? A self-locking BATCH FUNCTION that operates on sequences rather than just a single item provides a solution. In the style of the standard library, overloaded versions of `enqueue` and `dequeue` work over iterator ranges rather than specific sequence classes:

```
class queue
{
public:
    template<typename task_iterator>
      void enqueue(task_iterator begin, task_iterator end);
    template<typename task_iterator>
      void dequeue(task_iterator begin, task_iterator end);
    ....
};
```

There is still an issue if `task` objects that are enqueued together must be dequeued and handled together: A BATCH FUNCTION allows multiple enqueuing and dequeuing, but the number dequeued together need not be the number enqueued together. A COMPOSITE has these properties and provides a suitable alternative that does not require modification to the `queue` class interface:

```
class composite_task : public task
{
    ....
private:
    vector<task *> sequence;
};
```

Internal locking simplifies many common uses of a queue, e.g. single or multiple enqueuing and dequeuing. However, it does not work well for arbitrary use. For instance, in the following example the intent is to dequeue a task, use it, and then requeue it, all without interruption:

```
void requeue(queue *tasks)
{
    task *task_in_hand = tasks->dequeue();
    .... // handle task_in_hand
    tasks->enqueue(task_in_hand);
}
```

External locking accommodates this easily with the lifetime of an EXECUTE AROUND OBJECT corresponding to the scope of use. If internal locking is the preferred strategy, and synchronization is recursive, an EXECUTE AROUND FUNCTION provides a hook for achieving the same effect:

```
class queue
{
public:
    template<typename unary_function>
    void apply(unary_function callback)
    {
        locker<queue> critical(guard);
        callback(this);
    }
    ....
};
```

The arbitrary task may now be defined as either a function or a FUNCTION OBJECT class:

```
void requeue(queue *tasks)
{
    task *task_in_hand = tasks->dequeue();
    .... // handle task_in_hand
    tasks->enqueue(task_in_hand);
}
```

The function is now passed into the target object for application:

```
tasks->apply(requeue);
```

Other Patterns

Table 2 presents other key patterns that are used in this paper. The references given indicate where the pattern has been formally documented as such or, alternatively, where it has been documented as a proven, recognizable practice, possibly by a different name.

Name	*Problem*	*Solution*
BATCH FUNCTION [COSS, Henney1999]	How can a sequence, that is a repetition of an action, be treated atomically?	Define a single function that performs the action repeatedly. The function is declared to take all the arguments for each execution of the action, e.g. an array or iterator, and to return results by similar means.
COMBINED FUNCTION [COSS, Henney2000]	How can a common sequence of actions be treated atomically?	Define a single function that performs all the actions together, managing any atomicity issues. The function is declared to take all the arguments and return the results required by its component actions.
COMMAND [Gamma+1995]	How can selection of functionality be decoupled from its execution?	Represent the function as an object, providing it with the any necessary context at construction.
COMPOSITE [Gamma+1995]	How can a client treat individual objects and groups of objects uniformly?	Define a common interface for individual objects and groups, such that a group holds other objects via the common interface, forwarding requests as necessary.
FUNCTION OBJECT [Coplien1992, ISO1998]	How can an object that essentially defines its behavior through a single function be used transparently alongside conventional functions in generic algorithms?	Define the single function as `operator()`, supporting conventional function call notation. Ensure that the object is accessed directly rather than indirectly, i.e. without an extra level of indirection, otherwise provide a SMART POINTER to it that also supports `operator()`.
HANDLE/BODY [Coplien1992, Gamma+1995]	How can the representation of an object be decoupled from its usage?	Place the abstraction and representation into separate objects and hierarchies, so that the abstraction is accessed via a handle object and its representation is a separate, hidden, body object.
PROXY [Buschmann+1996, Gamma+1995]	How can a client transparently communicate with a target object when the communication must be managed?	Provide a proxy that stands in for the actual target object, forwarding and managing requests to the target as necessary.
SMART POINTER [Meyers1996, Stroustrup1994, Stroustrup1997]	How should a PROXY be defined where control on the target is the same for access to any of its members and no actions are required after the request has been forwarded?	Define a class that supports conventional pointer operations, e.g. `operator*` and `operator->`, so that access to the target object is provided but is also managed.

Table 2. Thumbnails for patterns used but not documented in this paper, listed alphabetically.

Acknowledgments

My thanks to John Vlissides for his patient shepherding and feedback, and Jon Jagger and Mark Radford for their reviewing.

References

[Beck1997] Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.

[Boost] Boost library website, `http://www.boost.org`.

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

[Cargill1996] Tom Cargill, "Localized Ownership: Managing Dynamic Objects in C++", *Pattern Languages of Program Design 2*, edited by John M Vlissides, James O Coplien, and Norman L Kerth, Addison-Wesley, 1996.

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

[COSS] *CORBAservices: The Common Object Services Specification*, OMG, `http://www.omg.org`.

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

[Glassborow2000] Francis Glassborow, "Tower of Babel", *EXE* 14(12), May 2000.

[Henney1996] Kevlin Henney, "C++ Advanced Design Issues: Asynchronous C++", presented at *Visual Tools Developers' Academy*, Oxford, September 1996.

[Henney1998] Kevlin Henney, "Creating Stable Assignments", *C++ Report* 10(6), June 1998.

[Henney1999] Kevlin Henney, "Patterns Inside Out", presented at *Application Development '99*, July 1999.

[Henney2000] Kevlin Henney, "C++ Patterns: Principles, Idioms and Techniques", presented at *OOP 2000*, January 2000.

[ISO1998] *International Standard: Programming Language - C++*, ISO/IEC 14882:1998(E), 1998.

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

[Murray1993] Robert B Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993.

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

[Stroustrup1997] Bjarne Stroustrup, *C++ Programming Language*, 3rd edition, Addison-Wesley, 1997.

[Stroustrup2000] Bjarne Stroustrup, "Wrapping C++ Member Function Calls", *C++ Report* 12(6), June 2000.

[Sutter2000] Herb Sutter, *Exceptional C++*, Addison-Wesley, 2000.

[Taligent1994] *Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++*, Addison-Wesley, 1994.