Weaving a Way Past the C++ One Definition Rule

Edward D. Willink[1] and Vyacheslav B. Muchnick[2]

[1]Racal Research Limited, Worton Drive, Reading, England
Ed.Willink@rrl.co.uk

[2]Department of Computing, School of Electronic and Electrical Engineering,
Information Technology and Mathematics, University of Surrey, Guildford, England
V.Muchnick@ee.surrey.ac.uk

Position Paper for the Aspect Oriented Programming Workshop at the
European Conference on Object Oriented Programming (ECOOP'99), Lisbon, June 14, 1999

Abstract

Aspect Orientation requires that different programming concerns be separated, and often results in code that cuts across classes. The One Definition Rule of C++ prevents code being organised by aspects. In this position paper we show by way of example how meta-level facilities and syntax generalisations provide the requisite weaving to bridge the gap between the programmer's and compiler's organisational requirements.

1 Introduction

The [C++] One Definition Rule (ODR) requires declarations to occur just once and consequently means that classes are closed: it is not possible to extend a class declaration. This prevents code that cuts across classes being organised by aspects.

In this paper, we show how introduction of a pre-processing (or meta-compilation) stage can remove the problems that the ODR causes for Aspect Orientation.

We first very briefly describe the Flexible Object Generator (FOG), a meta-compiler that performs the requisite weaving of multiple definitions so that the resulting C++ code satisfies the ODR.

We then rework the well known example of a monitor [Hoare74] to show how the synchronisation aspect can be completely separated once the ODR is eliminated from, and meta-level programming added to, C++.

2 FOG

FOG is a meta-compiler for C++ supporting a pure super-set of C++. No new reserved keywords are added and existing semantics are unchanged. New keywords are introduced in a non-reserved fashion and meaning is given to some syntax that has no C++ meaning.

FOG is a source to source translator and reorganiser that converts super-set C++ source files to conventional C++ interface and implementation files.

FOG provides a static meta-level and renders Cpp (the C preprocessor) redundant. Meta-functions and meta-variables replace function-like and object-like macros. Meta-statements replace conditionalisation. Meta-classes, meta-iterators, meta-constructors, and meta-destructors support meta-programming and reflection. The syntax used for invocation of meta-functions and meta-variables supports lexical concatenation and stringizing thereby replacing the ## and # functionality of Cpp.

FOG supports Aspect Orientation by replacing the One Definition Rule (ODR) of C++ by a Composite Definition Rule (CDR). The CDR merely requires that the ODR be satisfied once all compatible declarations have been woven together.

The meta-level compile-time operation of FOG is very similar to the conventional run-time operation of C++. A brief explanation of each novel FOG syntactical usage is relegated to footnotes in this paper. A fuller explanation of FOG together with source, NT and Solaris executables can be found at

`http://www.ee.surrey.ac.uk/Research/CSRG/fog`

3 Monitor Example

The separation of concerns [Aksit96] will be demonstrated by applying a monitor aspect to a stack.

3.1 Application Aspect

We first define a simple stack class.

```
temaplate <class T>
class Stack
{
private:
   T *_elements;
   size_t _capacity;    // Allocated size of _elements[]
   size_t _tally;       // Used size of _elements[]
private:
   Stack(const Stack&);           // Not implemented
   Stack& operator=(const Stack&);// Not implemented
public:
   Stack() : _elements(0), _capacity(0), _tally(0) {}
   ~Stack() { /* ... */ }
   bool is_empty() const volatile { return _tally == 0; }
   T pop() { /* ... */ }
   void push(const T&) { /* ... */ }
   T top() const { return _elements[_tally-1]; }
};
```

The `const` qualifier is used conventionally to indicate that no change occurs. Concurrent readers are therefore permissible, but concurrent writing should not be permitted once a monitor aspect has been added.

The `volatile` qualifier is used to indicate that access may occur without the use of a lock.

Interleaved reading and writing by other threads is permitted. The qualification of `is_empty` as `volatile` as well as `const` goes beyond conventional practice.

Its use is safe because the implementation involves a single read, whereas `top` involves at least two reads. Its utility is limited, since the return accurately reflects a state that existed but may no longer exist by the time calling code interprets the result. Of course removal of the `volatile` qualifier would make no difference, since any lock should encompass both a `!is_empty()` and a subsequent `pop()`. The `is_empty` method is of use only in a polling loop that can recover on the next iteration. The presence of `volatile` avoids incurring locking costs for such a loop.

The usage is consistent because `volatile` indicates that concurrent change may occur and so inhibits any optimisation that could reorder the sequence of memory accesses.

3.2 The Monitor Aspect (run-time)

The monitor functionality is provided by a `Monitor` class, whose detailed implementation is not relevant to this paper.

```
class Monitor
{
   friend class ReadOnlyLock;
   friend class ReadWriteLock;
private:
   void acquire_exclusive() { /* ... */ }
   void acquire_shared() { /* ... */ }
   void release() { /* ... */ }
};
```

`acquire_exclusive` and `acquire_shared` block until exclusive or shared access is available to the resource(s) managed by the monitor. `release` terminates the resource reservation.

Reservation of the monitored resource is managed by a pair of nested lock classes, `ReadOnlyLock` and `ReadWriteLock`. They differ only in whether `acquire_shared` or `acquire_exclusive` is invoked.

```
public[1] class Monitor::ReadOnlyLock
{
private:
   Monitor& _monitor;
public:
   ReadOnlyLock(Monitor& aMonitor)
      : _monitor(aMonitor) { _monitor.acquire_shared(); }
   ~ReadOnlyLock() { _monitor.release(); }
};
```

A lock class invokes `acquire_shared` to acquire the resource during construction and ensures its release from the destructor, whose invocation C++ guarantees.

3.3 Aspect composition

The application code above is written independently of the synchronisation code. The presence of the `volatile` keyword is an optional optimisation.

The monitor aspect is added to the application aspect by providing additional declarations that are woven into the application code.

```
using[2] "monitor.fog";        // A better form of #include

template <class T>
class Stack
{
   $[3]Monitor::install();     // Invoke meta-function
};
```

Weaving results from the replacement of the ODR by the CDR. The additional appearance of `template <class T> class Stack {};` indicates additional declarations for the class, not a conflicting redeclaration[4].

3.4 The Monitor Aspect (compile-time)

The remainder of the code for this example is provided as part of the monitor.fog include file.

Invocation of `Monitor::install` invokes the `install` meta-function of the `Monitor` meta-class[5].

```
auto[6] declaration[7] Monitor::install()     //1
{
   class $This[8] : auto[9] $Dynamic[10]      //2
   {
      private[1] $Dynamic[10] _monitor;       //3
      auto[6] number has_monitor = true;      //4
   };
}

auto number Monitor::has_monitor = false; //5
```

`//1` declares the compile-time meta-function.
`//2` adds the `Monitor` class as a meta-base class of `Stack`.

```
   template <class T> class Stack : auto Monitor { ... }
```

`//3` adds a private instance of the `Monitor` to the `Stack` class.

```
   private:
      Monitor _monitor;
```

`//4` defines a meta-variable whose truth flags the presence of `_monitor` in `Stack` and its derived classes.
`//5` defines a default `false` value for the flag meta-variable. The `false` value is visible in `Monitor` and

1. FOG permits an *access-specifier* as part of a *decl-specifier*, so that the distinction between definitions and declarations is removed in order to allow all components of a 'declaration' to accompany a 'definition'. This avoids the need for interface declarations that duplicate subsequent definitions.

2. FOG provides an include only once replacement for the traditional Cpp `#include` preprocessor directive. This avoids the need for include file guards.

3. A $ expression such as any of `$var`, `$func(x,y,z)`, `$Nested::Meta::function(q)`, `$aClass.bases()`, `$anIterator->is_const()` or `$*anIterator` causes the value of the meta-variable, meta-function or meta-expression to replace $ invocation. An alternate `${...}` form may be used when ambiguity could arise with respect to following characters.

4. Classes, enumerations, arrays, and functions are all open, that is they may be extended by additional declarations.

5. Every class and built-in type has a corresponding meta-class for use at compile time. Meta-classes observe the same inheritance relationship as their class counterparts.

6. FOG reuses the redundant `auto` keyword to define meta functionality. The usage indicates that the following function is a meta-function at `//1` and a meta-variable at `//4`.

in derived scopes, except where the value is overridden by the `true` value provided by `//4`.

The extra `_monitor` member variable provides the run-time support for the monitor aspect. It now remains to add the run-time code to use `_monitor`.

FOG provides two hooks that may be used by compile-time code to reflect upon the prevailing declarations. Each class may have a meta-constructor that may create additional declarations, and a meta-destructor that may refine existing declarations. For the monitor aspect we need to refine some function declarations to add lock acquisition. We therefore define a meta-destructor, which is invoked for the `Monitor` meta-class and for the derived `Stack` meta-class.

```
auto Monitor::~Monitor()
{
   if (has_monitor)  // Avoid execution for inappropriate
   {                 // base and derived classes.
      for (iterator[11] f = functions[12](); f; ++f)
      {
         if (f->is_static[13]())      // static functions
            ;                         // have nothing to lock
         else if (f->is_volatile())// volatile functions
            ;                         // need no lock
         else if (f->is_const())   // const functions
         {                            // need a shared lock
            $f->signature[14]()
               [[entry]][15] { ReadOnlyLock aLock(_monitor);}
         }
         else                         // non-const functions
         {                            // need exclusive lock
            $f->signature()
               [[entry]] { ReadWriteLock aLock(_monitor); }
         }
      }
   }
} [[derived tree]][16];
```

The meta-destructor first tests the `has_monitor` flag, which is `false` when executing for the `Monitor` class, but `true` for the `Stack` class. This test avoids application of a lock to support methods such as `Monitor::release`. Once the class validation is over a `for` loop iterates over the declared functions and does nothing if the function is `static` or `volatile`. Once the degenerate cases have been bypassed, a `ReadOnlyLock` or `ReadWriteLock` is applied according to whether the function is `const` or not.

3.5 Function Weaving

FOG performs function weaving by concatenating the code from multiple function bodies. In order to provide adequate flexibility for practical applications concatenation is performed in five distinct named regions each of which are concatenated to yield the overall function body. The five named regions are concatenated in the order `entry`, `pre`, `body`, `post` and `exit`.

The `entry` and `exit` regions typically contain local variable initialisations and a `return` statement necessary to define the overall structure of the function. `pre` and `post` regions provide for support of code that performs pre-condition or post-condition operations around the default `body` region.

Insertion of the lock into the `top` function therefore occurs by weaving the following pair of declarations, that are represented here in simplified form showing how the `$` expressions are resolved.

```
template <class T>
T Stack::top() const        // from application aspect
   { return _elements[_tally-1]; }
```

and

```
template <class T>
T Stack::top() const        // from the meta-destructor
   [[entry]] { ReadOnlyLock aLock(_monitor); }
```

are woven to generate the final C++ result:

```
template <class T>
T Stack::top() const
{
#line ...
   ReadOnlyLock aLock(_monitor);
#line ...
   return _elements[_tally-1];
}
```

This form of weaving by concatenation is highly pragmatic enabling responsibly written code to be combined. However as indicated by [Ossher98], there is considerable opportunity for incorrect composition. FOG requires that multiple function declarations share the same spelling of template and function parameter names. FOG also permits mismatching code, so that a `for` loop may start in a `pre` region and finish in a `post` region. Non-trivial

7. Meta-functions and meta-variables use meta-types in the same way as functions and variables use types. All meta-types are built-in. The meta-types include `identifier`, `number` and `string` which correspond to basic lexical elements, `declaration` and `expression` which correspond to standard grammar productions, and polymorphic meta-types such as `iterator` and `list` to support meta-programming.

8. The built-in meta-variable `This` resolves to the current declaration scope, which in subsequent usage is the class `Stack`.

9. The `auto` keyword as part of a *base-specifier* indicates that the subsequent class is a meta-base-class, that is an extension to the inheritance of the meta-class (at compile-time). The conventional class inheritance at run-time is unaffected.

10. The built-in meta-variable `Dynamic` resolves to the current execution scope, which in this case is `Monitor`, but would be a derived class if a derived monitor were in use.

11. The polymorphic built-in meta-type `iterator` supports iteration over a list of meta-entities: functions in the example. The meta-type supports normal pointer operations such as `operator*`, `operator->` and `operator++`.

12. The built-in `list class::functions()` meta-function returns a list of all functions for the current scope.

13. The `number object::is_static()`, `is_const()`, `is_volatile()` predicates return `true` or `false` according to the attributes of the associated object.

14. The `string function::signature()` meta-function returns the entire declaration less any function body.

15. The `[[entry]]` declaration indicates that the subsequent *function-body* contributes to the `entry` region of the eventual function.

16. `[[derived tree]]` is a derivation rule specifying that the associated declaration (the meta-destructor) is to be repeated throughout the inheritance tree rooted at the definition scope (`Monitor`). Derivation rules are beyond the scope of this paper. The usage is necessary here only because meta-destructors are not inherited by default.

function body weaving is perhaps best restricted to co-operating programmers. The example in this paper is relatively safe, since the functions are parameterless, however the introduction of the symbols `_monitor` and `aLock` by the monitor aspect potentially conflicts with the application aspect.

4 Selected Related Work

Monitor and stack are fundamental concepts and consequently staples for numerous articles in many Computer Science fields.

[Hedin97] considered the monitor from an Aspect Oriented perspective, and introduced an attribute extension language to enable a preprocessing stage to validate that the requisite coding constraints had been observed. In this paper we use reflection to synthesise the required code directly.

[Bjarnason97] advocates an extensible language, so that the required monitor protocol can be incorporated into the extended language. Language extension involves manipulation of syntax trees, and it is not clear how practical this is for a language with as challenging a syntax as C++.

[Chiba95] describes Open C++, an extension of C++ with a Meta Operation Protocol, operating somewhat in the CLOS tradition [Kiczales91] by intercepting compilation activities to enable user supplied code to update the syntax tree description of the program. This is a very powerful approach but rather alien to normal C++ programming. FOG shares the compile-time flexibility, zero run-time cost attributes of OpenC++. However the meta-level provided by FOG is much closer to C++, operating by merging relatively conventional declarations, rather than constructing syntax tree nodes. The FOG meta-level renders Cpp redundant.

[Lopes98] describes AspectJ, a weaver for Java. FOG provides a meta-level for C++ and weaves as a result of syntax generalisation.

5 Conclusion

The example shows how elimination of the One Definition Rule permits a useful interpretation to be applied to multiple declarations so that a preprocessing or meta-compilation phase can weave the declarations corresponding to independent aspects to produce a conventional C++ program.

6 Acknowledgements

The first author is grateful to Racal Research Ltd. for providing the time and resources to pursue this research. Both authors thank Brian Hillam, Mike Quinlan and Lois Sewell for helpful comments on earlier drafts.

References

[Aksit96] Mehmet Aksit. *Composition and separation of concerns in the Object-Oriented model.* ACM Computing Surveys, vol. 28A, no. 4es, 148, December 1996.
http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/al48-aksit/al48-aksit.html

[Bjarnason97] Elizabeth Bjarnason. *Tool Support for Framework-specific Language Extensions.* In *Proceedings of the Language Support for Design Patterns and Frameworks Workshop at ECOOP'97*. In *Workshop Reader of the European Conference on Object-Oriented Programming(ECOOP),* LNCS 1357, Springer-Verlag, June 1997.
http://bilbo.ide.hk-r.se:8080/~bosch/lsdforg/bjarnason.ps

[C++] ANSI X3J16/96-0225. *Working Paper for Draft Proposed International Standard for Information Systems - Programming Language - C++*. American National Standards Institute, 1996 (also known as Committee Draft 2).

[Chiba95] Shigeru Chiba. *A Metaobject Protocol for C++*. Proceedings of the 1995 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'95, ACM SIGPLAN Notices, vol. 30, no. 10, 285-299, October 1995.
http://www.softlab.is.tsukuba.ac.jp/~chiba/pub/chiba-oopsla95.ps.gz

[Hedin97] Görel Hedin. *Attribute Extension - A Technique for Enforcing Programming Conventions.* Nordic Journal of Computing, vol. 4, no 1, 93-122, 1997.

[Hoare74] C.A.R. Hoare. *Monitors: An Operating System Structuring Concept*. Communications of the ACM, vol. 17, no 10, 549-557, October 1974.

[Kiczales91] G. Kiczales, J. des Rivières and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Lopes98] Cristina Videira Lopes and Gregor Kiczales. *Recent developments in AspectJ.* In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*. In *Workshop Reader of the European Conference on Object-Oriented Programming(ECOOP),* LNCS?, Springer-Verlag, June 1998.
http://wwwtrese.cs.utwente.nl/aop-ecoop98/papers/Lopes.pdf

[Ossher98] Harold Ossher and Peri Tarr. *Operation-Level Composition: A Case in (Join) Point*. In Cristina Lopes, Gail Murphy and Gregor Kiczales. *Proceedings of the Aspect-Oriented Programming Workshop at ICSE'98. 98. April 1998.* And in *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*, in *Workshop Reader of the European Conference on Object-Oriented Programming(ECOOP),* LNCS?, Springer-Verlag, June 1998.
http://www.parc.xerox.com/spl/projects/aop/icse98/aop-icse98-proceedings.pdf