The Static Reflection Pattern

Bob Jolliffe
University of Pretoria
bobj@cs.up.ac.za

August 2001

Abstract

Somewhere near the bottom of the food chain of object oriented programming, the developer frequently encounters the rock face of a non object oriented API. This paper describes a specialisation of the Wrapper Facade [Sch99] [SSRB00] pattern. Wrapper Facades encapsulate functions and data provided by existing non-object oriented API's. The Static Reflector addresses the particular problem of building wrappers which contain functions which take C function pointers as parameters. The pattern makes use of a static reflection method to facilitate the construction of cohesive, reusable framework classes which make use of such C functions. I show that the application of this pattern is surprisingly wide. Though concerned primarily with the interface between C and C++, the pattern has implications and applications to other languages as diverse as Java and [incr Tcl].

1 Problem

Many non-object oriented API's contain functions which arrange for another function (the target function) to be dispatched, perhaps in the context of a new thread or in the future, in response to an I/O, timer or user interface event. Building object oriented components on top of such API's is complicated by the fact that the target function must be statically declared. There is generally no way to directly specify a member function of an object instance to be the target of such an API function.

As a motivating problem, consider the problem of implementing a Java Thread class in a Java Virtual Machine written in C++ using the POSIX threads API. Java threads have a `start()` method which causes a new thread to be spawned to run, with its `run()` hook method as the thread entry point. In Java we would create and dispatch the thread like this:

```
Thread t = new Thread();

t.start();
```

A starting point might be to collect together the POSIX threads functions (`pthread_create()` and family) into a cohesive Wrapper Facade[SSRB00]. It would then be convenient if we could build a C++ implementation as follows:

```
class Thread {
public:
   Thread() {}
   int  start()
     {
       // incorrect - run is not static
       return pthread_create(&tid, NULL,
                          run, NULL);
     }
protected:
   virtual void* run()
     {/* thread function */}

   pthread_t tid; //thread id
   /* ... other member data for
      the thread object */
};
```

The resulting `run()` method of Thread instances would have access to the member data of the instances. We could create new thread classes by inheriting from the base Thread and simply providing an overloaded `run()` method. Unfortunately, the call to `pthread_create()` is illegal as the third parameter refers to our `run()` method, which is not a valid static C function.

The above code can be modified to compile correctly by simply declaring the `run()` method as `static`, but this has a serious drawback. Static class functions have no direct access to the instance data of the object, nor can they benefit from inheritance and polymorphism.

2 Context

This problem recurs frequently in the context of building C++ classes around C functions which take C-style function pointers as parameters. Such functions are usually scheduling functions of some sort i.e. they request that another function be dispatched after some event occurs or in the context of a new thread.

Examples include:

The POSIX threads library The function `pthread_create()`[1], which is used for creating new threads, has an argument which specifies the entry function for the new thread. The function prototype is as follows:

[1] The Windows `beginthreadex()` function has a similar form and is also a candidate for static reflection

```
int pthread_create(pthread_t *thread,
       pthread_attr_t* attr,
       void * (*start_routine)(void *),
       void * arg);
```

The start_routine argument specifies the entry function.

The Win32 API This interface is rich in its use of function callbacks. The `SetWaitableTimer()` function, for example, has an optional argument to cause an Asynchronous Procedure Call (APC) to be queued when the timer expires. The argument specifies a pointer to a C function. An application places itself in an alertable state, such as in a call to `SleepEx()`, to receive notification of the event and dequeue and dispatch the APC.

The Tcl C library This library provides a number of useful functions for creating event-driven applications based around the Tcl Notifier[Ous94]. A commonly used one is `Tcl_CreateFileHandler()` which has the form:

```
Tcl_CreateFileHandler(int fd, int mask,
            Tcl_FileProc proc,
            ClientData clientdata)
```

The proc argument is a pointer to a C function. Typically, the application waits for events in an infinite loop, blocking in calls to `Tcl_DoOneEvent()`.

Each of the above is characterized by having at least two arguments; one being a pointer to a C function and the other a general purpose argument which is passed through to the target function (the *void* arg* in pthread_create, the *ClientData clientdata* in `Tcl_CreateFileHandler`). The purpose of this argument is to pass data to the target function.

Such functions are commonly found in system APIs as well as in legacy C libraries. These functions are important in the implementation of components for use in extensible object oriented frameworks. Template and Hook methods[Pre95] commonly form the metapatterns for such components, with the initiating method which makes the call to the C API scheduling function, being the template method. Hook methods are the application specific "hotspots" ie. the methods which are dispatched as a consequence of invoking the template method. They provide the application specific behavior. The problem-solution pair description below illustrates how the impedance mismatch between C and C++ frequently dictates the use of a third participant in this collaboration, the static reflector function.

3 Forces

- The developer needs to interact with a non object-oriented API for reasons of efficiency or fine grained functionality. The creation of Wrapper Facades i.e. clustering cohesive groups of functions into classes, is a proven good strategy[Sch99] for dealing with this interactiion. Functions which arrange the dispatch of other functions,

such as those described above, present an implementation problem in creating classes based on Wrapper Facades because the target functions, in each case, must fall outside of the Wrapper Facade (or any other) class. The `run()` method of a thread class, for example, should form part of the cohesive cluster of functions which operate on thread objects.

- An important benefit of building object-oriented infrastructure on top of a non object-oriented API is the encapsulation of data with the methods which operate on that data. It is therefore important for the dispatched function to have access to object instance data. In the thread case above, for example, the thread `run()` method should have access to the `tid` member data.

4 Solution

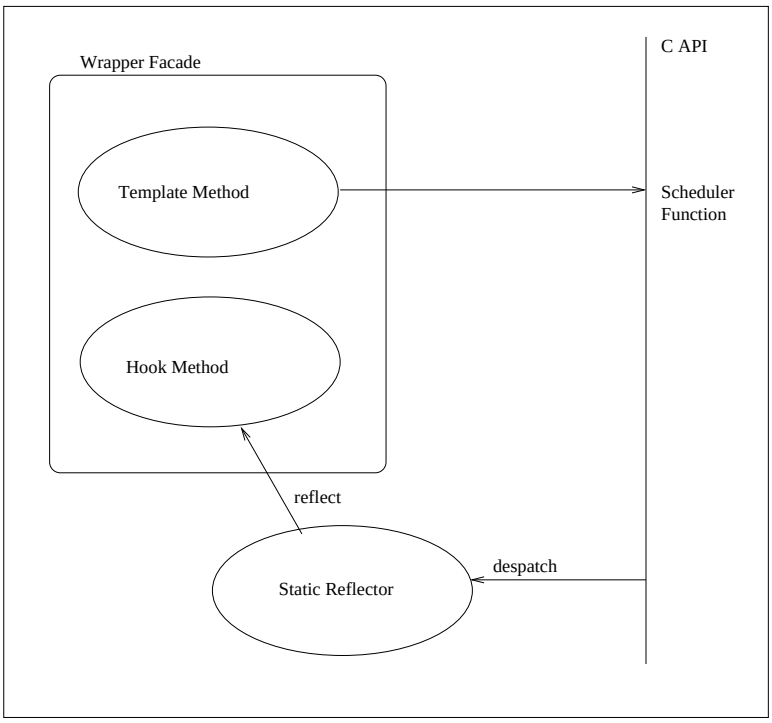

Figure 1: Collaborations

The Static Reflection pattern resolves these forces by providing a mechanism for causing the dispatch of an object member function. It does this by introducing a static method to the collaboration, which is the intermediate target of the scheduler function. It makes use of the generic void* type argument provided by the scheduler functions to send, not explicit

data to the target function, but a *reference* back to the originator of the scheduling call (a `this` pointer in C++). This collaboration is illustrated in Figure 1.

Applying this pattern to our Java Thread class implementation yields a solution which resolves the problems encountered earlier:

```
class Thread {
public:
   Thread() {}
   int  start()
     {
       return pthread_create(&tid, NULL,
                          reflect, this);
     }
protected:
   inline static void reflect(void* id)
     {
        (Thread*)(id)->run();
     }

   virtual void* run()
     {/* thread function */}

   pthread_t tid; //thread id
   /* ... other member data for object */
};
```

The thread `start()` method, as before, is just a thin wrapper for the `pthread_create()` API function. What is different now is that the entry point of the new thread is the static `reflect()` function. This function uses the `void*` parameter passed to it to find a way back to the thread instance `run()` method. The `run()` method is a non-static member function with access to the Thread instance data.

5 Resulting Context

An important consequence of the Static Reflection pattern is the ability to build framework objects through inheritance. By making the hook method virtual in the base class, derived classes need simply to provide an implementation of the hook method. The reflector in the base class will ensure that the hook is dynamically bound and despatched.

A desirable consequence of placing the static reflector within the namespace of the class is that scoping can be used to make the hook method protected. Placing the reflector outside of the class would require either the hook method to be declared public or the reflector to be a friend.

A possible negative consequence of this pattern is the extra overhead involved with the double dispatch. This overhead is minimized by inlining the reflector method. Combined with the effects of compiler optimization the overhead should be negligible.

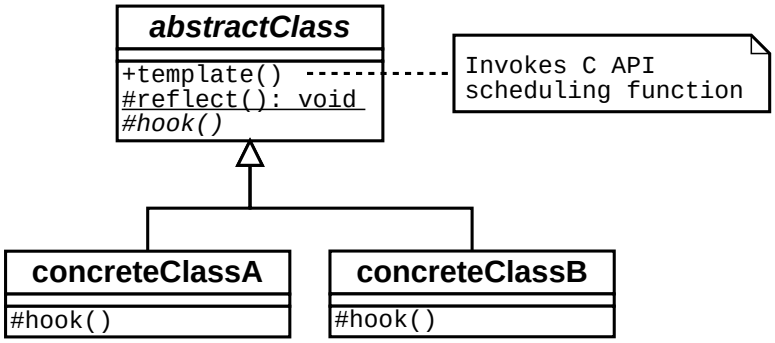

Figure 2: Inheritance

6 Rationale

The task of the static reflector method is to simply delegate to the hook method of the object which originated the message. The mechanism is similar to the double dispatch of the Visitor Pattern of [GHJV95][2] By providing a reflector method to reflect the message back to the originating object the pattern solution effectively resolve the forces in the given context:

- Cohesion is achieved, because we can include our target function among the other Wrapper Facade functions which act upon the object.
- Access to encapsulated data is achieved, because the target function is a non-static member function of the class.

7 Examples

The Java thread example in the previous section is an example of a synchronous application of the Static Reflector pattern. The pattern is also used in this way in [SSRB00] to implement a threaded TCP service handler. Another example from the POSIX threads API is in the installation of thread exit handlers with `pthread_cleanup_push()`.

The pattern is more commonly seen in the context of asynchronous, event-driven scenarios. In this section I describe two such cases: one using C++ and the Win32 API and the other using IncrTcl and the underlying Tcl Notifier. Both cases describe the implementation of timer handlers. I/O event handlers and GUI event handlers can be constructed using the same pattern, but the code for timers is shorter.

[2] In fact it is more like a triple dispatch, where the dispatching of the reflector is separated by space (a new thread context) or time (an event handler) from the originating call

7.1 Win32 APCs

The Win32 API provides a mechanism known as an Asynchronous Procedure Call (APC)[Mic00]. Threads have an APC queue upon which APCs are queued when they are due to be scheduled. Threads need to be in an alertable state for the APCs to be dequeued and dispatched. APCs are typically used for timer and I/O event handlers and provide an alternative mechanism to the `WaitForMultipleObjects()` family of functions for demultiplexing and dispatching. APCs are a more explicit event delivery mechanism. `WaitForMultipleObjects()`, like the Unix `select()` and `poll` functions[Ste98], lends itself more to a state-driven rather than event-driven design[GB99].

In the example below, an APC is used to implement a timer handler. The template method in this class is the `start()` method and the hook method, `timedout()` is a pure virtual method. Note how the static reflector method, `reflect()`, and the `timedout()` hook are both protected. A concrete timer class, `myTimer`, is implemented by providing an implementation of the `timedout()` hook.

```
#include <windows.h>
#include <iostream>
#include <string>

class Timer {
public:
  Timer()
    {
      thndl = CreateWaitableTimer(NULL, FALSE, NULL);
    }
  void start(int fire, int repeat)
    {
      // scale everything up to milliseconds
      liDueTime.QuadPart=-fire*10000;
      interval = repeat;
      // arrange for Win32 APC to reflector
      SetWaitableTimer(thndl, &liDueTime, interval, \
                       Timer::reflect, this, FALSE);
    }
protected:
  // the static reflector function
  static VOID CALLBACK
  reflect(LPVOID self, \
          DWORD dwTimerLowValue, \
          DWORD dwTimerHighValue)
    {
      Timer* id = (Timer*)self;
      id->TimedOut();
    }

  virtual void TimedOut() = 0;
  HANDLE thndl;
```

```
  LARGE_INTEGER liDueTime;
  int interval;
};

class myTimer : public Timer {
public:
  myTimer(const string& name = "Anonymous")
    :myname(name) {}
  // the callback - with access to member data!
  void TimedOut()
    {
      cout << myname << " timed out" << endl;
    }
protected:
  string myname;
};

int main()
{
  myTimer T1("A Win32 alertable timer"),T2;
  cerr << "Starting timers ...\n";
  T1.start(3000, 3000);
  T2.start(4000, 3000);
  // A primitive event loop ...
  while(1) {
    SleepEx(INFINITE,TRUE);
  }
}
```

7.2 [incr Tcl]

Incremental Tcl [incr Tcl] is an object system for the Tcl language created by Michael J. McLennan of Lucent Technologies[McL93]. Being an interpreted language, the mechanics are considerably less sophisticated than C++. [incr Tcl] supports classes, scoping and inheritance, but has no notion of polymorphism and virtual methods. The underlying event demultiplexing and dispatching mechanism is based on the C language Tcl Notifier, which necessitates the application of the Static Reflector pattern to build notifiable, event driven objects. The form is slightly different from the previous examples, but the pattern is the same.

```
class Timer {
    # Note: the after command causes
    # the reflex scriptlet to be
    # evaluated at global scope after
    # the elapsed ms.
    # reflex thus plays the role of
    # the static reflector
```

```
    method schedule {ms} {
        set reflex "$this hook"
        after $ms $reflex
    }

    method hook {} {
        puts "Timer expired"
    }
}

class myTimer {
    inherit Timer

    method hook {} {
        # reschedule for 2 sec later
        schedule 2000
        puts "myTimer expired!!"
    }
}

myTimer t1
t1 schedule 2000

# wait forever in event loop
# t1's hook will be despatched
#      after 2 seconds
vwait 1
```

It may not be immediately clear how static reflection is being used here. The key point is that the semantics of the tcl *after* command determines that the argument script to *after* is evaluated at global scope. Notice how the template method (schedule) creates a string variable (reflex) which acts as the reflector to call back the hook method. In this case the hook method must be public because reflex is evaluated outside of the class namespace. We could have made the reflex script call back to a class wide procedure within Timer, which in turn called hook. This way the hook method could be declared protected, but at some cost.

There are many non object oriented APIs to which this this pattern can be meaningfully applied. One other such API the author is aware of is the Gtk toolkit, which is a C GUI framework used in the Gnome project[3]. Functions such as `gtk_signal_connect()` bind a C style function to a user-interface event. Static reflection is required to route such event handlers to object methods.

[3]More information about gnome and gtk can be found at http://www.gnome.org/

8 Exceptions and Variations

Not all scheduling type functions are candidates for static reflection. One notable exception is the installation and dispatch of signal handlers. The BSD `signal()` function and it's POSIX counterpart, `sigaction()`, specify a function to be dispatched in response to an operating system signal. Neither API function provides the facility for passing a *this* pointer, so static reflection cannot be used. [Sch97] demonstrates how design patterns can be applied to the development of signal handling components.

The OpenGL GLUT library supports C-style callback functions for GUI events. These callbacks do not have the facility for passing a *this* pointer, so static reflection cannot be used. The author is aware of object oriented interfaces to OpenGL but not how they are implemented.

Variations on the static reflector pattern are commonly seen when the collaboration between template, reflector and hook methods transgress class boundaries. I have shown examples where all three are defined in a single class. There are cases where it may be desirable to more clearly separate the functionality of these three.

Creating pools of managed threads, for example, may suggest a design strategy where the template and reflector methods occur in a thread factory class, and the thread entry hook in a separate thread class. Similarly, one can separate an event handler class from the event dispatch and demultiplexing mechanism, as is done in the Reactor[SO95] pattern. The essence of the template-reflect-hook collaboration remains the same in each case.

9 Related Patterns

This pattern is closely related to the Wrapper Facade[Sch99] pattern, which addresses the problem of building object oriented infrastructure on top of non object-oriented APIs. Whereas the Wrapper Facade deals with cohesive grouping of related existing API functions, the Static Reflector provides a mechanism for extending Wrapper Facades to include scheduled functions such as thread entry points and event callbacks.

In its application to the context of event callbacks, there is also some relationship with the Reactor pattern. The TkReactor implementation of the Reactor in the ACE toolkit makes use of static reflectors to dispatch timer and I/O handlers.

Hook Method, Template Method, Double Dispatch[GHJV95].

10 Acknowledgments

Karen Renaud for proof-reading the first draft. Doug Shmidt for suggesting the Java thread example. Dorin Sandu for a marvellous sheperding job.

References

[GB99] Peter Druschel Gaurav Banga, Jeffrey C. Mogul. A scalable and explicit event delivery mechanism for unix. In *USENIX Annual Technical Conference*, June 1999.

[GHJV95] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[McL93] M. McLennan. [incr Tcl]: Object-Oriented Programming with Tcl, 1993.

[Mic00] Microsoft Corporation. *Microsoft Developer Network (MSDN) Library*, April 2000.

[Ous94] John K. Ousterhoudt. *Tcl and The Tk Toolkit*. Addison-Wesley, 1994.

[Pre95] Wolfgang Pree. *Design Patterns for Object Oriented Software Development*. ACM Press, Addison Wesley, 1995.

[Sch97] Douglas C. Schmidt. "Applying Design Patterns to Simplify Signal Handling". *C++ Report, SIGS*, 9(6), June 1997.

[Sch99] Douglas C. Schmidt. Wrapper facade: A structural pattern for encapsulating functions within classes. *C++ Report*, 1999.

[SO95] Douglas C. Schmidt and James O'Coplan, editors. *Pattern Languages of Program Design I*. Addison Wesley, 1995.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. Wiley, 2000.

[Ste98] W. Richard Stevens. *Unix Network Programming*, volume 1. Prentice Hall, 2nd edition, 1998.