Wrapper Facade

A Structural Pattern for Encapsulating Functions within Classes

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130, (314) 935-7538

This paper appeared in the C++ Report magazine, February, 1999.

1 Introduction

This paper describes the Wrapper Facade pattern. The intent of this pattern is to encapsulate low-level functions and data structures with object-oriented (OO) class interfaces. Common examples of the Wrapper Facade pattern are class libraries like MFC, ACE, and AWT that encapsulate native OS C APIs, such as sockets, pthreads, or GUI functions.

Programming directly to native OS C APIs makes networking applications verbose, non-robust, non-portable, and hard to maintain because it exposes many low-level, error-prone details to application developers. This paper illustrates how the Wrapper Facade pattern can help to make these types of applications more concise, robust, portable, and maintainable.

This paper is organized as follows: Section 2 describes the Wrapper Facade pattern in detail using the Siemens format [1] and Section 3 presents concluding remarks.

2 Wrapper Facade Pattern

2.1 Intent

Encapsulate low-level functions and data structures within more concise, robust, portable, and maintainable higher-level object-oriented class interfaces.

2.2 Example

To illustrate the Wrapper Facade pattern, consider the server for a distributed logging service shown in Figure 1. Client applications use the logging service to record information about their execution status in a distributed environment. This status information commonly includes error notifications, debugging traces, and performance diagnostics. Logging records are sent to a central logging server, which writes the records to various output devices, such as a network management console, a printer, or a database.

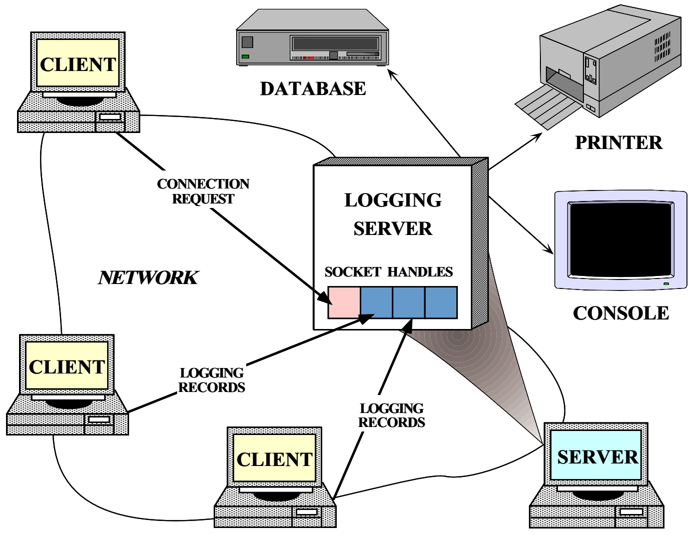


Figure 1: Distributed Logging Service

The logging server shown in Figure 1 handles connection requests and logging records sent by clients. Logging records and connection requests can arrive concurrently on multiple socket *handles*. Each handle identifies network communication resources managed within an OS.

Clients communicate with the logging server using a connection-oriented protocol like TCP [2]. Thus, when a client wants to log data, it must first send a connection request to the logging server. The server accepts connection requests using a *handle factory*, which listens on a network address known to clients. When a connection request arrives, the OS handle factory accepts the client's connection and creates a socket handle that represents this client's connection endpoint. This handle is returned to the logging server, which waits for client logging requests to arrive on this and other socket handles. Once clients are connected, they can send logging records to the server. The server receives these records via the connected socket handles, processes the records, and writes them to their output device(s).

A common way to develop a logging server that processes multiple clients concurrently is to use low-level C language functions and data structures for threading, synchronization and network communication. For instance, Figure 2 illustrates the use of Solaris threads [3] and the socket [4] network programming API to develop a multi-threaded logging server.

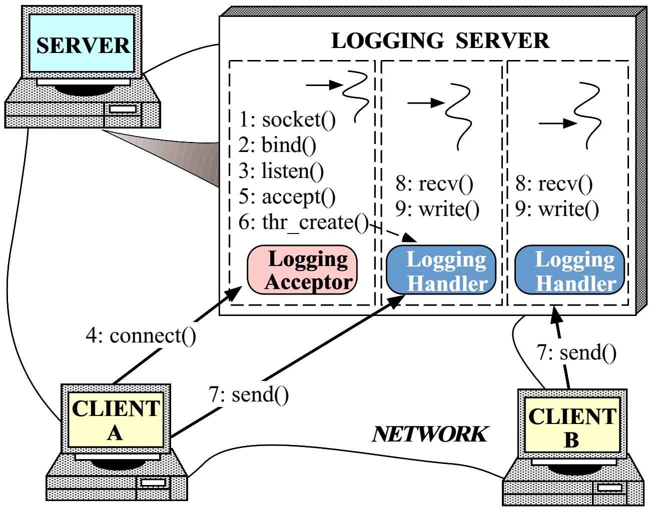


Figure 2: Multi-threaded Logging Server

In this design, the logging server's handle factory accepts client network connections in its main thread. It then spawns a new thread that runs a `logging_handler` function to process logging records from each client in a separate connection. The following two C functions illustrates how to implement this logging server design using the native Solaris OS APIs for sockets, mutexes, and threads.[1]

```
// At file scope.

// Keep track of number of logging requests.
static int request_count;

// Lock to protect request_count.
static mutex_t lock;

// Forward declaration.
static void *logging_handler (void *);

// Port number to listen on for requests.
static const int logging_port = 10000;

// Main driver function for the multi-threaded
// logging server.  Some error handling has been
// omitted to save space in the example.

int
main (int argc, char *argv[])
{
  struct sockaddr_in sock_addr;

  // Handle UNIX/Win32 portability differences.
#if defined (_WINSOCKAPI_)
  SOCKET acceptor;
#else
  int acceptor;
#endif /* _WINSOCKAPI_ */

  // Create a local endpoint of communication.

  acceptor = socket (PF_INET, SOCK_STREAM, 0);
```

[1]Readers who are not interested in the complete code details of this example can skip to the pattern's Context in Section 2.3.

```
  // Set up the address to become a server.
  memset (reinterpret_cast <void *> (&sock_addr),
          0, sizeof sock_addr);
  sock_addr.sin_family = AF_INET;
  sock_addr.sin_port = htons (logging_port);
  sock_addr.sin_addr.s_addr = htonl (INADDR_ANY);

  // Associate address with endpoint.
  bind (acceptor,
        reinterpret_cast <struct sockaddr *>
          (&sock_addr),
        sizeof sock_addr);

  // Make endpoint listen for connections.
  listen (acceptor, 5);

  // Main server event loop.
  for (;;) {
    thread_t t_id;

    // Handle UNIX/Win32 portability differences.
#if defined (_WINSOCKAPI_)
    SOCKET h;
#else
    int h;
#endif /* _WINSOCKAPI_ */

    // Block waiting for clients to connect.
    int h = accept (acceptor, 0, 0);

    // Spawn a new thread that runs the
    // <logging_handler> entry point and
    // processes client logging records on
    // socket handle <h>.
    thr_create (0, 0,
                logging_handler,
                reinterpret_cast <void *> (h),
                THR_DETACHED,
                &t_id);
  }

  /* NOTREACHED */
  return 0;
}
```

The `logging_handler` function runs in a separate thread of control, *i.e.*, one thread per connected client. It receives and processes logging records on each connection, as follows:

```
// Entry point that processes logging records for
// one client connection.
void *logging_handler (void *arg)
{
  // Handle UNIX/Win32 portability differences.
#if defined (_WINSOCKAPI_)
  SOCKET h = reinterpret_cast <SOCKET> (arg);
#else
  int h = reinterpret_cast <int> (arg);
#endif /* _WINSOCKAPI_ */

  for (;;) {
    UINT_32 len; // Ensure a 32-bit quantity.
    char log_record[LOG_RECORD_MAX];

    // The first <recv> reads the length
    // (stored as a 32-bit integer) of
    // adjacent logging record.  This code
    // does not handle "short-<recv>s".
    ssize_t n = recv
      (h,
       reinterpret_cast <char *> (&len),
       sizeof len, 0);

    // Bail out if we don't get the expected len.
    if (n <= sizeof len) break;
    len = ntohl (len); // Convert byte-ordering.
    if (len > LOG_RECORD_MAX) break;

    // The second <recv> then reads <len>
    // bytes to obtain the actual record.
    // This code handles "short-<recv>s".
    for (size_t nread = 0;
         nread < len;
         nread += n) {
      n = recv (h,
                log_record + nread,
                len - nread, 0);
      // Bail out if an error occurs.
      if (n <= 0)
        return 0;
    }

    mutex_lock (&lock);

    // Execute following two statements in a
    // critical section to avoid race conditions
    // and scrambled output, respectively.
    ++request_count; // Count # of requests received.

    if (write (1, log_record, len) == -1)
      // Unexpected error occurred, bail out.
      break;

    mutex_unlock (&lock);
  }

  close (h);
  return 0;
}
```

Note how all the threading, synchronization, and networking code is programmed using the low-level C functions and data structures provided by the Solaris operating system.

2.3 Context

Applications that access services provided by low-level functions and data structures.

2.4 Problem

Networking applications are often written using the low-level functions and data structures illustrated in Section 2.2. Although this is common practice, it causes problems for application developers by failing to resolve the following forces:

Verbose, non-robust programs: Application developers who program directly to low-level functions and data structures must repeatedly rewrite a great deal of tedious software logic. In general, code that is tedious to write and maintain often contains subtle and pernicious errors.

For instance, the code for creating and initializing an acceptor socket in the `main` function in Section 2.2 is prone to errors, such as failing to zero-out the `sock_addr` or not using `htons` on the `logging_port` number [5]. The `mutex_lock` and `mutex_unlock` are also easy to misuse. For example, if the `write` call returns -1 the `logging_handler` code breaks out of the loop without releasing the mutex lock. Similarly, the socket handle `h` will not be closed if the nested `for` loop returns when it encounters an error.

Lack of portability: Software written using low-level functions and data structures is often non-portable between different OS platforms and compilers. Moreover, it's often not even portable to program to low-level functions across different versions of the same OS or compiler. Non-portability stems from the lack of information hiding in low-level APIs based on functions and data structures.

For instance, the logging server implementation in Section 2.2 has hard-coded dependencies on several non-portable native OS threading and network programming C APIs. In particular, the use of `thr_create`, `mutex_lock`, and `mutex_unlock` is not portable to non-Solaris OS platforms. Likewise, certain socket features, such as the use of `int` to represent a socket handle, are not portable to non-UNIX platforms like WinSock on Win32, which represent a socket handle as a pointer.

High maintenance effort: C and C++ developers typically achieve portability by explicitly adding conditional compilation directives into their application source code using `#ifdef`s. However, using conditional compilation to address platform-specific variations *at all points of use* increases the *physical design* complexity [6] of application source code. It is hard to maintain and extend such software since platform-specific implementation details are scattered throughout the application source files.

For instance, the `#ifdef`s that handle Win32 and UNIX portability with respect to the data type of a socket, *i.e.*, `SOCKET` vs. `int`, impedes the readability of the code. Developers who program to low-level C APIs like these must have intimate knowledge of many OS platform idiosyncrasies to write and maintain this code.

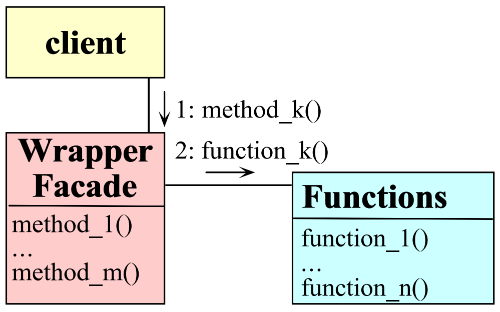
As a result of these drawbacks, developing applications by programming directly to low-level functions and data structures is rarely an effective design choice for application software.

2.5 Solution

An effective way to ensure applications avoid accessing low-level functions and data structures directly is to use the *Wrapper Facade* pattern. For each set of related functions and data structures, create one or more wrapper facade classes that encapsulate low-level functions and data structures within more concise, robust, portable, and maintainable methods provided by the wrapper facade interface.

2.6 Structure

The structure of the participants of the Wrapper Facade pattern is illustrated in the following UML class diagram:



The key participants in the Wrapper Facade pattern include the following:

Functions:

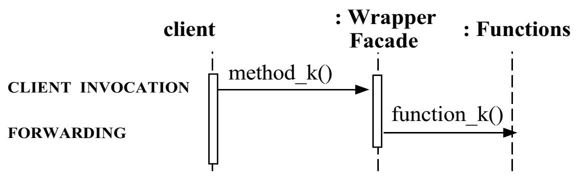
- The *Functions* are existing low-level functions and data structures that provide a cohesive service.

Wrapper Facade:

- The *Wrapper Facade* is a set of one or more classes that encapsulate the Functions and their associated data structures. The Wrapper Facade provides methods that forward client invocations to one or more of the low-level Functions.

2.7 Dynamics

The following figure illustrates the collaborations in the Wrapper Facade pattern:



These collaborations are straightforward, as described below:

1. Client invocation: The client invokes a method via an instance of the Wrapper Facade.

2. Forwarding: The Wrapper Facade method forwards the request to one or more of the underlying Functions that it encapsulates, passing along any internal data structures needed by the function(s).

2.8 Implementation

This section explains the steps involved in implementing components and applications with the Wrapper Facade pattern. We illustrate how these Wrapper Facades overcome the problems with verbose, non-robust programs, lack of portability, and high maintenance effort plaguing the solution that used low-level functions and data structures.

Figure 3 illustrates the structure and participants in this example, which is based on the logging server described in Section 2.2 The examples in this section apply the reusable components from the ACE framework [7]. ACE provides a rich set of reusable C++ wrappers and framework components that perform common communication software tasks across a wide range of OS platforms.

The following steps can be taken to implement the Wrapper Facade pattern:

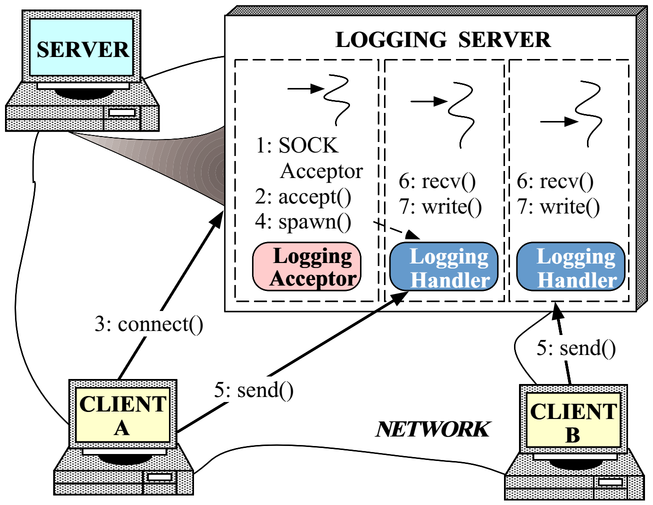


Figure 3: Multi-threaded Logging Server

1. Identify the cohesive abstractions and relationships among existing functions: Conventional APIs like Win32, POSIX, or X Windows that are implemented as individual functions and data structures provide many cohesive abstractions, such as mechanisms for network programming, synchronization and threading, and GUI management. Due to the lack of data abstraction support in low-level languages like C, however, it is often not immediately obvious how these existing functions and data structures are related to each other. Therefore, the first step in applying the Wrapper Facade pattern is to identify the cohesive abstractions and relationships among the lower level functions in an existing API. In other words, we define an "object model" by clustering the existing low-level API functions and data structures into one or more classes.

In our logging example, we start by carefully examining our original logging server implementation. This implementation used many low-level functions that actually provide several cohesive services, such as synchronization and network communication. For instance, the `mutex_lock` and `mutex_unlock` functions are associated with a mutex synchronization abstraction. Likewise, the `socket`, `bind`, `listen`, and `accept` functions play various roles as a network programming abstraction.

2. Cluster cohesive groups of functions into Wrapper Facade classes and methods: This step can be broken down into the following substeps:

In this step, we define one or more wrapper facade classes for each group of functions and data structures that are related to a particular abstraction.

A. Create cohesive classes: We start by define one or more wrapper facade classes for each group of functions and data structures that are related to a particular abstraction. Several common criteria used to create cohesive classes include the following:

- Coalesce functions with high *cohesion* into individual classes, while minimizing unnecessary *coupling* between classes.
- Determine what is *common* and what is *variable* in the underlying functions and to group functions into classes that isolate the variation behind a uniform interface.

In general, if the original API contains a wide range of related functions it may be necessary to create several wrapper facade classes in order to properly separate concerns.

B. Coalesce multiple individual functions into class methods: In addition to grouping existing into classes, it is also often beneficial to combine multiple individual functions into a smaller number of methods in each wrapper facade class. For instance, this design may be necessary to ensure that a group of low-level functions are called in the appropriate order.

C. Select the level of indirection: Most wrapper facade classes simply forward their method calls directly to the underlying low-level functions. If the wrapper facade methods are inlined there may be no additional indirection compared to invoking the low-level functions directly. To enhance extensibility, it is also possible to add another level of indirection by dynamically dispatching the wrapper facade method implementations. In this case, the wrapper facade classes play the role of the *Abstraction* in the Bridge pattern [8].

D. Determine where to handle platform-specific variation: Minimizing platform-specific application code is an important benefit of using the Wrapper Facade pattern. Thus, although wrapper facade class method *implementations* may differ across different OS platforms they should provide uniform, platform-independent *interfaces*.

One strategy for handling platform-specific variation is to `#ifdef`s the wrapper facade class method implementations. When `#ifdef`s are used in conjunction with auto-configuration tools, such as GNU `autoconf`, uniform, platform-independent wrapper facades can be created with a single source tree. An alternative strategy is to factor out different wrapper facade class implementations into separate directories, *e.g.*, one per platform, and configure the language processing tools to include the appropriate wrapper facade class into applications at compile-time.

Choosing a particular strategy depends largely on how frequently the wrapper facade method implementations change. For instance, if they change frequently, it can be tedious to update the `#ifdef`s correctly for each platform. Likewise, all files that depend on this file may need to be recompiled, even if the change is only necessary for one platform.

In our logging example, we'll define wrapper facade classes for mutexes, sockets, and threads in order to illustrate how each of these substeps can be addressed, as follows:

- **The mutex wrapper facade:** We first define a `Thread_Mutex` abstraction that encapsulates the Solaris mutex functions in a uniform and portable class interface:

```
class Thread_Mutex
{
public:
  Thread_Mutex (void) {
    mutex_init (&mutex_, 0, 0);
  }
  ~Thread_Mutex (void) {
    mutex_destroy (&mutex_);
  }
  int acquire (void) {
    return mutex_lock (&mutex_);
  }
  int release (void) {
    return mutex_unlock (&mutex_);
  }

private:
  // Solaris-specific Mutex mechanism.
  mutex_t mutex_;

  // = Disallow copying and assignment.
  Thread_Mutex (const Thread_Mutex &);
  void operator= (const Thread_Mutex &);
};
```

By defining a `Thread_Mutex` class interface, and then writing applications to use it rather than the low-level native OS C APIs, we can easily port our wrapper facade to other platforms. For instance, the following `Thread_Mutex` implementation works on Win32:

```
class Thread_Mutex
{
public:
  Thread_Mutex (void) {
    InitializeCriticalSection (&mutex_);
  }

  ~Thread_Mutex (void) {
    DeleteCriticalSection (&mutex_);
  }

  int acquire (void) {
    EnterCriticalSection (&mutex_); return 0;
  }

  int release (void) {
    LeaveCriticalSection (&mutex_); return 0;
  }

private:
  // Win32-specific Mutex mechanism.
  CRITICAL_SECTION mutex_;

  // = Disallow copying and assignment.
  Thread_Mutex (const Thread_Mutex &);
  void operator= (const Thread_Mutex &);
};
```

As described earlier, we can simultaneously support multiple OS platforms by using `#ifdef`s in the `Thread_Mutex` method implementations along with auto-configuration tools, such as GNU `autoconf`, to provide a uniform, platform-independent mutex abstraction using a single source tree. Conversely, we could also factor out the different `Thread_Mutex` implementations into separate directories and instruct our language processing tools to include the appropriate version into our application at compile-time.

In addition to improving portability, our `Thread_Mutex` wrapper facade provides a mutex interface that is less error-prone than programming directly to the low-level Solaris functions and `mutex_t` data structure. For instance, we can use the C++ `private` access control specifier to disallow copying and assignment of mutexes, which is an erroneous use-case that is not prevented by the less strongly-typed C programming API.

• **The socket wrapper facades:** The socket API is much larger and more expressive than the Solaris mutex API [5]. Therefore, we must define a group of related wrapper facade classes to encapsulate sockets. We'll start by defining the following typedef that handles UNIX/Win32 portability differences:

```
#if !defined (_WINSOCKAPI_)
typedef int SOCKET;
#define INVALID_HANDLE_VALUE -1
#endif /* _WINSOCKAPI_ */
```

Next, we'll define an `INET_Addr` class that encapsulates the Internet domain address `struct`:

```
class INET_Addr
{
public:
  INET_Addr (u_short port, long addr) {
    // Set up the address to become a server.
    memset (reinterpret_cast <void *> (&addr_),
            0, sizeof addr_);
    addr_.sin_family = AF_INET;
    addr_.sin_port = htons (port);
    addr_.sin_addr.s_addr = htonl (addr);
  }

  u_short get_port (void) const {
    return addr_.sin_port;
  }

  long get_ip_addr (void) const {
    return addr_.sin_addr.s_addr;
  }

  sockaddr *addr (void) const {
    return reinterpret_cast <sockaddr *>
      (&addr_);
  }

  size_t size (void) const {
    return sizeof (addr_);
  }
  // ...

private:
  sockaddr_in addr_;
};
```

Note how the `INET_Addr` constructor eliminates several common socket programming errors by zeroing-out the `sockaddr_in` field and ensuring the port and IP address are converted into network byte order.

The next wrapper facade class, `SOCK_Stream`, encapsulates the I/O operations, such as `recv` and `send`, that an application can invoke on a connected socket handle:

```
class SOCK_Stream
{
public:
  // = Constructors.

  // Default constructor.
  SOCK_Stream (void)
   : handle_ (INVALID_HANDLE_VALUE) {}

  // Initialize from an existing HANDLE.
  SOCK_Stream (SOCKET h): handle_ (h) {}

  // Automatically close the handle on destruction.
  ~SOCK_Stream (void) { close (handle_); }

  void set_handle (SOCKET h) { handle_ = h; }
  SOCKET get_handle (void) const { return handle_; }

  // = I/O operations.
  int recv (char *buf, size_t len, int flags = 0);
  int send (const char *buf, size_t len,
            int flags = 0);
  // ...

private:
  // Handle for exchanging socket data.
  SOCKET handle_;
};
```

Note how this class ensures that a socket handle is automatically closed when a `SOCK_Stream` object goes out of scope.

`SOCK_Stream` objects are created by a connection factory, `SOCK_Acceptor`, which encapsulates *passive* connection establishment logic [9]. The `SOCK_Acceptor` constructor initializes the passive-mode acceptor socket to listen at the `sock_addr` address. Likewise, the `accept` factory method initializes the `SOCK_Stream` with the newly accepted connection, as follows:

```
class SOCK_Acceptor
{
public:
  SOCK_Acceptor (const INET_Addr &sock_addr) {
    // Create a local endpoint of communication.
    handle_ = socket (PF_INET, SOCK_STREAM, 0);

    // Associate address with endpoint.
    bind (handle_,
          sock_addr.addr (),
          sock_addr.size ());

    // Make endpoint listen for connections.
    listen (handle_, 5);
  };

  // Accept a connection and initialize
  // the <stream>.
  int accept (SOCK_Stream &stream) {
    stream.set_handle (accept (handle_, 0, 0));
    if (stream.get_handle ()
        == INVALID_HANDLE_VALUE)
      return -1;
    else return 0;
  }

private:
  // Socket handle factory.
  SOCKET handle_;
};
```

Note how the constructor for the `SOCK_Acceptor` ensures that the low-level `socket`, `bind`, and `listen` functions are always called in the right order.

A complete set of wrapper facades for sockets [5] would also include a `SOCK_Connector`, which encapsulates the *active* connection establishment logic [9].

• **The threading facade:** Many threading APIs are available on different OS platforms, including Solaris threads, POSIX Pthreads, and Win32 threads. These APIs exhibit subtle syntactic and semantic differences, *e.g.*, Solaris and POSIX threads can be spawned in "detached" mode, whereas Win32 threads cannot. It is possible, however, to provide a `Thread_Manager` wrapper facade that encapsulates these differences within a uniform API, as follows:

```
class Thread_Manager
{
public:
  int spawn (void *(*entry_point) (void *),
             void *arg,
             long flags,
             long stack_size = 0,
             void *stack_pointer = 0,
             thread_t *t_id = 0) {
    thread_t t;
    if (t_id == 0)
      t_id = &t;
    return thr_create (stack_size,
                       stack_pointer,
                       entry_point,
                       arg,
                       flags,
                       t_id);
  }

  // ...
};
```

The `Thread_Manager` can also provide methods for joining and canceling threads, as well.

3. Determine an error handling mechanism: Low-level C function APIs typically use return values and integer codes, such as `errno`, to communicate errors back to their callers. This technique is error-prone, however, since callers may neglect to check the return status of their function calls.

A more elegant way of reporting errors is to use exception handling. Many programming languages, such as C++ and Java, use exception handling as an error reporting mechanism. It is also used in some operating systems, such as Win32.

There are several benefits to using exception handling as the error handling mechanism for wrapper facade classes:

• **It is extensible:** Modern programming languages allow the extension of exception handling policies and mechanisms via features that have minimal intrusion on existing interfaces and usage. For instance, C++ and Java use inheritance to define hierarchies of exception classes.

• **It cleanly decouples error handling from normal processing:** For example, error handling information is not passed explicitly to an operation. Moreover, an application cannot accidentally ignore an exception by failing to check function return values.

• **It can be type-safe:** In a languages like C++ and Java exceptions are thrown and caught in a strongly-typed manner to enhance the organization and correctness of error handling code. In contrast to checking a thread-specific error value explicitly, the compiler ensures that the correct handler is executed for each type of exception.

However, there are several drawbacks to the use of exception handling for wrapper facade classes:

• **It is not universally available:** Not all languages provide exception handling. For instance, some C++ compilers do not implement exceptions. Likewise, when an OS provides exception handling services, they must be supported by language extensions, thereby reducing the portability of the code.

• **It complicates the use of multiple languages:** Since languages implement exceptions in different ways, or do not implement exceptions at all, it can be hard to integrate components written in different languages when they throw exceptions. In contrast, reporting error information using integer values or structures provides a more universal solution.

• **It complicates resource management:** Resource management can be complicated if there are multiple exit paths from a block of C++ or Java code [10]. Thus, if garbage collection is not supported by the language or programming environment, care must be taken to ensure that dynamically allocated objects are deleted when an exception is thrown.

• **It is potentially time and/or space inefficient:** Poor implements of exception handling incur time and/or space overhead even when exceptions are not thrown [10]. This overhead can be particularly problematic for embedded systems that must be efficient and have small memory footprints.

The drawbacks of exception handling are particularly problematic for wrapper facades that encapsulate kernel-level device drivers or low-level native OS APIs that must run portably on many platforms [5]. For these types of wrapper facades, a more portable, efficient, and thread-safe way to handle errors is to define an error handler abstraction that maintains information about the success or failure of operations explicitly. One widely used solution for these system-level wrapper facades is to use the Thread-Specific Storage pattern [11].

4. Define related helper classes (optional): Once the low-level functions and data structures are encapsulated within cohesive wrapper facade classes, it often becomes possible to create other helper classes that further simplify application development. The utility of these helper classes typically becomes apparent only after the Wrapper Facade pattern has been applied to cluster low-level functions and their associated data into classes.

In our logging example, for instance, we can leverage the following `Guard` class that implements the C++ *Scoped Locking* idiom, which ensures that a `Thread_Mutex` is properly released regardless of how the program's flow of control exits a scope:

```
template <class LOCK>
class Guard
{
public:
  Guard (LOCK &lock): lock_ (lock) {
    lock_.acquire ();
  }

  ~Guard (void) {
    lock_.release ();
  }

private:
  // Hold the lock by reference to avoid
  // the use of the copy constructor...
  LOCK &lock_;
```

The Guard class applies the C++ idiom described in [12] whereby "a constructor acquires resources and the destructor releases them" within a scope, as follows:

```
  // ...
  {
    // Constructor of <mon> automatically
    // acquires the <mutex> lock.
    Guard<Thread_Mutex> mon (mutex);

    // ... operations that must be serialized ...

    // Destructor of <mon> automatically
    // releases the <mutex> lock.
  }

  // ...
```

Since we use a *class* as the Thread_Mutex wrapper facade, we can easily substitute a different type of locking mechanism, while still reusing the Guard's automatic locking/unlocking protocol. For instance, we can replace the Thread_Mutex class with a Process_Mutex class, as follows:

```
  // Acquire a process-wide mutex.
  Guard<Process_Mutex> mon (mutex);
```

It's much harder to achieve this degree of "pluggability" if C functions and data structures are used instead of C++ classes.

2.9 Example Resolved

The code below illustrates the main function of the logging server after its been rewritten to use our wrapper facades for mutexes, sockets, and threads described in Section 2.8:

```
// At file scope.

// Keep track of number of logging requests.
static int request_count;

// Manage threads in this process.
static Thread_Manager thr_mgr;

// Lock to protect request_count.
static Thread_Mutex lock;

// Forward declaration.
static void *logging_handler (void *);

// Port number to listen on for requests.
static const int logging_port = 10000;

// Main driver function for the multi-threaded
// logging server.  Some error handling has been
// omitted to save space in the example.

int
main (int argc, char *argv[])
{
  // Internet address of server.
  INET_Addr addr (port);

  // Passive-mode acceptor object.
  SOCK_Acceptor server (addr);

  SOCK_Stream new_stream;

  // Wait for a connection from a client.

  for (;;) {
    // Accept a connection from a client.
    server.accept (new_stream);

    // Get the underlying handle.
    SOCKET h = new_stream.get_handle ();

    // Spawn off a thread-per-connection.
    thr_mgr.spawn (logging_handler,
                   reinterpret_cast <void *> (h),
                   THR_DETACHED);
  }
```

The logging_handler function runs in a separate thread of control, *i.e.*, one thread for each connected client. It receives and processes logging records on each connection, as follows:

```
// Entry point that processes logging records for
// one client connection.
void *logging_handler (void *arg)
{
  SOCKET h = reinterpret_cast <SOCKET> (arg);

  // Create a <SOCK_Stream> object from SOCKET <h>.
  SOCK_Stream stream (h);

  for (;;) {
    UINT_32 len; // Ensure a 32-bit quantity.
    char log_record[LOG_RECORD_MAX];

    // The first <recv_n> reads the length
    // (stored as a 32-bit integer) of
    // adjacent logging record.  This code
    // handles "short-<recv>s".
    ssize_t n = stream.recv_n
      (reinterpret_cast <char *> (&len),
       sizeof len);

    // Bail out if we're shutdown or
    // errors occur unexpectedly.
    if (n <= 0) break;
    len = ntohl (len); // Convert byte-ordering.
    if (len > LOG_RECORD_MAX) break;

    // The second <recv_n> then reads <len>
    // bytes to obtain the actual record.
    // This code handles "short-<recv>s".
    n = stream.recv_n (log_record, len);

    // Bail out if we're shutdown or
    // errors occur unexpectedly.
    if (n <= 0) break;

    {
      // Constructor of Guard automatically
      // acquires the lock.
      Guard<Thread_Mutex> mon (lock);

      // Execute following two statements in a
      // critical section to avoid race conditions
      // and scrambled output, respectively.
      ++request_count; // Count # of requests

      if (write (STDOUT, log_record, len) == -1)
         break;

      // Destructor of Guard automatically
      // releases the lock, regardless of
      // how we exit this block!
    }
  }

  // Destructor of <stream> automatically
  // closes down <h>.
  return 0;
}
```

Note how the code above fixes the various problems with the previous code shown in Section 2.2. For instance, the destructors of SOCK_Stream and Guard will close down the socket handle and release the Thread_Mutex, respectively, regardless of how the blocks of code are exited. Likewise, this code is much easier to port and maintain since it uses no platform-specific APIs.

2.10 Known Uses

The example in this paper focuses on concurrent network programming. However, the Wrapper Facade pattern has also been applied to many other domains, such as GUI frameworks and database class libraries. The following are some well known uses of the Wrapper Facade pattern:

Microsoft Foundation Classes (MFC): MFC provides a set of wrapper facades that encapsulate most of the low-level C Win32 APIs, focusing largely on providing GUI components that implement the Microsoft Document/Template architecture.

The ACE framework: The mutex, thread, and socket wrapper facades described in Section 2.8 are based on components in the ACE framework [7], such as the `ACE_Thread_Mutex`, `ACE_Thread_Manager`, and `ACE_SOCK*` classes, respectively.

Rogue Wave class libraries: Rogue Wave's `Net.h++` and `Threads.h++` class libraries implement wrapper facades for sockets, threads, and synchronization mechanisms on a number of OS platforms.

ObjectSpace System<Toolkit>: Wrapper facades for sockets, threads, and synchronization mechanisms are also provided by the ObjectSpace `System<Toolkit>`.

Java Virtual Machine and Java foundation class libraries: The Java Virtual Machine (JVM) and various Java foundation class libraries, such as AWT and Swing, provide a set of wrapper facades that encapsulate most of the low-level native OS system calls and GUI APIs.

2.11 Consequences

The Wrapper Facade pattern provides the following benefits:

More concise and robust programming interface: The Wrapper Facade pattern encapsulates many low-level functions within a more concise set of OO class methods. This reduces the tedium of developing applications using low-level functions and data structures, thereby reducing the potential for programming errors.

Improve application portability and maintainability: Wrapper Facade classes can be implemented to shield application developers from non-portable aspects of low-level functions and data structures. Moreover, the Wrapper Facade pattern improves software structure by replacing an application configuration strategy based on *physical design* entities, such as files and # ifdefs, with *logical design* entities, such as base classes, subclasses, and their relationships [6]. It is generally easier to understand and maintain applications in terms of their logical design rather than their physical design.

Improve modularity, reusability, and configurability of applications: The Wrapper Facade pattern creates reusable class components that can be "plugged" in and out of other components in a wholesale fashion using OO language features like inheritance and parameterized types. In contrast, it is much harder to replace groups of functions without resorting to coarse-grained OS utilities, such as linkers or file systems.

The Wrapper Facade pattern has the following liability:

Additional indirection: The Wrapper Facade pattern can incur additional indirection compared with using low-level functions and data structures directly. However, languages that support inlining, such as C++, can implement this pattern with no significant overhead since compilers can inline the method calls used to implement the wrapper facades.

2.12 See Also

The Wrapper Facade pattern is similar to the Facade pattern [8]. The intent of the Facade pattern is to simplify the interface for a subsystem. The intent of the Wrapper Facade pattern is more specific: it provides concise, robust, portable, and maintainable class interfaces that encapsulate low-level functions and data structures, such as the native OS mutex, socket, thread, and GUI C language APIs. In general, Facades hide complex class relationships behind a simpler API, whereas Wrapper Facades hide complex function and data structure relationships behind a richer class API.

The Wrapper Facade pattern can be implemented using the Bridge pattern [8] if dynamic dispatching is used to implement wrapper facade methods that play the role of the *Abstraction* in the Bridge pattern.

3 Concluding Remarks

This paper describes the Wrapper Facade pattern and gives a detailed example illustrating how to use it. Implementations of the ACE wrapper facade components described in this paper are freely available in the ACE [7] software distribution at URL `www.cs.wustl.edu/~schmidt/ACE.html`. This distribution contains complete C++ source code, documentation, and example test drivers developed at Washington University, St. Louis. ACE is currently being used in many communication software projects at companies like Bellcore, Boeing, DEC, Ericsson, Kodak, Lucent, Motorola, SAIC, and Siemens.

Acknowledgements

Thanks to Hans Rohnert, Regine Meunier, Michael Stal, Christa Schwanninger, Frank Buschmann, and Brad Appleton for extensive comments that greatly improved the form and content of the Wrapper Facade pattern description.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[2] W. R. Stevens, *UNIX Network Programming, First Edition*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[3] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[4] W. R. Stevens, *UNIX Network Programming, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.

[5] D. C. Schmidt, "IPC_SAP: An Object-Oriented Interface to Interprocess Communication Services," *C++ Report*, vol. 4, November/December 1992.

[6] J. Lakos, *Large-scale Software Development with C++*. Reading, MA: Addison-Wesley, 1995.

[7] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[9] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[10] H. Mueller, "Patterns for Handling Exception Handling Successfully," *C++ Report*, vol. 8, Jan. 1996.

[11] D. C. Schmidt, T. Harrison, and N. Pryce, "Thread-Specific Storage – An Object Behavioral Pattern for Accessing per-Thread State Efficiently," *C++ Report*, vol. 9, November/December 1997.

[12] Bjarne Stroustrup, *The C++ Programming Language,* 3^{rd} *Edition*. Addison-Wesley, 1998.