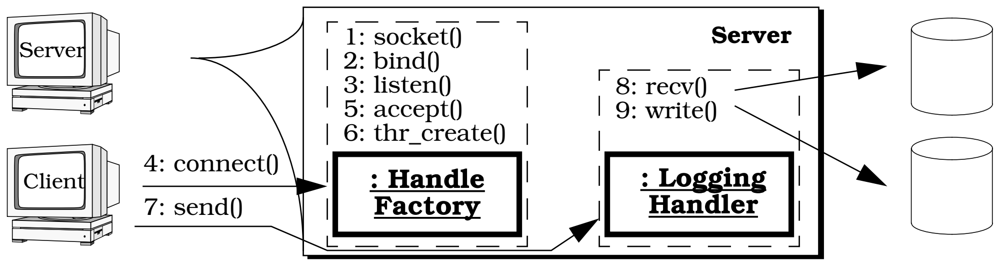
Wrapper Facade

The *Wrapper Facade* design pattern encapsulates the functions and data provided by existing lower-level, non-object-oriented APIs within more concise, robust, portable, maintainable, and cohesive higher-level object-oriented class interfaces.

Example Consider a server for a distributed logging service that handles multiple clients concurrently using the connection-oriented TCP protocol [Ste90]. To log data, a client must initiate a connection request to a designated server address. In the logging server, a *handle factory* listens on this address for connection requests to arrive. The handle factory 'accepts' the connection request and creates a socket handle that represents the client's connection endpoint. This handle is passed up to the server, which spawns a *logging handler* thread that waits for client logging requests. Once a client is connected, it sends logging requests to the server. The server receives these requests via the connected socket handle, processes the requests in the logging handler thread, and writes the requests to a log file.



A common way to develop the logging server is to use low-level C language functions and data for threading, synchronization and network communication, for example by using Solaris threads [EKBF+92] and the socket [Ste97] network programming API.

If the logging server runs on multiple platforms, however, there will be differences between low-level function/data APIs and compiler features and bugs. Developers commonly handle these differences by inserting conditional compilation directives, such as C/C++ `#ifdefs`, throughout the code. For instance, the following illustrates a logging server that has been implemented using `#ifdefs` to run on multiple platforms, such as Solaris *and* Windows NT:

```
#if defined (_WIN32)
   #include <windows.h>
   typedef int ssize_t;
#else
   typedef unsigned int UINT32;
   #include <thread.h>
   #include <unistd.h>
   #include <sys/socket.h>
   #include <netinet/in.h>
   #include <memory.h>
#endif /* _WIN32 */

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

// Lock that serializes concurrent access to request_count.
#if defined (_WIN32)
   static CRITICAL_SECTION lock;
#else
   static mutex_t lock;
#endif /* _WIN32 */

// Maximum size of a logging record.
static const int LOG_RECORD_MAX = 1024;

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

// Entry point that writes logging records.
int write_record (char log_record[], int len) {
   /* ... */
   return 0;
}

// Entry point that processes logging records for
// one client connection.
#if defined (_WIN32)
   u_long
#else
   void *
#endif /* _WIN32 */
logging_handler (void *arg) {
   // Handle UNIX/Win32 portability.
#if defined (_WIN32)
   SOCKET h = reinterpret_cast <SOCKET> (arg);
#else
   int h = reinterpret_cast <int> (arg);
#endif /* _WIN32 */

   for (;;) {
#if defined (_WIN32)
       ULONG len;
#else
       UINT32 len;
#endif /* _WIN32 */

       // 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're shutdown or
       // errors occur unexpectedly.
       if (n <= sizeof len) break;
       len = ntohl (len);
       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;
       }

#if defined (_WIN32)
       EnterCriticalSection (&lock);
#else
       mutex_lock (&lock);
#endif /* _WIN32 */
       // Execute following two statements in a critical
       // section to avoid ace conditions and scrambled
       // output, respectively.
       ++request_count;
       // A return value of -1 signifies failure.
       if (write_record (log_record, len) == -1)
           break;

#if defined (_WIN32)
       LeaveCriticalSection (&lock);
#else
       mutex_unlock (&lock);
#endif /* _WIN32 */
   }

#if defined (_WIN32)
   closesocket (h);
#else
   close (h);
#endif /* _WIN32 */
   return 0;
}

// Main driver function for the server.
int main (int argc, char *argv[]) {
   struct sockaddr_in sock_addr;

   // Handle UNIX/Win32 portability.
#if defined (_WIN32)
   SOCKET acceptor;
   WORD version_requested = MAKEWORD(2, 0);
   WSADATA wsa_data;
   int error = WSAStartup(version_requested, &wsa_data);
   if (error != 0) return -1;
#else
   int acceptor;
#endif /* _WIN32 */
   // Create a local endpoint of communication.
   acceptor = socket (AF_INET, SOCK_STREAM, 0);
   // Set up the address to become a server.
   memset (reinterpret_cast<void *> (&sock_addr),
              0, sizeof sock_addr);
   sock_addr.sin_family = PF_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 (;;) {
       // Handle UNIX/Win32 portability.
#if defined (_WIN32)
       SOCKET h;
       DWORD t_id;
#else
       int h;
       thread_t t_id;
#endif /* _WIN32 */
       // Block waiting for clients to connect.
       h = accept (acceptor, 0, 0);
       // Spawn a new thread that runs the <server>
       // entry point.
#if defined (_WIN32)
       CreateThread (0, 0,
            LPTHREAD_START_ROUTINE(&logging_handler),
            reinterpret_cast <void *> (h), 0, &t_id);
#else
       thr_create
           (0, 0, logging_handler,
            reinterpret_cast <void *> (h),
            THR_DETACHED, &t_id);
#endif /* _WIN32 */
   }
   return 0;
}
```

While the design shown above may work for short-lived, 'throw-away' code [FoYo99] or prototypes, it is inadequate for software that must be maintained and evolved over time. By using conditional compilation directives and directly programming to low-level APIs, the code is unnecessarily hard to understand, debug, port, maintain, and evolve.

Some of these problems can be alleviated by moving platform-specific declarations, such as the types of mutexes and sockets, into separate configuration header files. This solution is incomplete, however, because the `#ifdefs` that separate the use of platform-specific APIs, such as the thread creation calls, will still pollute application code. Moreover, adding support for new platforms will require application developers to modify and update the platform-specific declarations, whether they are included directly into application code or separated into configuration files.

Several well-known patterns address similar problems. For instance, Facade [GHJV95] encapsulates object-oriented subsystems rather than low-level, non-object-oriented APIs. Decorator [GHJV95] extends an object *dynamically* by attaching additional responsibilities transparently. which incurs unnecessary performance overhead. Likewise, Bridge and Adapter [GHJV95] introduce an additional layer of indirection overhead. In general, therefore, these patterns are not well-suited to encapsulate existing lower-level non-object oriented APIs, where it is typically more important that the solution be efficient than dynamically extensible.

Context Maintainable and evolvable applications that access mechanisms or services provided byexisting lower-level, non-object-oriented APIs.

Problem Applications are often written using lower-level, non-object-oriented operating system APIs, such as those for network and thread programming, as well as other libraries, such as those for user interface or database programming. Although this design is common practice, it causes problems for application developers by failing to resolve the following *forces*:

- In general, concise code is more robust because it is easier to develop and maintain. Moreover, using languages that support higher-level features, such as constructors, destructors, exceptions, and garbage collection, reduces the likelihood of common programming errors. However, developers who program to lower-level, function-based APIs directly tend to rewrite a great deal of verbose and error-prone software repeatedly.

➥ The code for creating and initializing an acceptor socket in the `main()` function of our logging server example is prone to subtle errors, such as failing to initialize the `sock_addr` to `0` or not using the `htons()` macro to convert the `logging_port` number into network byte order [Sch92]. Moreover, the lack of constructors and destructors in C make it hard to ensure that resources are allocated and released properly. For example, note how the lock that serializes access to `request_count` will not be released correctly if the `write_record()` function returns `-1`. ❏

- Software that is ported, or can be readily ported, to different operating systems, compilers, and hardware platforms helps to increase product marketshare. Although reusing existing lower-level APIs helps reduce the overall software development effort, applications programmed directed to lower-level APIs are often non-portable. Moreover, programming to lower-level APIs across different versions of the *same* operating system or compiler is often non-portable, as a result of the absence of source-level or binary-level compatibility across software releases[Box97].

➥ Our logging server has hard-coded dependencies on several non-portable native operating system threading and network programming C APIs. For instance, `thr_create()`, `mutex_lock()` and `mutex_unlock()` are not portable to Win32 platforms. Although the code is designed to be quasi-portable—it also compiles and runs on Win32 platforms—there are various subtle portability problems. In particular, there will be resource leaks on Win32 platforms because there is no equivalent to the Solaris `THR_DETACHED` feature, which spawns a 'detached' thread whose exit status is not stored by the threading library [Lew95]. ❏

- Improving software maintainability helps reduce lifecycle costs. Programs written directly to low-level non-object-oriented APIs are often hard to maintain, however. For instance, C and C++ developers often address portability issues by embedding conditional compilation directives directly into their application source. Unfortunately, addressing platform-specific variations via conditional compilation at *all points of use* increases the *physical design* complexity [Lak95] of the software by scattering platform-specific implementation details throughout application source files.

➥ The maintainability of our logging server is impeded by the `#ifdef`s that handle Win32 and Solaris portability. For instance, several `#ifdef`s are required to handle differences in the type of a socket on Win32 and Solaris. In general, developers who program to low-level C APIs like these must have intimate knowledge of many operating system idiosyncrasies to maintain their code. ❑

- Cohesive components are easier to learn, enhance, and maintain. However, low-level APIs are not encapsulated into cohesive components supported by language features, such as classes, namespaces, or packages. This lack of cohesion makes it unnecessarily hard to comprehend the extent of low-level APIs. In addition, programming with non-cohesive stand-alone function APIs scatters common code throughout an application, making it hard to 'plug in' new components wholesale to support different policies and mechanisms.

➥ The socket API is particularly hard to learn because the several dozen functions in the socket library lack a uniform naming convention. Non-uniform naming makes it hard to determine the extent of the socket API. In particular, it is not immediately obvious that `socket()`, `bind()`, `listen()`, `connect()`, and `accept()` are related. Other low-level network programming APIs address this problem by prepending a common prefix before each function, such as the `t_` prefixed before each function in the TLI API. However, the use of a common prefix does not by itself make the TLI API more pluggable than sockets because it is still a low-level function-based API, rather than a more cohesive object-oriented class interface. ❑

In conclusion, developing applications by programming to lower-level, non-object-oriented APIs directly is a poor design choice for non-trivial software that must be maintained and evolved over time.

Solution Avoid accessing lower-level, non-object-oriented APIs directly. For each set of related functions and data in a lower-level API, create one or more *wrapper facade* classes that encapsulate these functions and data within more concise, robust, portable, and maintainable methods provided by higher-level object-oriented wrapper facade(s).

Structure There are two participants in the Wrapper Facade pattern.

Functions are the building blocks of existing lower-level APIs. They provide a single service or mechanism and manipulate data passed as parameters or accessed through global variables.

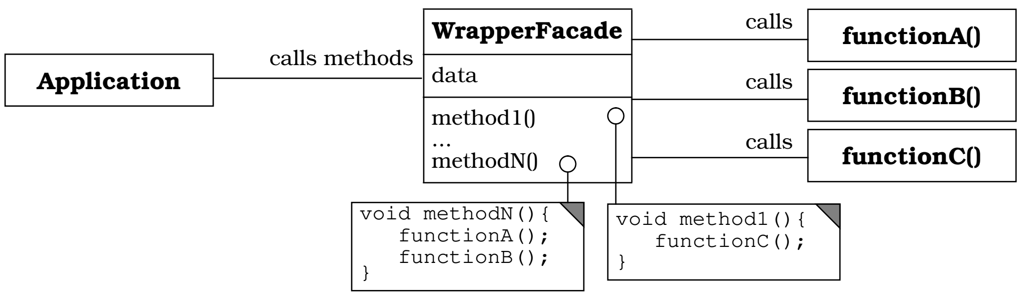
A *wrapper facade* is a set of one or more higher-level classes that encapsulate the lower-level functions and their associated data. These class(es) export a cohesive abstraction that provides a specific type of functionality. Each class represents a specific role in this abstraction.

The methods in the wrapper facade class(es) generally forward application invocations to one or more of the lower-level functions, passing the data as parameters. The data are typically hidden within the private portion of the wrapper facade and are not accessible to client applications. Moreover, compilers can enforce type-safety because primitive data types, such as pointers or integers, are encapsulated within strongly-typed wrapper facade classes.

Class Wrapper Facade	***Collaborator***
Responsibility • Encapsulates low-level functions and data-structures into a cohesive abstraction	• Functions

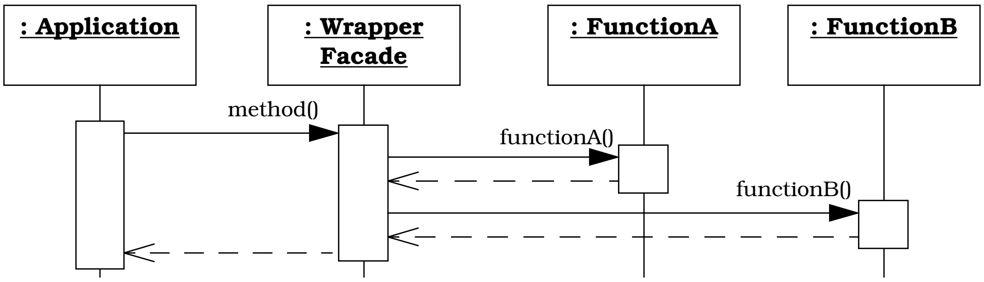
Class Function	***Collaborator***
Responsibility • Provides a single low-level service	

The following UML class diagram illustrates the structure of Wrapper Facade. [Frank, this figure is really great!]



Dynamics Collaborations in the Wrapper Facade pattern are typically straightforward:

- The application code invokes a method on an instance of the wrapper facade.
- The wrapper facade method forwards the request and its parameters to one or more of the lower-level functions that it encapsulates, passing along any internal data needed by the underlying function(s).



Implementation This section describes how to implement the Wrapper Facade pattern in C++. Certain activities may require multiple iterations to identify and implement suitable abstractions.

1 *Identify the cohesive abstractions and relationships among existing low-level APIs.* Mature low-level APIs, such as the C APIs for Win32 synchronization and threading, POSIX network programming, and X Windows GUI event loops, contain functions and data structures that define many cohesive abstractions and map cleanly onto object-oriented classes and methods. Due to the lack of data abstraction support in languages like C, however, it may not be immediately obvious how functions in these existing APIs relate to each other. The first step in implementing the Wrapper Facade pattern, therefore, is to identify the cohesive abstractions and relations among existing lower-level APIs.

➥ For instance, we first examine our original implementation of our logging server carefully. This implementation uses many low-level functions that provide several cohesive operating system mechanisms, such as synchronization and network communication. For instance, the Solaris `mutex_lock()` and `mutex_unlock()` functions are associated with a mutex synchronization abstraction. Likewise, the `socket()`, `bind()`, `listen()`, and `accept()` functions play various roles in a network programming abstraction. ❑

If existing functions and data structures were developed as throw-away code or piecemeal growth [FoYo99] they may not exhibit any cohesive abstractions. In this case, the code should be refactored [Opd92] [FBBOR99] before proceeding with the implementation of the Wrapper Facade pattern.

2 *Cluster cohesive groups of functions into wrapper facade classes and methods*. This step defines one or more class abstractions that shield applications from low-level data representations, arbitrary variations in function syntax, and other implementation details. This step can be decomposed into the following substeps:

2.1 *Create cohesive classes*. We start by defining one or more wrapper facade classes for each group of existing lower-level APIs that are related to a particular abstraction. Several common criteria used to create cohesive classes include the following:

- Coalesce functions and data with high *cohesion* into individual classes, while minimizing unnecessary *coupling* between classes. Common examples of cohesive functions are ones that manipulate a common data structure, such as a socket, a file, or a signal set [Ste97].
- Determine the *common* and *variable* aspects [Cope98] in the underlying functions and data. Aspects that are common typically include mechanisms for synchronization, threading, memory management, addressing formats, and operating system platform APIs. Aspects that vary typically include the implementations of these mechanisms. Whenever possible, variation in functions and data should be abstracted into classes that isolate variation behind uniform interfaces.

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

2.2 *Coalesce multiple individual functions into a single method.* In addition to grouping existing functions into classes, it may be beneficial to coalesce multiple individual functions into a smaller number of methods in each wrapper facade class. For instance, coalescing can be used to ensure that a group of lower-level functions are called in the appropriate order, which is a theme also shared by the Template Method pattern [GHJV95].

2.3 *Automate creation and destruction operations, if possible.* Lower-level APIs typically require programmers to call functions explicitly to create and destroy data structures that implement instances of the API. This procedure is error-prone, however, because developers may forget to call these functions in one or more paths through their code. A more robust approach, therefore, is to leverage the implicit creation and destruction operation capabilities provided by object-oriented languages, such as C++ and Java. In fact, even if all the wrapper facade methods simply forward to the lower-level function calls, the ability to create and destroy objects automatically can justify the use of the Wrapper Facade pattern.

2.4 *Select the level of indirection.* Most wrapper facade classes simply forward their method calls directly to the underlying low-level functions. Thus, if the wrapper facade methods can be inlined implicitly or explicitly there may be no run-time indirection overhead compared to invoking the low-level functions directly. In addition, it is possible to add another level of indirection by dispatching wrapper facade method implementations using dynamically bound methods or some other form of polymorphism. In this case, the wrapper facade classes play the role of the *abstraction* class in the Bridge pattern [GHJV95].

2.5 *Determine where to encapsulate any platform-specific variation.* A common use of the Wrapper Facade pattern is to minimize platform-specific variation in application code. Although wrapper facade method *implementations* may differ across different operating system platforms they should provide uniform, platform-independent *interfaces*. When platform-specific variation exists, there are two general strategies for encapsulating it: *conditional compilation* and *separate directories*.

- *Conditional compilation* can be used to selective among different wrapper facade class method implementations. The use of conditional compilation is inelegant and tedious when `#ifdefs` are littered throughout application code. Conditional compilation may be acceptable, however, if it is localized to a small number of platform-specific wrapper facade classes or files that are not accessed directly by application developers. Moreover, when conditional compilation is used in conjunction with auto-configuration tools, such as GNU `autoconf`, platform-independent wrapper facades can be created from a single source file. Therefore, as long as the number of variations supported in this file is not unwieldy, conditional compilation can actually help to localize variation and simplify the maintenance effort.

- *Separate directories* can be used to factor out different wrapper facade implementations, thereby avoiding or minimizing conditional compilation altogether. For example, each operating system platform can have its own directory containing implementations of platform-specific wrapper facades. Language processing tools can be used to include the appropriate wrapper facade class from the appropriate directory into applications at compile-time. To obtain a different implementation, a different include path could be provided to the compiler. This strategy avoids the problems with conditional compilation described above because it physically decouples the various alternative implementations into separate directories.

Choosing one of these strategies depends on how frequently the wrapper facade interfaces and implementations change. For instance, if changes occur frequently, it can be time-consuming to update the conditional compilation sections for each platform. Likewise, all files that depend on the affected files may need to be recompiled, even if the change is only necessary for one platform. Therefore, the larger the number of different platforms that must be supported, the more complex the use of condition compilation becomes.

Regardless of which strategy is selected, however, the burden of maintaining the wrapper facade implementations becomes the responsibility of wrapper facade developers, rather than application developers.

➥ For our logging server example, we define wrapper facade classes that encapsulate existing low-level C APIs for mutexes, sockets, and threads and illustrate how each of the issues outlined above can be addressed. We focus on defining wrapper facades for low-level C functions because C is commonly used to define widely-used operating system APIs, such as POSIX or Win32. However, the same design principles and techniques can be applied to other non-object-oriented languages, such as FORTRAN, Ada, Scheme, or Pascal, as well as to non-operating system APIs, such as X windows or database programming toolkits.[1]

Mutex wrapper facades. We first define a Thread_Mutex abstraction that encapsulates the Solaris mutex functions with 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 &);
};
```

Note how we define the copy constructor and assignment operator as private methods in the Thread_Mutex class. This C++ idiom, described in [Mey98] [Str98], ensures that application programmers cannot accidentally copy or assign one Thread_Mutex to another. Copying mutexes is a semantically invalid operation that is allowed by the less strongly-typed C programming API. Thus, our Thread_Mutex wrapper facade provides a mutex interface that is less error-prone than programming directly to the lower-level Solaris and Win32 functions.

By defining a Thread_Mutex class interface, and then writing applications to use it, rather than lower-level native operating system C APIs, we can easily port our wrapper facade to other platforms. For instance, the identical Thread_Mutex interface can be implemented to work on Win32:

1. We present an in-depth discussions of several concrete wrapper facades in this section since the implementation examples of other patterns in this book, including Acceptor-Connector (175), Strategized Locking (245), and Thread-Specific Storage (329), build upon the classes we present here. Although this makes the current discussion somewhat longer, it eliminates subsequent redundancy.

```
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 support multiple operating systems simultaneously by using conditional compilation and `#ifdef`'ing the `Thread_Mutex` method implementations. If conditional compilation is unwieldy due to the number of platforms that must be supported, it is possible to factor out the different `Thread_Mutex` implementations into separate directories. In this case, language processing tools, such as compilers and preprocessors, can be instructed to include the appropriate platform-specific variant into our application at compile-time.

Socket wrapper facades. Our next wrapper facade encapsulates the Socket API. This API is much larger and more expressive than the Solaris mutex API [Sch92]. Thus, we must define a *group* of related wrapper facade classes to encapsulate sockets. We start by defining a `typedef` and a macro that encapsulate UNIX/Win32 portability differences:

```
typedef int SOCKET;
#define INVALID_HANDLE_VALUE -1
```

Both `SOCKET` and `INVALID_HANDLE_VALUE` are defined in the Win32 API already. Naturally, we would either integrate this using `#ifdefs` or using separate platform-specific directories, as discussed earlier.

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

```
class INET_Addr {
public:
     INET_Addr (u_short port, u_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; }
     u_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. For instance, it zeros-out the `sockaddr_in` field and ensures the TCP port number and IP address are converted into network byte order by automatically applying the `ntons()` and `ntonl()` macros [Ste97].

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:
     // Default and copy constructor.
     SOCK_Stream (void) : handle_ (INVALID_HANDLE_VALUE){}
     SOCK_Stream (SOCKET h): handle_ (h) {}
     // Automatically close the handle on destruction.
     ~SOCK_Stream (void) {close (handle_); }
     // Set/get the underlying SOCKET handle.
     void set_handle (SOCKET h) {handle_ = h; }
     SOCKET get_handle (void) const {return handle_; }
     // Regular I/O operations.
     int recv (char *buf, size_t len, int flags);
     int send (const char *buf, size_t len, int flags);
     // I/O operations that handle "short" receives and
     // sends.
     int recv_n (char *buf, size_t len, int flags);
     int send_n (const char *buf, size_t len, int flags);
     // ... other methods omitted.
```

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

As discussed in *Implementation Activity 2.3*, this class leverages the semantics of C++ destructors to ensure that a socket handle is closed automatically when a `SOCK_Stream` object goes out of scope. In addition, note that the `send_n()` and `recv_n()` methods are designed to handle networking idiosyncrasies such as 'short' send and receive operations.

`SOCK_Stream` objects can be created by a connection factory, called `SOCK_Acceptor`, which encapsulates *passive* establishment of socket connections. The constructor of `SOCK_Acceptor` initializes the passive-mode acceptor socket to listen at the `sock_addr` address. The `SOCK_Acceptor::accept()` method is a factory that initializes the `SOCK_Stream` parameter with the newly accepted connection:

```
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 &s) {
          s.set_handle (accept (handle_, 0, 0));
          if (s.get_handle () == INVALID_HANDLE_VALUE)
               return -1;
          else
               return 0;
     }
private:
     // Socket handle factory.
     SOCKET handle_;
};
```

Note how the constructor for the `SOCK_Acceptor` applies the strategy discussed in *Implementation Activity 2.2* to ensure that the low-level `socket()`, `bind()`, and `listen()` functions are always called together and in the correct order.

A complete set of wrapper facades for sockets [Sch97] would also include a `SOCK_Connector`, which encapsulates the logic for actively establishing connections. The `SOCK_Acceptor` and `SOCK_Connector` classes are *concrete IPC mechanisms* that can be used to configure the generic acceptor and connector classes described in Acceptor-Connector pattern (175) to perform passive and active connection establishment.

Thread wrapper facade. Our final wrapper facade encapsulates operating system APIs related to multi-threading. Many threading APIs are available on different operating system platforms, including Solaris threads, POSIX Pthreads, and Win32 threads. These APIs exhibit subtle syntactic and semantic differences. For example, 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. The following `Thread_Manager` wrapper facade illustrates the `spawn` method implemented for Solaris threads:

```
class Thread_Manager {
public:
    int spawn (void *(*entry_point_function) (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_function,
                           arg,
                           flags,
                           t_id);
    }
    // ...
};
```

In addition, the `Thread_Manager` can provide methods for joining and canceling threads. These methods can be ported to most operating system platforms. ❑

2.6 *Consider allowing applications controlled access to implementation details.* One benefit of defining a wrapper facade is to make it hard to write incorrect or non-portable applications. For instance, wrapper facades can shield applications from error-prone or platform-specific implementation details, such as whether a socket is represented as a pointer or an integer. Use cases may arise, however, where the extra abstraction and type-safety actually *prevents* programmers from using a wrapper facade in useful ways that the original designer did not anticipate. This experience can be frustrating and may discourage programmers from leveraging the other benefits of wrapper facades. A common solution to this problem is to provide an 'escape hatch' mechanism or an 'open implementation' technique, such as AOP [KLM+97] , that allows applications to access implementation details in a controlled manner.

➥ For example, the `SOCK_Stream` class defines a pair of methods that set and get the underlying `SOCKET` handle, as follows:

```
class SOCK_Stream {
public:
     // Set/get the underlying SOCKET handle.
     void set_handle (SOCKET h) {handle_ = h; }
     SOCKET get_handle (void) const {return handle_;
}
```

These methods can be used to set/get certain socket options, such as support for 'out-of-band' data, that were not defined on the original socket wrapper facades. ❏

Naturally, escape hatch mechanisms should be used sparingly because they decrease portability and increase the potential for errors, which nullify key benefits of the Wrapper Facade pattern. Moreover, if applications tend to uses these escape hatches repeatedly in similar situations, it may indicate that explicit methods should be added to the public interface of the wrapper facade. The Extension Interface pattern (53) defines techniques for adding these new methods relatively unobtrusively.

3 *Determine an error handling mechanism.* Low-level C operating system function APIs typically use return values and integer codes, such as `errno`, to communicate errors back to their callers. This technique can be error-prone, however, if callers 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, support exception handling as a fundamental error reporting mechanism. It is also used in some operating systems, such as Win32 [Sol98]. [Frank, this rules!!! thanks!] There are several benefits to using exception handling as the error handling mechanism for wrapper facade classes:

- It is extensible, for instance, by defining hierarchies of exception classes in C++ and Java.
- It cleanly decouples error handling from normal processing. Error handling information is neither passed explicitly to an operation, nor can an application accidentally ignore an exception by failing to check function return values.
- It can be type-safe. In languages like C++ and Java exceptions are thrown and caught in a strongly-typed manner.

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

- Not all languages provide exception handling. For instance, some C++ compilers do not implement exceptions. Moreover, using proprietary exception handling mechanisms, such as Windows NT's structured exception handling [Sol98], can reduce the portability of applications that use these mechanisms.
- Languages implement exceptions in different ways, or do not implement exceptions at all. Thus, it can be hard to integrate components written in different languages when they throw exceptions.
- Resource management can be complicated if there are multiple exit paths from a block of C++ or Java code [Mue96]. 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.
- Poor implements of exception handling incur time and/or space overhead even when exceptions are *not* thrown [Mue96]. 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 operating system APIs, such as the mutex, socket, and thread wrapper facades described above, that must run portably on many platforms [Sch92]. One widely used error handling mechanism for the system-level wrapper facades is to use the Thread-Specific Storage pattern (329), in conjunction with `errno`. This solution is efficient, portable, and thread-safe, which is why we use this approach for most of the wrapper facades shown in this book.

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

➥ In our example, for instance, we can leverage the `Guard` template class defined in the Strategized Locking pattern (245) [Str98], which ensures that a `Thread_Mutex` is acquired and released properly within a scope, regardless of how the methods's flow of control exits the scope. The `Guard` class constructor automatically acquires the mutex and the destructor releases it 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.
}
```

Due to the fact that we used 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 and 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 is substantially harder to achieve this degree of 'pluggability' if lower-level C functions and data structures are used instead of C++ classes. The main problem, of course, is that the functions and data lack language support for cohesion, whereas the C++ classes provide this support naturally. ❏

Example Resolved The code below illustrates the `logging_handler()` function of the logging server after it has been rewritten to use our wrapper facades for mutexes, sockets, and threads described in the *Implementation* section. For comparison with the original code we present it in a two-column table with the original code from the example section in the left column, and the new code in the right column:

```
#if defined (_WIN32)
   #include <windows.h>
   typedef int ssize_t;
#else
   typedef unsigned int UINT32;
   #include <thread.h>
   #include <unistd.h>
   #include <sys/socket.h>
   #include <netinet/in.h>
   #include <memory.h>
#endif /* _WIN32 */

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

// Lock to protect request_count.
#if defined (_WIN32)
   static CRITICAL_SECTION lock;
#else
   static mutex_t lock;
#endif /* _WIN32 */

// Maximum size of a logging record.
static const int LOG_RECORD_MAX = 1024;

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

// Entry point that writes logging records.
int write_record (char log_record[], int len) {
   /* ... */
   return 0;
}

// Entry point that processes logging records for
// one client connection.
#if defined (_WIN32)
   u_long
#else
   void *
#endif /* _WIN32 */
logging_handler (void *arg) {
#if defined (_WIN32)
   SOCKET h = reinterpret_cast <SOCKET> (arg);
#else
   int h = reinterpret_cast <int> (arg);
#endif /* _WIN32 */

   for (;;) {
#if defined (_WIN32)
      ULONG len;
#else
      UINT32 len;
#endif /* _WIN32 */
      // 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're shutdown or
      // errors occur unexpectedly.
      if (n <= sizeof len) break;
      len = ntohl (len);
      if (len > LOG_RECORD_MAX) break;
```

```
#include "ThreadManager.h"
#include "ThreadMutex.h"
#include "Guard.h"
#include "INET_Addr.h"
#include "SOCKET.h"
#include "SOCK_Acceptor.h"
#include "SOCK_Stream.h"

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

// Maximum size of a logging record.
static const int LOG_RECORD_MAX = 1024;

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

// Entry point that writes logging records.
int write_record (char log_record[], int len) {
   /* ... */
   return 0;
}

// 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.
   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);
       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;
      }
#if defined (_WIN32)
      EnterCriticalSection (&lock);
#else
      mutex_lock (&lock);
#endif /* _WIN32 */
      // Execute following two statements in a critical
      // section to avoid race conditions and scrambled
      // output, respectively.
      ++request_count;
      // A return value of -1 signifies failure.
      if (write_record (log_record, len) == -1)
         break;
#if defined (_WIN32)
      LeaveCriticalSection (&lock);
#else
      mutex_unlock (&lock);
#endif /* _WIN32 */
   }
#if defined (_WIN32)
   closesocket (h);
#else
   close (h);
#endif /* _WIN32 */
   return 0;
}
```

```
      // 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 an error occurs
      if (n <= 0) break;{

         // Constructor of Guard auto-
         // matically 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;
         // A return value of -1 signifies failure.
         if (write_record (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 in the right column addresses the various problems with the code shown in the left column. For instance, the destructors of `SOCK_Stream` and `Guard` will close the socket handle and release the `Thread_Mutex`, respectively, regardless of how the blocks of code are exited. Likewise, this code is easier to understand, maintain, and port because it is much more concise and uses no platform-specific APIs.

Analogously to the `logging_handler()` function, we present a two-column table that compares the original code for the `main()` function with the new code using wrapper facades:

```
// Main driver function for the server.
int main (int argc, char *argv[]) {
   struct sockaddr_in sock_addr;
   // Handle UNIX/Win32 portability.
#if defined (_WIN32)
   SOCKET acceptor;
   WORD version_requested = MAKEWORD(2, 0);
   WSADATA wsa_data;
   int error = WSAStartup(version_requested, &wsa_data);
   if (error != 0) return -1;
#else
   int acceptor;
#endif /* _WIN32 */
   // Create a local endpoint of communication.
   acceptor = socket (AF_INET, SOCK_STREAM, 0);
   // Set up the address to become a server.
   memset (reinterpret_cast<void *> (&sock_addr),
              0, sizeof sock_addr);
   sock_addr.sin_family = PF_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 driver function for the server.
int main (int argc, char *argv[]) {
   INET_Addr addr (port);

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

```
   // Main server event loop.
   for (;;) {
       // Handle UNIX/Win32 portability.
#if defined (_WIN32)
       SOCKET h;
       DWORD t_id;
#else
       int h;
       thread_t t_id;
#endif /* _WIN32 */
       // Block waiting for clients to connect.
       h = accept (acceptor, 0, 0);

       // Spawn a new thread that runs the <server>
       // entry point.
#if defined (_WIN32)
       CreateThread (0, 0,
            LPTHREAD_START_ROUTINE(&logging_handler),
            reinterpret_cast <void *> (h), 0, &t_id);
#else
       thr_create
           (0, 0, logging_handler,
            reinterpret_cast <void *> (h),
            THR_DETACHED, &t_id);
#endif /* _WIN32 */
   }
   return 0;
}
```

```
   // Main server event loop.
   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);
   }

   return 0;
}
```

Known Uses **Microsoft Foundation Classes (MFC)**. MFC provides a set of wrapper facades that encapsulate many lower-level C Win32 APIs, focusing largely on providing GUI components that implement the Microsoft Document/Template architecture, which is a synonym for the Document-View architecture described in [POSA1].

ACE. The socket, thread, and mutex wrapper facades described in the *Implementation* section are abstractions of ACE framework [Sch97] components, such as the `ACE_SOCK*`, `ACE_Thread_Manager`, and `ACE_Thread_Mutex` classes, respectively.

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

ObjectSpace System<Toolkit>. Like Rogue Wave, the ObjectSpace System<Toolkit> implements wrapper facades for sockets, threads, and synchronization mechanisms.

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

Siemens REFORM. The REFORM framework for hot rolling mill process automation [BGHS98] uses the Wrapper Facade pattern to shield the object-oriented parts of the system, such as material tracking and setpoint transmission, from a neural network for the actual process control. This neural network is programmed in C due to its algorithmic nature: it contains mathematical models that describe the physics of the automation process.

The wrapper facades defined in the REFORM framework differ from wrapper facades for operating system mechanisms because the process control APIs they encapsulate are at a higher-level of abstraction. In fact, the neural network is part of the REFORM system itself. However, the REFORM wrapper facades have similar goals and properties as the lower-level operating system wrapper facades:

- They project the views and abstractions that the object-oriented parts of the framework need onto the process control neural network. For every component using the neural network, there is a separate wrapper facade.
- They hide API variations. For different customer-specific instances of the framework there might be (slightly) different implementations of the neural network. As a result, semantically identical functions in these neural network implementations may differ in their signature. These differences should not affect the framework implementation, however.
- They ensure the correct order in which lower-level C functions are invoked.

Consequences The Wrapper Facade pattern provides the following **benefits**:

Concise, cohesive, and robust higher-level object-oriented programming interfaces. The Wrapper Facade pattern encapsulates many lower-level APIs within a more concise and cohesive set of higher-level object-oriented classes. These abstractions reduce the tedium of developing applications, thereby decreasing the potential for certain types of programming errors. In addition, the use of encapsulation eliminates programming errors that occur when using untyped data structures, such as socket or file handles, incorrectly. Thus, application code can use wrapper facades to access lower-level APIs correctly and uniformly.

Portability and maintainability. Wrapper facade classes can be implemented to shield application developers from non-portable aspects of lower-level APIs. 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 [Lak95]. It is generally easier to understand and maintain applications in terms of their logical design rather than their physical design.

Modularity, reusability, and configurability. In addition to making it easier to reuse existing lower-level APIs, the Wrapper Facade pattern creates cohesive and reusable class components that can be 'plugged' in and out of other components in a wholesale fashion using object-oriented language features like inheritance and parameterized types. In contrast, it is much harder to replace groups of functions without resorting to coarse-grained operating system utilities, such as linkers or file systems.

The Wrapper Facade pattern has the following **liabilities**:

Loss of functionality. Whenever an abstraction is layered atop an existing abstraction it is possible to lose functionality. In particular, situations can occur where the new abstraction prevents developers from accessing certain capabilities of the underlying abstraction. Defining a suitable high-level abstraction that covers all these use cases without becoming bloated is hard. A useful heuristic to follow, therefore, is to design wrapper facades so they are easy to use correctly, hard to use incorrectly, but not impossible to use in ways that the original designers did not anticipate. Typically, some type of escape hatch mechanism or 'open implementation' [KLM+97] technique can help reconcile these design forces cleanly.

Performance degradation. The Wrapper Facade pattern can incur degrade performance. For instance, if wrapper facade classes are implemented with the Bridge pattern, or make several forwarding function calls per-method, the additional indirection may be more costly than programming to the lower-level APIs directly. However, languages that support inlining, such as C++, can implement the Wrapper Facade pattern with no significant overhead because compilers can inline the method calls used to implement the wrapper facades. Thus, the overhead is often the same as calling lower-level functions directly.

Programming language and compiler limitations. Defining C++ wrapper facades for well-design C APIs is relatively straightforward because the C++ language and C++ compilers define features that facilitate cross-language integration. It may be hard to define wrapper facades for other languages, however, due to lack of language support or limitations with compilers. For example, there is no universally accepted standard for integrating C functions into languages like Ada, Smalltalk, and Java. Therefore, programmers may need to resort to non-portable mechanisms to develop wrapper facades.

See Also The Wrapper Facade pattern is related to several of the structural patterns in [GHJV95], including Facade, Bridge, Adapter, and Decorator. Below, we compare and contrast Wrapper Facade with these patterns.

Facade. The intent of the Facade pattern is to provide a unified interface that simplifies client access to a set of subsystem interfaces. The intent of the Wrapper Facade pattern is more specific: it provides concise, robust, portable, maintainable, and cohesive class interfaces that encapsulate lower-level APIs, such as the native operating system 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 API relationships behind a richer object-oriented class API. Moreover, wrapper facades provide building block components that can be 'plugged' into higher-level components.

Bridge. The intent of the Bridge pattern is to decouple an abstraction from its implementation so that the implementation can vary independently and dynamically via polymorphism. The Wrapper Facade pattern has a similar intent, though its implementation rarely varies dynamically in order to minimize the overhead of indirection and polymorphism.

Adapter. The intent of the Adapter pattern is to convert the interface of a class into another interface expected by a client. A common application of the Wrapper Facade pattern is to create a set of classes that 'adapt' myriad low-level operating system APIs to create a portable set of wrapper facades that appear the same for all applications. Although the structure of this solution is not identical to either the object or class form of the Adapter pattern in[GHJV95], the wrapper facades play a similar role as an adapter by exporting an object-oriented interface that is common across platforms.

Decorator. The intent of the Decorator pattern is to extend an object *dynamically* by attaching additional responsibilities transparently. In contrast, the Wrapper Facade pattern *statically* encapsulates lower-level functions and data structures with object-oriented class interfaces.

In general, the Wrapper Facade pattern should be applied in lieu of these other patterns when there are existing lower-level APIs to encapsulate and when it is more important that the solution be efficient than dynamically extensible.

The Layers pattern [POSA1] helps with organizing multiple wrapper facades into a separate layer component. This layer resides directly atop the operating system and shields an application from all low-level APIs that it uses.

Credits Thanks to Brad Appleton, Luciano Gerber, Ralph Johnson, Bob Hanmer, Roger Whitney, and Joe Yoder for extensive comments that improved the form and content of the Wrapper Facade pattern description substantially.