Using Design Patterns to Evolve System Software from UNIX to Windows NT

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

Paul Stephenson
ebupsn@ebu.ericsson.se
Ericsson, Inc.
Cypress, CA 90630

An earlier version of the paper appeared in the March/April 1995 issue of the C++ Report magazine.

1 Introduction

Developing system software that is reusable across OS platforms is challenging. Due to constraints imposed by the underlying OS platforms, it is often impractical to directly reuse existing algorithms, detailed designs, interfaces, or implementations. This article describes our experiences using a large-scale reuse strategy for system software based on *design patterns*. Design patterns capture the static and dynamic structures of solutions that occur repeatedly when producing applications in a particular context [1, 2]. Design patterns are an important technique for improving system software quality since they address a fundamental challenge in large-scale software development: *communication of architectural knowledge among developers* [3].

This article describes our experiences with a large-scale reuse strategy based upon design patterns. We have used this strategy at Ericsson to facilitate the development of efficient OO telecommunication system software. In this article, we present a case study that describes the cross-platform evolution of portions of an OO framework called the ADAPTIVE Service eXecutive (ASX) [4]. The ASX framework is an integrated collection of components that collaborate to produce a reusable infrastructure for developing distributed applications.

This article focuses on the ASX framework's support for event-driven distributed applications. One of the key components in the ASX framework is the `Reactor` class category [5]. The `Reactor` integrates the demultiplexing of events and the dispatching of the corresponding event handlers. Event handlers are triggered by various types of events such as timers, synchronization objects, signals, or I/O operations.

We recently ported the ASX framework from several UNIX platforms to the Windows NT platform. These OS platforms possess significantly different mechanisms for event demultiplexing and I/O. To meet our performance requirements, it was not possible to directly reuse many of the components in the ASX framework across the OS platforms. However, it was possible to reuse the underlying design patterns that were embodied in the ASX framework, thereby reducing project risk.

The remainder of the article is organized as follows: Section 2 outlines the background of our work using OO frameworks for telecommunications system softare; Section 3.1 presents an overview of the design patterns that are the focus of this article; Section 4 examines the issues that arose as we ported the components in the `Reactor` framework from several UNIX platforms to the Windows NT platform; Section 5 summarizes the experience we gained, both pro and con, while deploying a design pattern-based system development methodology in a production software environment; and Section 6 presents concluding remarks.

2 Background

The design patterns and framework described in this article are currently being applied at Ericsson on a family of distributed applications [6]. These applications use ASX framework as the basis for a highly flexible and extensible telecommunication system management framework. The ASX framework enhances the flexibility and reuse of system software that monitors and manages telecommunication switch performance across multiple hardware and software platforms.

The system software we are developing provides essential services and mechanisms used by higher-level application software. Our system software frameworks are comprised of components that access and manipulate hardware devices (such as telecommunication switches) and software mechanisms residing within an OS kernel (such as alarms, interval timers, synchronization objects, communication ports, and signal handlers).

In general, developing system software that is capable of being directly reused on different OS platforms is challenging. Several factors complicating cross-platform reuse of system software are outlined below:

• **Efficiency:** Since applications and other reusable components will be layered upon system software, the techniques used to develop system software must not degrade performance significantly. Otherwise, developers will reinvent special-purpose code rather than reuse existing components, thereby defeating a major benefit of reuse.

• **Portability:** In order to meet performance and functionality requirements, system software often must access non-portable mechanisms and interfaces (such as device registers within a network link-layer controller or event demultiplexing mechanisms) provided by the underlying OS and hardware platform.

• **Lack of functionality:** many OS platforms do not provide adequate functionality to develop portable, reusable system components. For example, the lack of kernel-level multi-threading, explicit dynamic linking, and asynchronous exception handling (as well as robust compilers that interact correctly with these features) greatly increases the complexity of developing and porting reusable system software.

• **Need to master complex concepts:** Successfully developing robust, efficient, and portable system software requires intimate knowledge of complex mechanisms (such as concurrency control, interrupt handling, and interprocess communication) offered by multiple OS platforms. It is also essential to understand the performance costs associated with using alternative mechanisms (such as shared memory vs. message passing) on different OS platforms.

There are trade-offs among the factors described above that further complicate the reuse of system software across OS platforms. Often, it may be difficult to develop portable system software that does not significantly degrade efficiency or subtly alter the semantics and robustness of commonly used operations. For instance, many traditional OS kernels do not support pre-emptive multi-threading. Therefore, writing a portable user-level threads mechanism may be less efficient than programming with thread mechanisms supported by the kernel [7]. Likewise, user-level threads may reduce robustness by restricting the use of OS features such as signals or synchronous I/O operations.

3 Design Pattern Overview

A design pattern is a recurring architectural theme that provides a solution to a set of requirements within a particular context [1]. Design patterns facilitate architectural level reuse by providing "blueprints" or guidelines for defining, composing, and reasoning about the key components in a software system. In general, a large amount of reuse is possible at the architectural level. However, reusing design patterns does not necessarily result in direct reuse of algorithms, detailed designs, interfaces, or implementations.

OO frameworks typically embody a wide range of design patterns. For example, the ET++ graphical user-interface (GUI) framework [8] incorporates design patterns (such as Abstract Factory [1]) that hide the details of creating user-interface objects. This enables an application to be portable across different window systems (such as X windows and Microsoft Windows). Likewise, the InterViews [9] GUI framework contains design patterns (such as Strategy and Iterator [1]) that allow algorithms and/or application behavior to be decoupled from mechanisms provided by the reusable GUI components.

In the context of distributed applications, OO toolkits such as the Orbix CORBA object request broker [10] and the ADAPTIVE Service eXecutive (ASX) framework [4] embody many common design patterns. These design patterns express recurring architectural themes (such as event demultiplexing, connection establishment, message routing, publish/subscribe communication, remote object proxies, and flexible composition of hierarchically-related services) found in most distributed applications.

This article focuses on two specific design patterns (the Reactor [5] and Acceptor patterns) that are implemented by the ASX framework. Components in the ASX framework have been ported to a number of UNIX platforms, as well as Windows NT. The ASX components, and the Reactor and Acceptor design patterns embodied by these components, are currently used in a number of production systems. These systems include the Bellcore Q.port ATM signaling software product, the system control segment for the Motorola Iridium global personal communications system, and a family of system/network management applications for Ericsson telecommunication switches [6].

The design patterns described in the following section provided a concise set of architectural blueprints that guided our porting effort from UNIX to Windows NT. In particular, by employing the patterns, we did not have to rediscover the key collaborations between architectural components. Instead, our development task focused on determining a suitable mapping of the components in the pattern onto the mechanisms provided by the different OS platforms. Finding an appropriate mapping was non-trivial, as we describe below. Nevertheless, our knowledge of the design patterns significantly reduced redevelopment effort and minimized the level of risk in our projects.

3.1 The Reactor Pattern

The Reactor pattern is an object behavioral pattern [1]. This pattern simplifies the development of event-driven applications (such as a CORBA ORB [10], an X-windows host resource manager, or a distributed logging service [5]). The Reactor pattern provides a common infrastructure that integrates event demultiplexing and the dispatching of event handlers. Event handlers perform application-specific processing operations in response to various types of events. An event handler may be triggered by different sources of events (such as timers, communication ports, synchronization objects, and signal handlers) that are monitored by an application. The callback-driven programming style provided by a Motif or Windows application is a prime example of the Reactor pattern.

The Reactor pattern provides several major benefits for event-driven distributed applications:

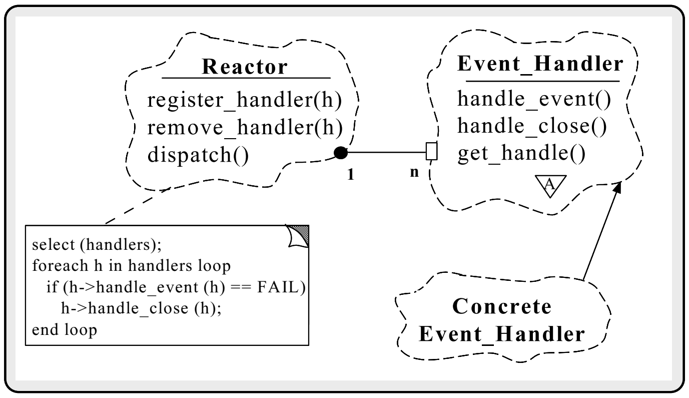

Figure 1: The Structure of Participants in the Reactor Pattern

• **Improve performance:** it enables an application to wait for activity to occur on multiple sources of events simultaneously *without* blocking or continuously polling for events on any single source.

• **Minimize synchronization complexity:** it provides applications with a low-overhead, coarse-grained form of concurrency control. The Reactor pattern serializes the invocation of event handlers at the level of "event demultiplexing and dispatching" within a single process or thread. For many applications, this eliminates the need for more complicated synchronization or locking.

• **Enchance reuse:** it decouples application-specific functionality from application-independent mechanisms. Application-specific functionality is performed by user-defined methods that override virtual functions inherited from an event handler base class. Application-independent mechanisms are reusable components that demultiplex events and dispatch pre-registered event handlers.

Figure 1 illustrates the structure of participants in the Reactor pattern.[1] The `Reactor` class defines an interface for registering, removing, and dispatching `Event_Handler` objects. An implementation of the `Reactor` pattern provides application-independent mechanisms that perform event demultiplexing and dispatch application-specific concrete event handlers. The `Reactor` class contains references to objects of `Concrete_Event_Handler` subclasses. These subclasses are derived from the `Event_Handler` abstract base class, which defines virtual methods for handling events. A `Concrete_Event_Handler` subclass may override these virtual methods to perform application-specific functionality when the corresponding events occur.

[1]Relationships between components are illustrated throughout the article via Booch notation [11]. Dashed clouds indicate classes; non-dashed directed edges indicate inheritance relationships between classes; dashed directed edges indicate a template instantiation relationship; and an undirected edge with a solid bullet at one end indicates a composition relation. Solid clouds indicate objects; nesting indicates composition relationships between objects; and undirected edges indicate some type of link exists between objects.

The `Reactor` triggers `Event_Handler` methods in response to events. These events may be associated with handles that are bound to sources of events (such as I/O ports, synchronization objects, or signals). To bind the `Reactor` together with these handles, a subclass of `Event_Handler` must override the `get_handle` method. When the `Reactor` registers an `Event_Handler` subclass object, the the object's handle is obtained by invoking its `Event_Handler::get_handle` method. The `Reactor` then combines this handle with other registered `Event_Handlers` and waits for events to occur on the handle(s).

The code annotation in Figure 1 outlines the behavior of the `dispatch` method. When events occur, the `Reactor` uses the handles activated by the events as keys to locate and dispatch the appropriate `Event_Handler` methods. The `handle_event` method is then invoked by the `Reactor` as a "callback." This method performs application-specific functionality in response to an event. If a call to `handle_event` fails, the `Reactor` invokes the `handle_close` method. This method performs any application-specific cleanup operations. When the `handle_close` method returns, the `Reactor` removes the `Event_Handler` subclass object from its internal tables.

An alternative way to implement event demultiplexing and dispatching is to use multi-tasking. In this approach, an application spawns a separate thread or process that monitors an event source. Every thread or process blocks until it receives an event notification. At this point, the appropriate event handler code is executed. Certain types of applications (such as file transfer, remote login, or teleconferencing) benefit from multi-tasking. For these applications, multi-threading or multi-processing helps to reduce development effort, improves application robustness, and transparently leverages off of available multi-processor capabilities.

Using multi-threading to implement event demultiplexing has several drawbacks, however. It may require the use of complex concurrency control schemes; it may lead to poor performance on uni-processors [4]; and it may not be available on widely available OS platforms (such as many variants of UNIX). In these cases, the Reactor pattern may be used in lieu of, or in conjunction with, OS multi-threading or multi-processing mechanisms, as described in Section 3.2.

3.2 The Acceptor Pattern

The Acceptor pattern is an object creational pattern [1] that decouples the act of establishing a connection from the service(s) provided after a connection is established. Connection-oriented services (such as file transfer, remote login, distributed logging, and video-on-demand) are particularly amenable to this pattern. The Acceptor pattern simplifies the development of these services by allowing the application-specific portion of a service to be modified independently of the mechanism used to establish the connection. The UNIX "superserver" `inetd` is a prime example of an application that uses the Acceptor pattern.

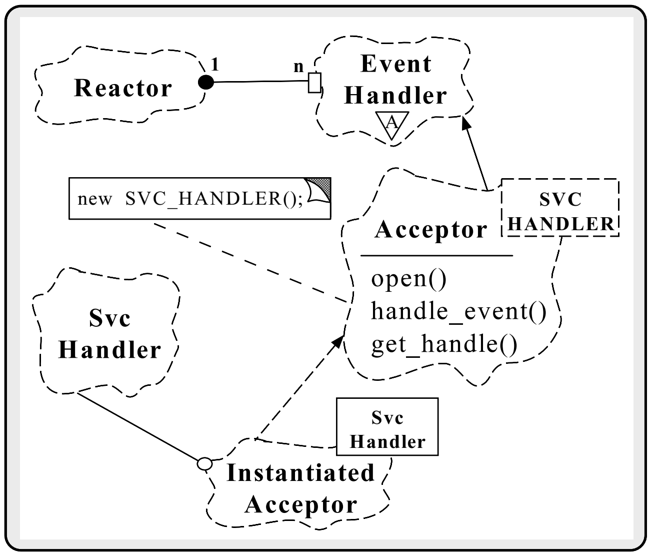

Figure 2: The Structure of Participants in the Acceptor Pattern

To build upon the interfaces and mechanisms already provided by the Reactor pattern, the `Acceptor` class inherits the `Event_Handler`'s demultiplexing and dispatching interface (shown in Figure 1). Figure 2 illustrates the structure of participants in the Acceptor pattern. The `open` method in template class `Acceptor` initializes a communication endpoint and listens for incoming connection requests from clients. The `get_handle` method returns the I/O handle corresponding to the communication endpoint.

When a connection request arrives from a client the `Reactor` triggers a callback on the `Acceptor`'s `handle_event` method. This method is a *factory* that dynamically produces a new `SVC_HANDLER` object In the example in Figure 2, `SVC_HANDLER` is a formal parameterized type argument in the `Acceptor` template class. The `Instantiated_Acceptor` class supplies an actual `Svc_Handler` class parameter. The `Svc_Handler` parameter implements a particular application-specific service (*i.e.,* transferring a file, permitting remote login, receiving logging records, sending a video sequence, etc.).

Note that the Acceptor pattern does not dictate the behavior or concurrency dynamics of the `Svc_Handler` service it creates. In particular, a dynamically created `Svc_Handler` service may be executed in any of the following ways:

- **Run in the same thread of control:** This approach may be implemented by inheriting the `Svc_Handler` from `Event_Handler` and registering each newly created `Svc_Handler` object with the `Reactor`. Thus, each `Svc_Handler` object is dispatched in the same thread of control as an `Acceptor` object. The implementation described in Section 4 uses this single-threaded behavior.

- **Run in a separate thread of control:** In this approach, the `Reactor` serves as the master connection dispatcher within an application. When a client connects, the `Acceptor`'s `handle_event` method spawns a separate thread of control. The `Svc_Handler` object then processes messages exchanged over the connection within the new slave thread. Threads are useful for cooperating services that frequently reference common memory-resident data structures shared by the threads within a process address space [7].

- **Run in a separate OS process:** This approach is similar in form to the previous bullet. However, a separate process is created rather than a separate thread. Network services that base their security and protection mechanisms on process ownership are typically executed in separate processes to prevent accidental or intentional access to unauthorized resources. For example, the standard UNIX superserver, `inetd`, uses the Acceptor pattern in this manner to execute the standard Internet `ftp` and `telnet` services in separate processes [12].

The ASX framework described in [4] provides mechanisms that support all three of these types of concurrency dynamics. Moreover, the selection of concurrency mechanism may be deferred until late in the design, or even until run-time. This flexibility increases the range of design alternatives available to developers.

In addition, the Acceptor pattern may be used to develop highly extensible event handlers that may be configured into an application at installation-time or at run-time. This enables applications to be updated and extended without modifying, recompiling, relinking, or restarting the applications at run-time. Achieving this degree of flexibility and extensibility requires the use of OO language features (such as templates, inheritance, and dynamic binding), OO design techniques (such as the Factory Method or Abstract Factory design patterns [1]), and advanced operating system mechanisms (such as explicit dynamic linking and multi-threading [4]).

4 Evolving Design Patterns Across OS Platforms

4.1 Motivation

Based on our experience at Ericsson, explicitly modeling design patterns is a very beneficial activity. In particular, design patterns focus attention on relatively stable aspects of a system's software architecture. They also emphasize the strategic collaborations between key participants in the architecture without overwhelming developers with excessive detail. Abstracting away from low-level implementation details is particularly important for system software since OS platform constraints often preclude direct reuse of system components.

In our experience, it is essential to illustrate how design patterns are realized in actual systems. One observation we discuss in Section 5 is that existing design pattern catalogs [1, 2] do not present "wide spectrum" coverage of patterns. Often, this makes it difficult for novices to recognize how to apply patterns in practice on their projects. We believe the development sequence that unfolds in this section will provide a technically rich, motivating, and detailed (yet comprehensible) roadmap to help shepard other developers into the realm of patterns.

With these goals in mind, this section outlines how the Reactor and Acceptor design patterns were implemented and evolved on BSD and System V UNIX, as well as on Windows NT. The discussion emphasizes the relevant functional differences between the various OS platforms and describes how these differences affected the implementation of the design patterns. To focus the discussion below, C++ is used as the implementation language. However, the principles and concepts underlying the Reactor and Acceptor patterns are independent of the programming language, the OS platform, and any particular implementation. Readers who are not interested in the lower-level details of implementing design patterns may wish to skip ahead to Section 5, where we summarize the lessons we learned from using design patterns on several projects at Ericsson.

4.2 The Impact of Platform Demultiplexing and I/O Semantics

The implementation of the Reactor pattern was affected significantly by the semantics of the event demultiplexing and I/O mechanisms in the underlying OS. In general, there are two types of demultiplexing and I/O semantics: *reactive* and *proactive*. Reactive semantics allow an application to inform the OS which I/O handles to notify it about when an I/O-related operation (such as a read, write, and connection request/accept) may be performed without blocking. Subsequently, when the OS detects that the desired operation may be performed without blocking on any of the indicated handles, it informs the application that the handle(s) are ready. The application then "reacts" by processing the handle(s) accordingly (such as reading or writing data, accepting connections, etc.). Reactive demultiplexing and I/O semantics are provided on standard BSD and System V UNIX systems [12].

In contrast, proactive semantics allow an application to proactively initiate I/O-related operations (such as a read, write, or connection request/accept) or general-purpose event-signaling operations (such as a semaphore lock being acquired or a thread terminating). The invoked operation proceeds asynchronously and does not block the caller. When an operation completes, it signals the application. At this point, the application runs a completion routine that determines the exit status of the operation and potentially starts up another asynchronous operation. Proactive demultiplexing and I/O semantics are provided on Windows NT [13] and VMS.

For performance reasons, we were not able to completely encapsulate the variation in behavior between the UNIX and Windows NT demultiplexing and I/O semantics. Thus, we could not directly reuse existing C++ code, algorithms, or detailed designs. However, it was possible to capture and reuse the concepts that underlay the Reactor and Acceptor design patterns.

4.3 UNIX Evolution of the Patterns

4.3.1 Implementing the Reactor Pattern on UNIX

The standard demultiplexing mechanisms on UNIX operating systems provide reactive I/O semantics. For instance, the UNIX `select` and `poll` event demultiplexing system calls inform an application which subset of handles within a set of I/O handles may send/receive messages or request/accept connections without blocking. Implementing the Reactor pattern using UNIX reactive I/O is straightforward. After `select` or `poll` indicate which I/O handles have become ready, the `Reactor` object reacts by invoking the appropriate `Event_Handler` callback methods (*i.e.*, `handle_event` or `handle_close`).

One advantage of the UNIX reactive I/O scheme is that it decouples (1) event detection and notification from (2) the operation performed in response to the triggered event. This allows an application to optimize its response to an event by using context information available when the event occurs. For example, when `select` indicates a "read" event is pending, a network server might check to see how many bytes are in a socket receive queue. It might use this information to optimize the buffer size it allocates before making a `recv` system call. A disadvantage of UNIX reactive I/O is that operations may not be invoked asynchronously with other operations. Therefore, computation and communication may not occur in parallel unless separate threads or processes are used.

The original implementation of the Reactor pattern provided by the `ASX` framework was derived from the `Dispatcher` class category available in the InterViews OO GUI framework [9]. The `Dispatcher` is an OO interface to the UNIX `select` system call. InterViews uses the `Dispatcher` to define an application's main event loop and to manage connections to one or more physical window displays. The `Reactor` framework's first modification to the `Dispatcher` framework added support for signal-based event dispatching. The `Reactor`'s signal-based dispatching mechanism was modeled closely on the `Dispatcher`'s existing timer-based and I/O handle-based event demultiplexing and event handler dispatching mechanisms.[2]

The next modification to the `Reactor` occurred when porting it from SunOS 4.x (which is based primarily on BSD 4.3 UNIX) to SunOS 5.x (which is based primarily on System V release 4 (SVR4) UNIX). SVR4 provides another event demultiplexing system call named `poll`. `Poll` is similar to `select`, though it uses a different interface and provides a broader, more flexible model for event demultiplexing that supports SVR4 features such as STREAM pipe band-data [12].

[2]The `Reactor`'s interfaces for signals and timer-based event handling are not shown in this article due to space limitations.

The SunOS 5.x port of the `Reactor` was enhanced to support either `select` or `poll` as the underlying event demultiplexer. Although portions of the `Reactor`'s internal implementation changed, its external interface remained the same for both the `select`-based and the `poll`-based versions. This common interface improves networking application portability across BSD and SVR4 UNIX platforms.

A portion of the public interface for the BSD and SVR4 UNIX implementation of the Reactor pattern is shown below:

```
// Bit-wise "or" these values to check
// for multiple activities per-handle.
enum Reactor_Mask { READ_MASK = 01,
  WRITE_MASK = 02, EXCEPT_MASK = 04 };

class Reactor
{
public:
  // Register an Event_Handler object according
  // to the Reactor_Mask(s) (i.e., "reading,"
  // "writing," and/or "exceptions").
  virtual int register_handler (Event_Handler *,
                                Reactor_Mask);

  // Remove the handler associated with
  // the appropriate Reactor_Mask(s).
  virtual int remove_handler (Event_Handler *,
                              Reactor_Mask);

  // Block process until I/O events occur or
  // a timer expires, then dispatch Event_Handler(s).
  virtual int dispatch (void);

// ...
};
```

Likewise, the `Event_Handler` interface for UNIX is defined as follows:

```
typedef int HANDLE; // I/O handle.

class Event_Handler
{
protected:
  // Returns the I/O handle associated with the
  // derived object (must be supplied by a subclass).
  virtual HANDLE get_handle (void) const;

  // Called when an event occurs on the HANDLE.
  virtual int handle_event (HANDLE, Reactor_Mask);

  // Called when object is removed from the Reactor.
  virtual int handle_close (HANDLE, Reactor_Mask);

// ...
};
```

The next major modification to the `Reactor` extended it for use with multi-threaded applications on SunOS 5.x using Solaris threads [7]. Adding multi-threading support required changes to the internals of both the `select`-based and `poll`-based versions of the `Reactor`. These changes involved a SunOS 5.x mutual exclusion mechanism known as a "mutex." A mutex serializes the execution of multiple threads by defining a critical section where only one thread executes the code at a time [7]. Critical sections of the `Reactor`'s code that concurrently access shared resources (such as the `Reactor`'s internal dispatch table containing `Event_Handler` objects) are protected by a mutex.

The standard SunOS 5.x synchronization type (`mutex_t`) provides support for *non-recursive* mutexes. The SunOS 5.x non-recursive mutex provides a simple and efficient form of mutual exclusion based on adaptive spin-locks. However, non-recursive mutexes possess the restriction that the thread currently owning a mutex may not reacquire the mutex without releasing it first. Otherwise, deadlock will occur immediately.

While developing the multi-threaded `Reactor`, it quickly became obvious that SunOS 5.x mutex variables were inadequate to support the synchronization semantics required by the `Reactor`. In particular, the Reactor's `dispatch` interface performs callbacks to methods of pre-registered, application-specific event handler objects as follows:

```
void Reactor::dispatch (void)
{
  for (;;) {
    // Block until events occur.
    this->wait_for_events (this->handler_set);
    // Obtain the mutex.
    this->lock->acquire ();

    // Dispatch all the callback methods
    // on handlers who contain active events.
    foreach handler in this->handler_set {
      if (handler->handle_event
          (handler, mask) == FAIL)
        // Cleanup on failure.
        handler->handle_close (handler);
    }
    // Release the mutex.
    this->lock->release ();
  }
}
```

Callback methods (such as `handle_event` and `handle_close`) defined by `Event_Handler` subclass objects may subsequently re-enter the `Reactor` object by calling its `register_handler` and `remove_handler` methods as follows:

```
// Global per-process instance of the Reactor.
extern Reactor reactor;

// Application-specific method called
// back by the Reactor.

int Acceptor::handle_event (HANDLE handle,
                            Reactor_Mask)
{
  Concrete_Event_Handler *new_handler =
    new Concrete_Event_Handler;

  *new_handler = this->accept (handle);

  // Re-enter the Reactor object.
  reactor.register_handler (new_handler,
                            READ_MASK);
  // ...
}
```

In the code fragment shown above, non-recursive mutexes will result in deadlock since (1) the mutex within the `Reactor`'s `dispatch` method is locked throughout the callback and (2) the `Reactor`'s `register_handler` method tries to acquire the same mutex.

One solution to this problem involved recoding the `Reactor` to release its mutex lock before invoking callbacks to application-specific `Event_Handler` methods. However, this solution was tedious and error-prone. It also increased synchronization overhead by repeatedly releasing

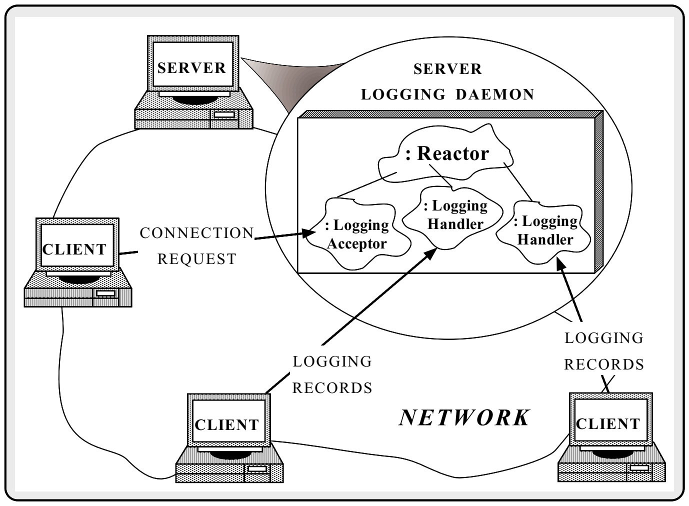

Figure 3: The Distributed Logging Service

and reacquiring mutex locks. A more elegant and efficient solution used *recursive* mutexes to prevent deadlock and to avoid modifying the `Reactor`'s concurrency control scheme. A recursive mutex allows calls to its `acquire` method to be nested as long as the thread that owns the lock is the one attempting to re-acquire it.

The current implementation of the UNIX-based Reactor pattern is about 2,400 lines of C++ code (not including comments or extraneous whitespace). This implementation is portable between both BSD and System V UNIX variants.

4.3.2 Implementing the Acceptor Pattern on UNIX

To illustrate the Reactor and Acceptor patterns, consider the event-driven server for a distributed logging service shown in Figure 3. Client applications use this service to log information (such as error notifications, debugging traces, and status updates) in a distributed environment. In this service, logging records are sent to a central logging server. The logging server outputs the logging records to a console, a printer, a file, or a network management database, etc.

In the architecture of the distributed logging service, the logging server shown in Figure 3 handles logging records and connection requests sent by clients. These records and requests may arrive concurrently on multiple I/O handles. An I/O handle identifies a resource control block managed by the operating system.[3]

The logging server listens on one I/O handle for connection requests to arrive from new clients. In addition, a separate I/O handle is associated with each connected client. Input from multiple clients may arrive concurrently. Therefore, a single-threaded server must not block indefinitely reading from any individual I/O handle. A blocking `read` on one handle may significantly delay the response time for clients associated on other handles.

A highly modular and extensible way to design the server logging daemon is to combine the Reactor and Acceptor patterns. Together, these patterns decouple (1) the application-independent mechanisms that demultiplex and dispatch pre-registered `Event_Handler` objects from (2) the application-specific connection establishment and logging record transfer functionality performed by methods in these objects.

Within the server logging daemon, two subclasses of the `Event_Handler` base class (`Logging_Handler` and `Logging_Acceptor`) perform the actions required to process the different types of events arriving on different I/O handles. The `Logging_Handler` event handler is responsible for receiving and processing logging records transmitted from a client. Likewise, the `Logging_Acceptor` event handler is a factory that is responsible for accepting a new connection request from a client, dynamically allocating a new `Logging_Handler` event handler to handle logging records from this client, and registering the new handler with an instance of a `Reactor` object.

The following code illustrates an implementation the server logging daemon based upon the Reactor and Acceptor patterns. An instance of the `Logging_Handler` template class performs I/O between the server logging daemon and a particular instance of a client logging daemon. As shown in the code below, the `Logging_Handler` class inherits from `Event_Handler`. Inheriting from `Event_Handler` enables a `Logging_Handler` object to be registered with the `Reactor`. This inheritance also allows a `Logging_Handler` object's `handle_event` method to be dispatched automatically by a `Reactor` object to process logging records when they arrive from clients. The `Logging_Handler` class contains an instance of the template parameter `PEER_IO`. The `PEER_IO` class provides reliable TCP capabilities used to transfer logging records between an application and the server. The use of templates removes the reliance on a particular IPC interface (such as BSD sockets or System V TLI).

```
template <class PEER_IO>
class Logging_Handler
  : public Event_Handler
{
public:
  // Callback method that handles the reception
  // of logging transmissions from remote clients.
  // Two recv()'s are used to maintain framing
  // across a TCP bytestream.

  virtual int handle_event (HANDLE, Reactor_Mask) {
    long len;
    // Determine logging record length.
    long n = this->peer_io_.recv (&len, sizeof len);

    if (n <= 0) return n;
    else {
      Log_Record log_record;

      // Convert from network to host byte-order.
      len = ntohl (len);
      // Read remaining data in record.
      this->peer_io_.recv (&log_record, len);
```

[3]Different operating systems use different terms for I/O handles. For example, UNIX programmers typically refer to these as *file descriptors*, whereas Windows programmers typically refer to them as *I/O HANDLEs*. In both cases, the underlying concepts are the same.

```
      // Format and print the logging record.
      log_record.decode_and_print ();
      return 0;
    }
  }

  // Retrieve the I/O handle (called by Reactor
  // when Logging_Handler object is registered).

  virtual HANDLE get_handle (void) const {
    return this->peer_io_.get_handle ();
  }


  // Close down the I/O handle and delete the
  // object when a client closes the connection.

  virtual int handle_close (HANDLE,
                            Reactor_Mask) {
    delete this;
    return 0;
  }

private:
  // Private ensures dynamic allocation.
  ~Logging_Handler (void) {
    this->peer_io_.close ();
  }

   // C++ wrapper for data transfer.
   PEER_IO peer_io_;
};
```

The `Logging_Acceptor` template class is shown in the C++ code below. It is a generic factory that performs the steps necessary to (1) accept connection requests from client logging daemons and (2) create `SVC_HANDLER` objects that are used to perform an actual application-specific service on behalf of clients. Note that the `Logging_Acceptor` object and the `SVC_HANDLER` objects it creates run within the same thread of control. Logging record processing is driven reactively by method callbacks triggered by the `Reactor`.

The `Logging_Acceptor` subclass inherits from the `Event_Handler` class. Inheriting from the `Event_Handler` class enables an `Logging_Acceptor` object to be registered with the `Reactor`. The `Reactor` subsequently dispatches the `Logging_Acceptor` object's `handle_event` method. This method then invokes `SOCK_Acceptor::accept`, which accepts a new client connection. The `Logging_Acceptor` class also contains an instance of the template parameter `PEER_Acceptor`. The `PEER_Acceptor` class is a factory that listens for connection requests on a well-known communication port and accepts connections when they arrive on that port from clients.

```
// Global per-process instance of the Reactor.
extern Reactor reactor;

// Handles connection requests
// from a remote client.

template <class SVC_HANDLER,
          class PEER_Acceptor,
          class PEER_ADDR>
class Logging_Acceptor
  : public Event_Handler
{
public:

  // Initialize the Acceptor endpoint.
```

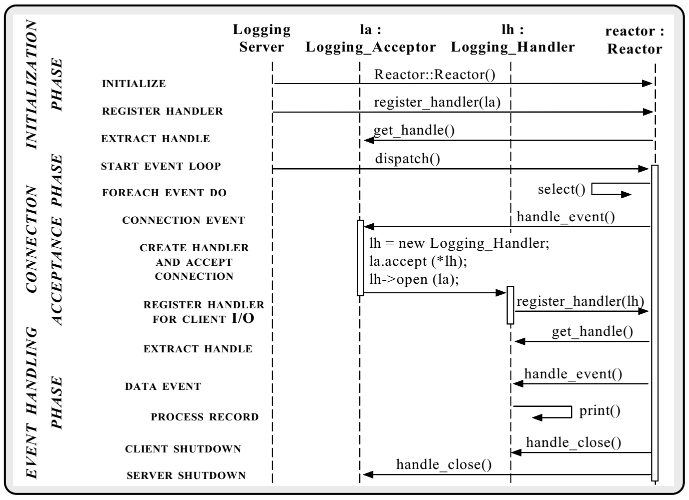

Figure 4: Server Logging Daemon Interaction Diagram

```
  Logging_Acceptor (PEER_ADDR &addr)
    : peer_Acceptor_ (addr) {}

  // Callback method that accepts a new
  // connection, creates a new SVC_HANDLER object
  // to perform I/O with the client connection,
  // and registers the new object with the Reactor.

  virtual int handle_event (HANDLE, Reactor_Mask) {
    SVC_HANDLER *handler = new SVC_HANDLER;

    this->peer_Acceptor_.accept (*handler);
    reactor.register_handler (handler, READ_MASK);
    return 0;
  }

  // Retrieve the I/O handle (called by Reactor
  // when an Logging_Acceptor object is registered).

  virtual HANDLE get_handle (void) const {
    return this->peer_Acceptor_.get_handle ();
  }

  // Close down the I/O handle when the
  // Logging_Acceptor is shut down.

  virtual int handle_close (HANDLE,
                            Reactor_Mask) {
    return this->peer_Acceptor_.close ();
  }
private:
  // Factory that accepts client connections.
  PEER_Acceptor peer_Acceptor_;
};
```

The C++ code shown below illustrates the main entry point into the server logging daemon. This code creates a `Reactor` object and an `Logging_Acceptor` object and registers the `Logging_Acceptor` with the `Reactor`. Note that the `Logging_Acceptor` template is instantiated with the `Logging_Handler` class, which performs the distributed logging service on behalf of clients. Next, the main program calls `dispatch` and enters the `Reactor`'s event-loop. The `dispatch` method continuously handles connection requests and logging records that arrive from clients.

The interaction diagram shown in Figure 4 illustrates the collaboration between the various objects in the server logging daemon at run-time. Note that once the `Reactor` object is initialized, it becomes the primary focus of the control flow within the server logging daemon. All subsequent activity is triggered by callback methods on the event handlers controlled by the `Reactor`.

```
// Global per-process instance of the Reactor.
Reactor reactor;

// Server port number.
const unsigned int PORT = 10000;

// Instantiate the Logging_Handler template.
typedef Logging_Handler <SOCK_Stream>
        LOGGING_HANDLER;

// Instantiate the Logging_Acceptor template.
typedef Logging_Acceptor<LOGGING_HANDLER,
                         SOCK_Acceptor,
                         INET_Addr>
        LOGGING_Acceptor;

int
main (void)
{
  // Logging server address and port number.
  INET_Addr addr (PORT);
  // Initialize logging server endpoint.
  LOGGING_Acceptor Acceptor (addr);

  reactor.register_handler (&Acceptor, READ_MASK);

  // Main event loop that handles client
  // logging records and connection requests.
  reactor.dispatch ();
  /* NOTREACHED */
  return 0;
}
```

The C++ code example shown above uses templates to decouple the reliance on the particular type of IPC interface used for connection establishment and communication. The `SOCK_Stream`, `SOCK_Acceptor` and `INET_Addr` classes used in the template instantiations are part of the `SOCK_SAP` C++ wrapper library [14]. `SOCK_SAP` encapsulates the `SOCK_STREAM` semantics of the socket transport layer interface within a type-secure, OO interface. `SOCK_STREAM` sockets support the reliable transfer of bytestream data between two processes, which may run on the same or on different host machines in a network [12].

By using templates, it is relatively straightforward to instantiate a different IPC interface (such as the `TLI_SAP` C++ wrappers that encapsulate the System V UNIX TLI interface). Templates trade additional compile-time and link-time overhead for improved run-time efficiency. Note that a similar degree of decoupling also could be achieved via inheritance and dynamic binding by using the Abstract Factory or Factory Method patterns described in [1].

4.4 Evolving the Design Patterns to Windows NT

This section describes the Windows NT implementation of the Reactor and Acceptor design patterns performed at the Ericsson facility in Cypress, California. Initially, we attempted to evolve the existing `Reactor` implementation from UNIX to Windows NT using the `select` function from the Windows Sockets (WinSock) library.[4] This approach failed because the WinSock version of `select` does not interoperate with standard Win32[5] I/O HANDLEs. Our applications required the use of Win32 I/O HANDLEs to support network protocols (such as Microsoft's NetBIOS Extended User Interface (NetBEUI)) that are not supported by WinSock version 1.1. Next, we tried to reimplement the `Reactor` interface using the Win32 API system call `WaitForMultipleObjects`. The goal was to maintain the original UNIX interface, but transparently supply a different implementation.

Transparent reimplementation failed to work due to fundamental differences in the proactive vs. reactive I/O semantics on Windows NT and UNIX outlined in Section 4. We initially considered circumventing these differences by asynchronously initiating a 0-sized `ReadFile` request on an overlapped I/O HANDLE. Overlapped I/O is an Win32 mechanism that supports asynchronous input and output. An overlapped event signals an application when data arrives, allowing `ReadFile` to receive the data synchronously. Unfortunately, this solution doubles the number of system calls for every input operation, creating unacceptable performance overhead. In addition, this approach does not adequately emulate the reactive output semantics provided by the UNIX event demultiplexing and I/O mechanisms.

It soon became clear that directly reusing class method interfaces, attributes, detailed designs, or algorithms was not feasible under the circumstances. Instead, we needed to elevate the level of abstraction for reuse to the level of design patterns. Regardless of the underlying OS event demultiplexing I/O semantics, the Reactor and Acceptor patterns are applicable for event-driven applications that must provide different types services that are triggered simultaneously by different types of events. Therefore, although OS platform differences precluded direct reuse of implementations or interfaces, the design knowledge we had invested in learning and documenting the Reactor and Acceptor patterns *was* reusable.

The remainder of this section describes the modifications we made to the implementations of the Reactor and Acceptor design patterns in order to port them to Windows NT.

4.4.1 Implementing the Reactor Pattern on Windows NT

Windows NT provides proactive I/O semantics that are typically used in the following manner. First, an application creates a HANDLE that corresponds to an I/O channel for the type of networking mechanism being used (such as named pipes or sockets). The overlapped I/O attribute is specified to the HANDLE creation system call (WinSock sockets are created for overlapped I/O by default). Next, an application creates a HANDLE to a Win32 event object and uses this

[4]WinSock is a Windows-oriented transport layer programming interface based on the BSD socket paradigm.

[5]Win32 is the 32-bit Windows subsystem of the Windows NT operating system.

event object HANDLE to initialize an overlapped I/O structure. The HANDLE to the I/O channel and the overlapped I/O structure are then passed to the WriteFile or ReadFile system calls to initiate a send or receive operation, respectively. The initiated operation proceeds asynchronously and does not block the caller. When the operation completes, the event object specified inside the overlapped I/O structure is set to the "signaled" state. Subsequently, Win32 demultiplexing system calls (such as WaitForSingleObject or WaitForMultipleObjects) may be used to detect the signaled state of the Win32 event object. These calls indicate when an outstanding asynchronous operation has completed.

The Win32 WaitForMultipleObjects system call is functionally similar to the UNIX select and poll system calls. It blocks on an array of HANDLEs waiting for one or more of them to signal. Unlike the two UNIX system calls (which wait only for I/O handles), WaitForMultipleObjects is a general purpose routine that may be used to wait for any type of Win32 object (such as a thread, process, synchronization object, I/O handle, named pipe, socket, or timer). It may be programmed to return to its caller either when any one of the HANDLEs becomes signaled or when all of the HANDLEs become signaled. WaitForMultipleObjects returns the index location in the HANDLE array of the lowest signaled HANDLE.

Windows NT proactive I/O has both advantages and disadvantages. One advantage over UNIX is that Windows NT WaitForMultipleObjects provides the flexibility to synchronize on a wide range Win32 objects. Another advantage is that overlapped I/O may improve performance by allowing I/O operations to execute asynchronously with respect to other computation performed by applications or the OS. In contrast, the reactive I/O semantics offered by UNIX do not support asynchronous I/O directly (threads may be used instead).

On the other hand, designing and implementing the Reactor pattern using proactive I/O on Windows NT turned out to be more difficult than using reactive I/O on UNIX. Several characteristics of WaitForMultipleObjects significantly complicated the implementation of the Windows NT version of the Reactor pattern.

First, applications that must synchronize simultaneous send and receive operations on the same I/O channel are more complicated to program on Windows NT. For example, to distinguish the completion of a WriteFile operation from a ReadFile operation, separate overlapped I/O structures and Win32 event objects must be allocated for input and output. Furthermore, two elements in the WaitForMultipleObjects HANDLE array (which is currently limited to a rather small maximum of 64 HANDLEs) are consumed by the separate event object HANDLEs dedicated to the sender and the receiver.

Second, Each Win32 WaitForMultipleObjects call only returns notification on a single HANDLE. Therefore, to achieve the same behavior as the UNIX select and poll system calls (which return a set of activated I/O handles), multiple WaitForMultipleObjects must be performed. In addition, the semantics of WaitForMultipleObjects do not result in a fair distribution of notifications. In particular, the lowest signaled HANDLE in the array is always returned, regardless of how long other HANDLEs further back in the array may have been pending.

The implementation techniques required to deal with these characteristics of Windows NT were rather complicated. Therefore, we modified the NT Reactor by creating a Handler_Repository class that shields the Reactor from this complexity. This class stores Event_Handler objects that registered with a Reactor. This container class implements standard operations for inserting, deleting, suspending, and resuming Event_Handlers. Each Reactor object contains a Handler_Repository object in its private data portion. A Handler_Repository maintains the array of HANDLEs passed to WaitForMultipleObjects and it also provides methods for inserting, retrieving, and "re-prioritizing" the HANDLE array. Re-prioritization alleviates the inherent unfairness in the way that the Windows NT WaitForMultipleObjects system call notifies applications when HANDLEs become signaled.

The Handler_Repository's re-prioritization method is invoked by specifying the index of the HANDLE which has signaled and been dispatched by the Reactor. The method's algorithm moves the signaled HANDLE toward the end of the HANDLE array. This allows signaled HANDLEs that are further back in the array to be returned by subsequent calls to WaitForMultipleObjects. Over time, HANDLEs that signal frequently migrate to the end of the HANDLE array. Likewise, HANDLES that signal infrequently migrate to the front of the HANDLE array. This algorithm ensures a reasonably fair distribution of HANDLE dispatching.

The implementation techniques described in the previous paragraph did not affect the external interface of the Reactor. Unfortunately, certain aspects of Windows NT proactive I/O semantics, coupled with the desire to fully utilize the flexibility of WaitForMultipleObjects, forced visible changes to the Reactor's external interface. In particular, Windows NT overlapped I/O operations must be initiated *immediately*. Therefore, it was necessary for the Windows NT Event_Handler interface to distinguish between I/O HANDLEs and synchronization object HANDLES, as well as to supply additional information (such as message buffers and event HANDLEs) to the Reactor. In contrast, the UNIX version of the Reactor does not require this information immediately. Therefore, it may wait until it is *possible* to perform an operation, at which point additional information may be available to help optimize program behavior.

The following modifications to the Reactor were required to support Windows NT I/O semantics. The Reactor_Mask enumeration was modified to include a new SYNC_MASK value to allow the registration of an Event_Handler that is dispatched when a general Win32 synchronization object signals. The `send` method was added to the `Reactor` class to proactively initiate output operations on behalf of an `Event_Handler`.

```
// Bit-wise "or" these values to
// check for multiple activities per-handle.
enum Reactor_Mask { READ_MASK = 01,
  WRITE_MASK = 02, SYNC_MASK = 04
};

class Reactor
{
public:
  // Same as UNIX Reactor...

  // Initiate an asynchronous send operation.
  virtual int send (Event_Handler *,
                    const Message_Block *);

// ...
};
```

Likewise, the `Event_Handler` interface for Windows NT was also modified as follows:

```
class Event_Handler
{
protected:
  // Returns the Win32 I/O HANDLE
  // associated with the derived object
  // (must be supplied by a subclass).
  virtual HANDLE get_handle (void) const;

  // Allocates a message for the Reactor.
  virtual Message_Block *get_message (void);

  // Called when event occurs.
  virtual int handle_event (Message_Block *,
                            Reactor_Mask);

  // Called when object is removed from Reactor.
  virtual int handle_close (Message_Block *,
                            Reactor_Mask);

// Same as UNIX Event_Handler...
};
```

When a derived `Event_Handler` is registered for input with the `Reactor` an overlapped input operation is immediately initiated on its behalf. This requires the `Reactor` to request the derived `Event_Handler` for an I/O mechanism HANDLE, destination buffer, and a Win32 event object HANDLE for synchronization. A derived `Event_Handler` returns the I/O mechanism HANDLE via its `get_handle` method and returns the destination buffer location and length information via the `Message_Block` abstraction described in [4].

The current implementation of the Windows NT-based Reactor pattern is about 2,600 lines C++ code (not including comments or extraneous whitespace). This code is approximately 200 lines longer than the UNIX version. The additional code primarily ensures the fairness of `WaitForMultipleObjects` event demultiplexing, as discussed above. Although Windows NT event demultiplexing is more complex than UNIX, the behavior of Win32 mutex objects eliminated the need for the separate `Mutex` interface with recursive-mutex semantics discussed in Section 4.3.1. Under Win32, a thread will not be blocked if it attempts acquire a mutex specifying the HANDLE to a mutex that it already owns. However, to release its ownership, the thread must release a Win32 mutex once for each time that the mutex was acquired.

4.4.2 Implementing the Acceptor Pattern on Windows NT

The following example C++ code illustrates an implementation of the Acceptor pattern based on the Windows NT version of the Reactor pattern.

```
template <class PEER_IO>
class Logging_Handler : public Event_Handler
{
public:
  // Callback method that handles the
  // reception of logging transmissions from
  // remote clients.  The Message_Block object
  // stores a message received from a client.

  virtual int handle_event (Message_Block *msg,
                            Reactor_Mask) {
    Log_Record *log_record =
      (Log_Record *) msg->get_rd_ptr ();

    // Format and print logging record.
    log_record.format_and_print ();
    delete msg;
    return 0;
  }

  // Retrieve the I/O HANDLE (called by Reactor
  // when a Logging_Handler object is registered).

  virtual HANDLE get_handle (void) const {
    return this->peer_io_.get_handle ();
  }

  // Return a dynamically allocated buffer
  // to store an incoming logging message.

  virtual Message_Block *get_message (void) {
    return new Message_Block (sizeof (Log_Record));
  }

  // Close down I/O handle and delete
  // object when a client closes connection.
  virtual int handle_close (Message_Block *msg,
                            Reactor_Mask) {
    delete msg;
    delete this;
    return 0;
  }

private:
  // Private ensures dynamic allocation.
  ~Logging_Handler (void) {
    this->peer_io_.close ();
  }

  // C++ wrapper for data transfer.
  PEER_IO peer_io_;
}
```

The `Logging_Acceptor` class is essentially the same as the one illustrated in Section 4.3.2. Likewise, the interaction diagram that describes the collaboration between objects in the server logging daemon is also very similar to the one shown in Figure 4.

The application is the same server logging daemon presented in Section 4.3.2. The primary difference is that Win32 `Named_Pipe` C++ wrappers are used instead of the `SOCK_SAP` socket C++ wrappers in the main program as shown below:

```
// Global per-process instance of the Reactor.
Reactor reactor;

// Server endpoint.
const char ENDPOINT[] = "logger";

// Instantiate the Logging_Handler template.
typedef Logging_Handler <NPipe_IO>
        LOGGING_HANDLER;

// Instantiate the Logging_Acceptor template.
typedef Logging_Acceptor<LOGGING_HANDLER,
                         NPipe_Acceptor,
                         Local_Pipe_Name>
        LOGGING_Acceptor;

int
main (void)
{
  // Logging server address.
  Local_Pipe_Name addr (ENDPOINT);
  // Initialize logging server endpoint.
  LOGGING_Acceptor Acceptor (addr);

  reactor.register_handler (&Acceptor,
                            SYNC_MASK);

  // Arm the proactive I/O handler.
  Acceptor.initiate ();

  // Main event loop that handles client
  // logging records and connection requests.
  reactor.dispatch ();
  /* NOTREACHED */
  return 0;
}
```

The Named Pipe Acceptor object (`Acceptor`) is registered with the Reactor to handle asynchronous connection establishment. Due to the semantics of Windows NT proactive I/O, the `Acceptor` object must explicitly initiate the acceptance of a Named Pipe connection via an `initiate` method. Each time a connection acceptance is completed, the Reactor dispatches the `handle_event` method of the Named Pipe version of the Acceptor pattern to create a new `Svc_Handler` that will receive logging records from the client. The `Reactor` will also initiate the next connection acceptance sequence asynchronously.

5 Lessons Learned

Our group at Ericsson has been developing OO frameworks based on design patterns for the past two years [6]. During this time, we have identified a number of pros and cons related to using design patterns as the basis for our system design, implementation, and documentation. We have also formulated a number of "workarounds" for the problems we observed using design patterns in a production environment. This section discusses the lessons we have learned thus far.

5.1 Pros and Cons of Design Patterns

Ironically, many pros and cons of using design patterns are "duals" of each other, representing "two sides of the same coin:"

• **Patterns are underspecified:** they generally do not over-constrain an implementation. This is beneficial since it permits flexible solutions that may be customized according to application requirements and the constraints imposed by the OS platform and network environment.

On the other hand, it is important for developers and managers to recognize that understanding a collection of design patterns is no substitute for design and implementation skills. Unfortunately, patterns often lead developers to think they know more about the solution to a recurring problem than they actually do. For example, recognizing the structure and participants in a pattern (such as the Reactor or Acceptor patterns) is only the first step. As we describe in Section 4, a major development effort is often required to fully realize the pattern correctly and efficiently.

• **Patterns enable large-scale architectural reuse:** even if reuse of algorithms, implementations, interfaces, or detailed designs is not feasible. Understanding these benefits was crucial in the design evolution we presented in Section 4. Our task became much simpler when we recognized how to leverage off our prior development effort and reduce risk by reusing the Reactor and Acceptor patterns across UNIX and Windows NT.

It is important, however, to manage the expectations of developers and managers, who may have misconceptions about the fundamental contribution of design patterns to a project. In particular, patterns do not lead to automated code reuse. Neither do they guarantee flexible and efficient design and implementation. As always, there is no substitute for creativity and diligence on the part of developers.

• **Patterns capture knowledge that is implicitly understood:** our experience has been that once developers are exposed to, and properly motivated by, the concepts of design patterns, they are generally very eager to adopt the nomenclature and methodology. Patterns tend to codify knowledge that is already understood intuitively. Therefore, once basic concepts, notations, and pattern template formats are mastered, it is straightforward to document and reason about many portions of a system's architecture and design using patterns.

The downside of the intuitive nature of patterns is a phenomenon we termed "pattern explosion." In this situation, all aspects of a project become expressed as patterns, which often leads to relabeling existing development practices without significantly improving them. We also noticed a tendency for developers to spend considerable time formalizing relatively mundane concepts (such as binary search, a linked list, or opening a file) as patterns. Although this may be intellectually satisfying, it does not necessarily improve productivity or software quality.

• **Patterns help improve communication within and across software development teams:** developers share a common vocabulary and a common conceptual "gestalt." By learning the key recurring patterns in their application domain, developers at Ericsson elevated the level of abstraction by which they communicated with their colleagues. For example, once our team understood the Reactor and Acceptor patterns, they began to use them in many other projects that benefited from these architectures.

As usual, however, restraint and a good sense of aesthetics is required to resist the temptation of elevating complex concepts and principles to the level of "buzz words" and hype. We noticed a tendency for many developers to get locked into "pattern-think," where they would try to apply patterns that were inappropriate simply because they were familiar with the patterns. For example, the Reactor pattern is often an inefficient event demultiplexing model for a multi-processor platform since it serializes application concurrency at a very coarse-grained level.

• **Patterns promote a structured means of documenting software architectures:** this documentation may be written at a high-level of abstraction, which captures the essential architectural interactions while suppressing unnecessary details.

One drawback we observed with much of the existing pattern literature [1, 2], however, is that it is often *too* abstract. Abstraction is a benefit in many cases since it avoids inundating a casual reader with excessive details. However, we found that in many cases that overly abstract pattern descriptions made it difficult for developers to understand and apply a particular pattern to systems they were building.

5.2 Solutions and Workarounds

Based on our experiences, we recommend the following solutions and workarounds to the various traps and pitfalls with patterns mentioned above.

• **Expectation management:** many of the problems with patterns we discussed above are related to managing the expectations of development team members. As usual, patterns are no silver bullet that will magically absolve managers and developers from having to wrestle with tough design and implementation issues. At Ericsson, we have worked hard to motivate the genuine benefits from patterns, without hyping them beyond their actual contribution.

• **Wide-spectrum pattern exemplars:** based on our experience using design patterns as a documentation tool, we believe that pattern catalogs should include more than just object model diagrams and structured prose. Although these notations are suitable for a high-level overview, we found in practice that they are insufficient to guide developers through difficult design and implementation tradeoffs. Therefore, it is very useful to have concrete source code examples to supplement the more abstract diagrams and text.

Hyper-text browsers, such as Mosaic and Windows Help Files, are particularly useful for creating compound documents that possess multiple levels of abstraction. Moreover, in our experience, it was particularly important to illustrate multiple implementations of a pattern. This helps to avoid "tunnel vision" and over-constrained solutions based upon a limited pattern vocabulary. The extended discussion in Section 4 is one example of a wide-spectrum exemplar using this approach. This example contains in-depth coverage of tradeoffs encountered in actual use.

• **Integrate patterns with OO frameworks:** Ideally, examples in pattern catalogs [2, 1] should reference (or better yet, contain hyper-text links to) source code that comprises an actual OO framework. We have begun building such an environment at Ericsson, in order to disseminate our patterns and frameworks to a wider audience. In addition to linking on-line documentation and source code, we have had good success with periodic design reviews where developers throughout the organization present interesting patterns they have been working on. This is another technique for avoiding "tunnel vision" and enhancing the pattern vocabulary within and across development teams.

6 Concluding Remarks

Design patterns facilitate the reuse of abstract architectures that are decoupled from concrete realizations of these architectures. This decoupling is useful when developing system software components and frameworks that are reusable across OS platforms. This article describes two design patterns, Reactor and Acceptor, that are commonly used in distributed system software. These design patterns characterize the collaboration between objects that are used to automate common activities (such as event demultiplexing, event handler dispatching, and connection establishment) performed by distributed applications. Using the design pattern techniques described in this article, we successfully reused major portions of our telecommunication system software development effort across several diverse OS platforms.

This case study describes how an OO framework based on the Reactor and Acceptor design patterns evolved from several UNIX platforms to the Windows NT Win32 platform. Due to fundamental differences between the platforms, it was not possible to directly reuse the algorithms, detailed designs, interfaces, or implementations of the framework across the different OS platforms. In particular, performance constraints and fundamental differences in the I/O mechanisms available on Windows NT and UNIX platforms prevented us from encapsulating event demultiplexing functionality within a directly reusable framework. However, we were able to reuse the underlying design patterns, which reduced project risk significantly and simplified our re-development effort.

Our experiences with patterns reinforce the observation that the transition from OO analysis to OO design and implementation is challenging [11]. Often, the constraints of the underlying OS and hardware platform influence design and implementation details significantly. This is particularly problematic for system software, which is frequently targeted for particular platforms with particular non-portable characteristics. In such circumstances, reuse of design patterns may be the only viable means to leverage previous development expertise.

The UNIX version of the ASX framework components described in this article are freely available via anonymous ftp from the Internet host `ics.uci.edu` (128.195.1.1) in the file `gnu/C++_wrappers.tar.Z`. This distribution contains complete source code, documentation, and example test drivers for the C++ components developed as part of the ADAPTIVE project [4] at the University of California, Irvine and Washington University.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.

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

[3] J. O. Coplien, "A Development Process Generative Pattern Language," in *Pattern Languages of Programs* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, June 1995.

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

[5] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[6] D. C. Schmidt and P. Stephenson, "An Object-Oriented Framework for Developing Network Server Daemons," in *Proceedings of the* 2^{nd} *C++ World Conference*, (Dallas, Texas), SIGS, Oct. 1993.

[7] 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.

[8] A. Weinand, E. Gamma, and R. Marty, "ET++ - an object-oriented application framework in C++," in *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, pp. 46–57, ACM, Sept. 1988.

[9] M. A. Linton, J. Vlissides, and P. Calder, "Composing User Interfaces with InterViews," *IEEE Computer*, vol. 22, pp. 8–22, February 1989.

[10] S. Vinoski, "Distributed Object Computing with CORBA," *C++ Report*, vol. 5, July/August 1993.

[11] G. Booch, *Object Oriented Analysis and Design with Applications (*2^{nd} *Edition)*. Redwood City, California: Benjamin/Cummings, 1993.

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

[13] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

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