Experience Using Design Patterns to Evolve Communication Software Across Diverse OS Platforms

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 this paper appeared in the proceedings of the 9^{th} European Conference on Object-Oriented Programming held in Aarhus, Denmark on August 7-11, 1995.

Abstract

Design patterns help to improve communication software quality since they address a fundamental challenge in large-scale software development: communication of architectural knowledge among developers. This paper makes several contributions to the study and practice of design patterns. It presents a case study that illustrates how design patterns helped to reduce development effort and project risk when evolving an object-oriented telecommunication software framework across UNIX and Windows NT OS platforms. Second, the paper discusses the techniques, benefits, and limitations of applying a design pattern-based reuse strategy to commercial telecommunication software systems.

1 Introduction

Developing communication software that is reusable across OS platforms is hard. Constraints imposed by the underlying OS platforms may make it impractical to reuse existing algorithms, detailed designs, interfaces, or implementations directly. This paper describes how we evolved an object-oriented framework from several UNIX platforms to the Windows NT Win32 platform. Fundamental differences in the I/O mechanisms available on Windows NT and UNIX platforms precluded the direct black-box reuse of framework components. We were, however, able to achieve significant reuse of the *design patterns* underlying the framework.

Design patterns capture the static and dynamic structures of solutions that occur repeatedly when developing applications in a particular context [1, 2, 3]. Systematically incorporating design patterns into the software development process helps improve software quality since patterns address a fundamental challenge in large-scale software development: *communication of architectural knowledge among developers*. In this paper, we describe our experience with a design pattern-based reuse strategy. We have successfully used this strategy at Ericsson to develop a family of object-oriented telecommunication system software based on the ADAPTIVE Service eXecutive (`ASX`) framework [4].

The `ASX` framework is an integrated collection of components that collaborate to produce a reusable infrastructure for developing communication software. The framework performs common communication software activities (such as event demultiplexing, event handler dispatching, connection establishment, routing, configuration of application services, and concurrency control). At Ericsson, we have used the `ASX` framework to enhance the flexibility and reuse of network management software, which monitors and manages telecommunication switches across multiple hardware and software platforms.

During the past year, we ported the `ASX` framework from several UNIX platforms to the Windows NT platform. These OS platforms possess different mechanisms for event demultiplexing and I/O. To meet our performance and functionality requirements, it was not possible to reuse several key components in the `ASX` framework directly across the OS platforms. It was possible, however, to reuse the underlying design patterns embodied by the `ASX` framework, which reduced project risk significantly and simplified our re-development effort.

The remainder of the paper is organized as follows: Section 2 presents an overview of the design patterns that are the focus of this paper; Section 3 examines the issues that arose when we ported the components in the `Reactor` framework from several UNIX platforms to the Windows NT platform; Section 4 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 5 presents concluding remarks.

2 Overview of Design Patterns

A design pattern represents a recurring solution to a design problem within a particular domain (such as business data processing, telecommunications, graphical user interfaces, databases, and distributed communication software) [1]. Design patterns facilitate architectural level reuse by providing "blueprints" that guide the definition, composition, and evaluation of key components in a software system. In general, a large amount of experience 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.

This paper focuses on two specific design patterns (the Reactor [5] and Acceptor patterns) that are implemented by the ASX framework. 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 and Siemens Q.port ATM signaling software product, the system control segment for the Motorola Iridium global personal communications system [6], a family of system/network management applications for Ericsson telecommunication switches [7], and a Global Limiting System developed by Credit Suisse that manages credit risk and market risk.

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. By using 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 these patterns onto the mechanisms provided on the OS platforms. Finding an appropriate mapping was non-trivial, as we describe in Section 3. Nevertheless, our knowledge of the design patterns significantly reduced redevelopment effort and minimized the level of risk in our projects.

2.1 The Reactor Pattern

The Reactor is an object behavioral pattern that decouples event demultiplexing and event handler dispatching from the services performed in response to events. This separation of concerns factors out the demultiplexing and dispatching mechanisms (which may be independent of an application and thus reusable) from the event handler processing policies (which are specific to an application). The Reactor pattern appears in many single-threaded event-driven frameworks (such as the Motif, Interviews [8], System V STREAMS [9], the ASX OO communication framework [4], and implementations of DCE [10] and CORBA [11]).

The Reactor pattern solves a key problem for single-threaded communication software: *how to efficiently demultiplex multiple types of events from multiple sources of events within a single thread of control.* This strategy provides coarse-grained concurrency control that serializes application event handling within a process at the event demultiplexing level. One consequence of using the Reactor pattern is that the need for more complicated threading, synchronization, or locking within an application may be eliminated.

Figure 1 illustrates the structure and participants in the Reactor pattern. The `Reactor` defines an interface for registering, removing, and dispatching `Event Handler` objects. An implementation of this interface provides a set of application-independent mechanisms. These mechanisms perform event demultiplexing and dispatching of application-specific event handlers in response to events. The `Event Handler` specifies an abstract interface used by the `Reactor` to dispatch callback methods defined by objects that register to handle input, output, signal, and timeout events of interest. The `Concrete Event Handler` selectively implements callback method(s) to process events in an application-specific manner.

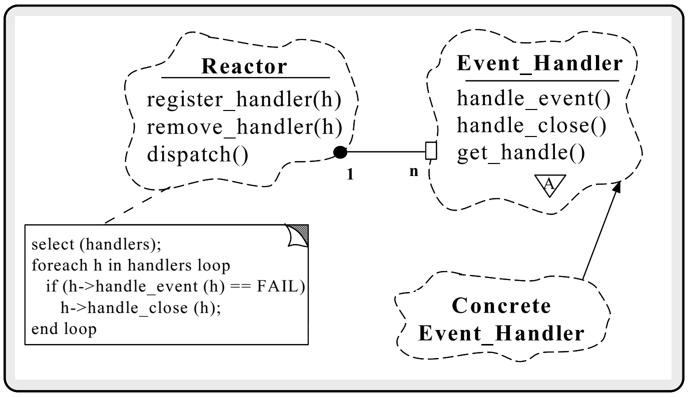

Figure 1: The Structure and Participants in the Reactor Pattern

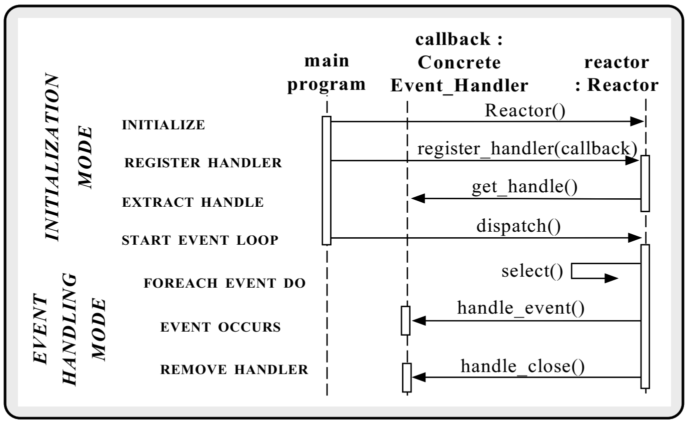

Figure 2: Object Interaction Diagram for the Reactor Pattern

Figure 2 illustrates the collaborations between participants in the Reactor pattern. These collaborations are divided into two modes:

1. *Initialization mode* – where `Concrete Event Handler` objects are registered with the `Reactor`;
2. *Event handling mode* – where methods on the objects are called back to handle particular types of events.

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

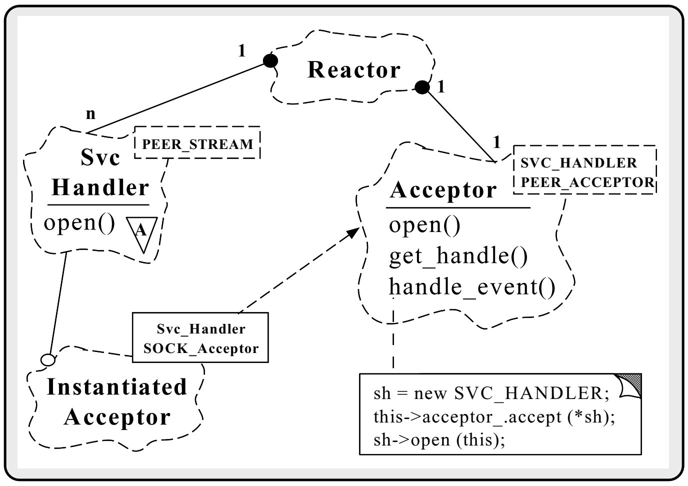

Figure 3: Structure and Participants in the Acceptor Pattern

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 is not available on many OS platforms. In these cases, the Reactor pattern may be used in lieu of, or in conjunction with, OS multi-threading or multi-processing mechanisms.

2.2 The Acceptor Pattern

The Acceptor is an object creational pattern that decouples passive connection establishment from the service performed once the connection is established. This separation of concerns enables the application-specific portion of a service to vary independently of the mechanism used to establish the connection. The Acceptor pattern appears in network "superservers" (such as `inetd` [12] and `listen` [13]). These superservers utilize a master Acceptor process that listens for connections on a set of communication ports. Each port is associated with a communication-related service (such as the standard Internet services `ftp`, `telnet`, `daytime`, and `echo`). When a service request arrives on a monitored port, the Acceptor process accepts the request and dispatches an appropriate pre-registered handler to perform the service.

The Acceptor pattern solves several problems encountered when writing communication software (particularly servers) using network programming interfaces like sockets [12] and TLI [13]:

1. How to avoid writing the same connection establishment code repeatedly for each server;
2. How to make the connection establishment code portable across platforms that may contain different network programming interfaces (such as sockets but not TLI, or vice versa);
3. How to ensure that a passive-mode I/O handle is not accidentally used to read or write data.

Figure 3 illustrates the structure and participants in the Acceptor pattern. This pattern leverages off the Reactor pattern's `Reactor` to passively establish multiple connections within a single thread of control. The `Acceptor` implements the strategy for establishing connections with peers. It is parameterized by concrete types that conform to the interfaces of the formal template arguments `SVC HANDLER` (which performs a service in conjunction with a connected peer) and `PEER ACCEPTOR` (which is the underlying IPC mechanism used to passively establish the connection). The `Svc Handler` shown in Figure 3 is a concrete type that defines the interface for an application-specific service. It inherits from `Event Handler` (shown in Figure **??**), which allows it to be dispatched by the `Reactor` when connection events occur. In addition, `Svc Handler` is parameterized by a `PEER STREAM` endpoint. The `Acceptor` associates this endpoint with its peer when a connection is established successfully.

Parameterized types are used to enhance portability since the Acceptor pattern's connection establishment strategy is independent of the type of service and the type of IPC mechanism. Programmers supply concrete arguments for these types to produce an `Instantiated Acceptor`. Note that a similar degree of decoupling could be achieved via inheritance and dynamic binding by using the Abstract Factory or Factory Method patterns described in [1]. Parameterized types were used to implement this pattern since they improve run-time efficiency at the expense of additional compile-time and link-time time and space overhead.

Figure 4 illustrates the collaboration among participants in the Acceptor pattern. These collaborations are divided into three phases:

1. *Endpoint initialization phase* – which creates a passive-mode endpoint and registers this endpoint with the `Reactor`, which then listens for connections from clients;
2. *Connection acceptance phase* – which creates a new `Svc Handler` object, accepts the connection into it, and then activates the `Svc Handler`
3. *Event handling phase* – which may employ the Reactor pattern discuss in Section 2.1 to process incoming events.

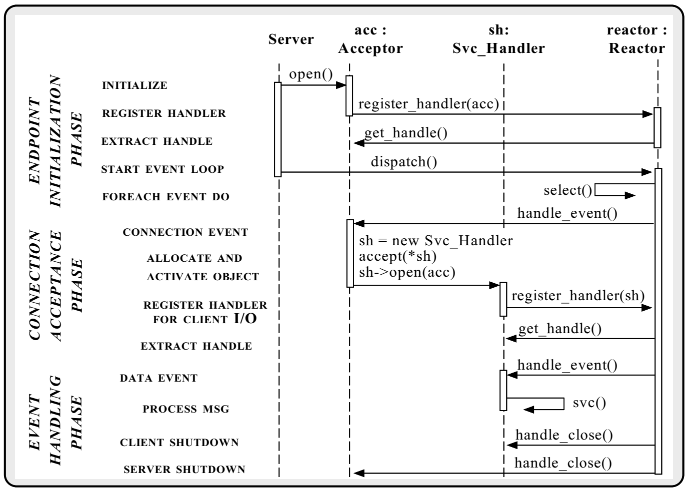

Figure 4: Object Interaction Diagram for the Acceptor Pattern

3 Evolving Design Patterns Across OS Platforms

3.1 Motivation

Our experience at Ericsson indicated that explicitly incorporating design patterns into the software development process is beneficial. Design patterns focus attention on the stable aspects of a system's software architecture. In addition, patterns emphasize the strategic collaborations between key participants in a software architecture without overwhelming developers with excessive details. Abstracting away from low-level implementation details is essential for communication software since OS platform constraints often preclude direct reuse of system components.

We observed that without concrete exemplars, however, developers at Ericsson found it hard to understand and apply patterns to components they were building. To address this concern, this section provides a technically rich, motivating, and detailed roadmap that we used at Ericsson to successfully shepard other developers into the realm of patterns. In this section, we discuss how the Reactor and Acceptor design patterns were implemented and evolved on BSD and System V UNIX platforms, as well as on Windows NT.

The discussion below outlines the relevant functional forces between the various OS platforms and describes how these differences affected the implementation of the design patterns. To focus the discussion, 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 skip ahead to Section 4. In this section we summarize the lessons we learned from using design patterns at Ericsson.

3.2 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. 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 or 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, initiating/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 actively 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. If there is more data to transfer, the program starts up another asynchronous operation. Proactive demultiplexing and I/O semantics are provided on Windows NT [14] 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 core concepts underlying the Reactor and Acceptor design patterns.

3.3 UNIX Evolution of the Patterns

3.3.1 Implementing the Reactor Pattern on UNIX

The standard demultiplexing mechanisms on UNIX operating systems provide reactive I/O semantics. 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 may 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 respect to 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 object-oriented GUI framework [8]. The `Dispatcher` is an object-oriented 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.[1]

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].

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 the portability of networking applications 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" to check for multiple
// activities per-handle.
enum Reactor_Mask { READ_MASK = 01,
  WRITE_MASK = 02, EXCEPT_MASK = 04 };

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

  // Remove handler associated with
  // the 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 [15]. 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 [15]. 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 became clear 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.
```

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

```
    foreach handler : h in this->handler_set_
      if (h->handle_event (handler, mask) == FAIL)
        // Cleanup on failure.
        h->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 acquires 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 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 platforms.

3.3.2 Implementing the Acceptor Pattern on UNIX

To illustrate the Reactor and Acceptor patterns, consider the event-driven server logging daemon for a distributed logging service shown in Figure 5. 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

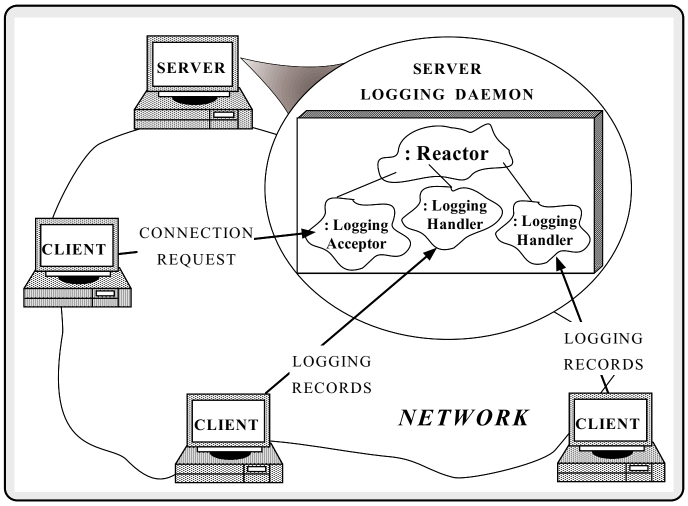

Figure 5: The Distributed Logging Service

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 5 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.[2]

The logging server listens on one I/O handle for new connection requests to arrive from 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 will delay the response time for clients connected 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` 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 connec-

[2] 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: the descriptor/handle is an opaque reference to a resource controlled by the OS.

tion request from a client, dynamically allocating a new `Logging Handler` to handle logging records from this client, and registering the new handler with an instance of the `Reactor`.

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 client. 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 STREAM`. The `PEER STREAM` class provides reliable TCP capabilities that transfer logging records between an application and the server. The use of templates avoids dependencies on a particular IPC interface (such as BSD sockets or System V TLI).

```
template <class PEER_STREAM>
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_stream_.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_stream_.recv (&log_record, len);

      // 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_stream_.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_stream_.close ();
  }

   // C++ wrapper for data transfer.
   PEER_STREAM peer_stream_;
};
```

The `Logging Acceptor` template class is shown in the C++ code below. It is a factory that performs the steps necessary to (1) accept connection requests from client logging daemons and (2) create `SVC HANDLER` objects that perform an application-specific service on behalf of clients. In this particular implementation of the Acceptor pattern the `Logging Acceptor` object and the `Logging 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 the `accept` method of the `SOCK Acceptor`, 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 client connections when they arrive on that port:

```
// 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 Logging_Acceptor endpoint.

  Logging_Acceptor (PEER_ADDR &a)
    : peer_acceptor_ (a) {}

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

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

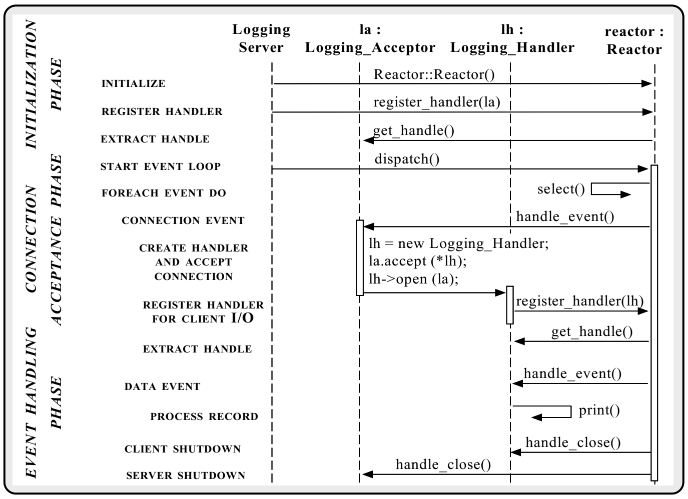

Figure 6: Server Logging Daemon Interaction Diagram

```
  // 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 6 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 [16]. `SOCK SAP` encapsulates the `SOCK STREAM` semantics of the socket transport layer interface within a type-secure, object-oriented 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 straightforward to instantiate a different network programming interface (such as the `TLI SAP` C++ wrappers that encapsulate the System V UNIX TLI interface) [17]. 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].

3.4 Evolving the Design Patterns to Windows NT

This section describes the Windows NT implementation of the Reactor and Acceptor design patterns performed at Ericsson in Cypress, California. Initially, we attempted to evolve the `Reactor` implementation described in Section 3.3.1 from UNIX to Windows NT using the `select` function in the Windows Sockets (WinSock) library.[3] This approach failed because the WinSock version of `select` does not interoperate with standard Win32[4] 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

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

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

the original UNIX interface, but transparently supply a different implementation.

Transparent reimplementation failed due to fundamental differences in the proactive vs. reactive I/O semantics on Windows NT and UNIX outlined in Section 3. 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 doubled the number of system calls for every input operation, creating unacceptable performance overhead. In addition, this approach did not adequately emulate the reactive output semantics provided by the UNIX event demultiplexing and I/O mechanisms.

It became clear that directly reusing class method interfaces, attributes, detailed designs, or algorithms was not feasible. 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 communication software 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.

3.4.1 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 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:

1. 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.
2. 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" 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 several hundred 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 3.3.1. 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.

3.4.2 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_STREAM>
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_stream_.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_stream_.close ();
  }

  // C++ wrapper for data transfer.
  PEER_STREAM peer_stream_;
}
```

The Logging Acceptor class is essentially the same as the one illustrated earlier. Likewise, the interaction diagram that describes the collaboration between objects in the server logging daemon is similar to the one shown in Figure 6. In addition, the application is the same server logging daemon presented above. As shown below, the primary difference is that Win32 Named Pipe C++ wrappers are used in the main program instead of the SOCK SAP socket C++ wrappers:

```
// 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.

4 Lessons Learned

Our group at Ericsson has been developing object-oriented frameworks based on design patterns for the past two years [7]. During this time, we have learned many lessons, both positive and negative, about using design patterns as the basis for our system design, implementation, and documentation. This section discusses the lessons we have learned and outlines workarounds for problems we encountered when using design patterns in a production software environment.

4.1 Pros and Cons of Design Patterns

Many of our experiences with patterns at Ericsson are similar to those observed on other projects using design patterns, such as the Motorola Iridium project [6]. Recognizing these common themes across different companies increased our confidence that our experiences with patterns generalize to other large-scale software projects. Note that many pros and cons of using design patterns are duals of each other, representing "two sides of the same coin."

• **Patterns are underspecified:** since they generally do not overconstrain implementations. This is beneficial since patterns permit flexible solutions that are customizable to account for application requirements and constraints imposed by the OS platform and network infrastructure.

On the other hand, developers and managers must recognize that learning a collection of patterns is no substitute for design and implementation skills. In fact, patterns often lead team members to think they know more about the solution to a 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 3, a major development effort may be required to fully realize the patterns correctly, efficiently, and portably.

• **Patterns enable widespread reuse of software architecture:** even if reuse of algorithms, implementations, interfaces, or detailed designs is not feasible. Recognizing the benefit of architectural reuse (which is inherently indirect), was crucial in the design evolution we presented in Section 3. Our task became much simpler after 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, at this point in time, software tools do not exist to automatically transform patterns into code. 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:** we found that once developers are exposed to, and properly motivated by, design patterns they are eager to adopt the pattern nomenclature and concepts. This stems from the fact that patterns 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.

A drawback to the intuitive nature of patterns is a phenomenon we termed *pattern overload.* In this situation, so many aspects of the project are expressed as patterns that the concept becomes diluted. This situation occurs when existing development practices are relabled as patterns without significantly improving them. Likewise, developers may spend their time recasting mundane concepts (such as binary search or building a linked list) into pattern form. Although this is intellectually satisfying, it becomes counter productive if it does not lead to software quality improvements.

• **Patterns help improve communication within and across software development teams:** since developers share a common vocabulary and a common conceptual gestalt. By learning the key recurring patterns in the communication software domain, developers at Ericsson elevated the level of discourse they used to communicate with colleagues. For example, once our team understood the Reactor and Acceptor patterns, they used them in other projects that were suited to these architectures.

The focus on design patterns has also helped us to move away from "programming language-centric" views of the object paradigm. This has been beneficial at Ericsson since it enabled experienced developers from different language communities (such as Lisp, Smalltalk, C++, C, and Erlang) to share design expertise of mutual interest.

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 hyperbole. We noticed a tendency for some developers to adopt a tunnel vision where they would try to apply patterns that were inappropriate simply because they were familiar with the patterns. For example, the Reactor pattern may be an inefficient event demultiplexing model for a multi-processor platform since it serializes application concurrency at a fairly coarse-grained level.

• **Patterns promote a structured means of documenting software architectures:** by capturing capturing the structure and collaboration of participants in a software architecture at a higher level than source code. This abstraction process is beneficial since it captures the essential architectural interactions while suppressing unnecessary details.

One of our concerns with conventional pattern catalogs [1, 2], however, is that they are too abstract. 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.

4.2 Solutions and Workarounds

Based on our experiences, we recommend the following solutions and workarounds to the various traps and pitfalls with patterns discussed 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 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. Hyper-text browsers, such as HTML 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 3 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 object-oriented frameworks:** Ideally, examples in pattern catalogs [2, 1] should reference (or better yet, contain hyper-text links to) source code that comprises an actual object-oriented framework that illustrate the patterns. 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 useful patterns they have been working on. This is another technique for avoiding tunnel vision and enhancing the pattern vocabulary within and across development teams.

5 Concluding Remarks

Design patterns facilitate the reuse of abstract architectures that are decoupled from concrete realizations of these architectures. This decoupling is particularly useful when developing communication software components and frameworks that are reusable across OS platforms. This paper describes two design patterns, Reactor and Acceptor, that are commonly used to build communication software. Using the design pattern techniques described in this paper, we successfully reused major portions of our telecommunication system software development effort and experience across diverse UNIX and Windows NT OS platforms.

Our experiences with patterns reinforce the fact that the transition from object-oriented analysis to object-oriented design and implementation is challenging. Often, the constraints of the underlying OS and hardware platform influence design and implementation details significantly. This is particularly problematic for communication software, which is frequently targeted for OS platforms that contain non-portable features. 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 paper are freely available via anonymous ftp from the Internet host `ics.uci.edu` (128.195.1.1) in the files `gnu/C++_wrappers.tar.Z` and `gnu/C++_wrappers_doc.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, and M. Stal, *Pattern-Oriented Software Architecture - A Pattern System*. Wileys and Sons, 1995.

[3] J. O. Coplien and D. C. Schmidt, eds., *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 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, "Experience with a System of Reusable Design Patterns for Motorola Iridium Communication Software," in *Submitted to OOPSLA '95*, (Austin, Texas), ACM, October 1995.

[7] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

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

[9] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[10] J. Dilley, "OODCE: A C++ Framework for the OSF Distributed Computing Environment," in *Proceedings of the Winter Usenix Conference*, USENIX Association, January 1994.

[11] C. Horn, "The Orbix Architecture," tech. rep., IONA Technologies, August 1993.

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

[13] S. Rago, *UNIX System V Network Programming*. Reading, MA: Addison-Wesley, 1993.

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

[15] 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.

[16] D. C. Schmidt and T. Harrison, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the Conference on Object-Oriented Technologies*, (Monterey, CA), USENIX, June 1995.

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