Reactor

An Object Behavioral Pattern for
Concurrent Event Demultiplexing
and Event Handler Dispatching

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

An earlier version of this paper appeared as a chapter in the book "Pattern Languages of Program Design" ISBN 0-201-6073-4, edited by Jim Coplien and Douglas C. Schmidt and published by Addison-Wesley, 1995.

1 Intent

Support the demultiplexing and dispatching of multiple *event handlers*, which are triggered concurrently by multiple events. The Reactor pattern simplifies event-driven applications by integrating the demultiplexing of events and the dispatching of the corresponding event handlers.

2 Also Known As

Dispatcher

3 Motivation

To illustrate the Reactor pattern, consider the event-driven server for a distributed logging service shown in Figure 1. 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 1 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.[1]

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.

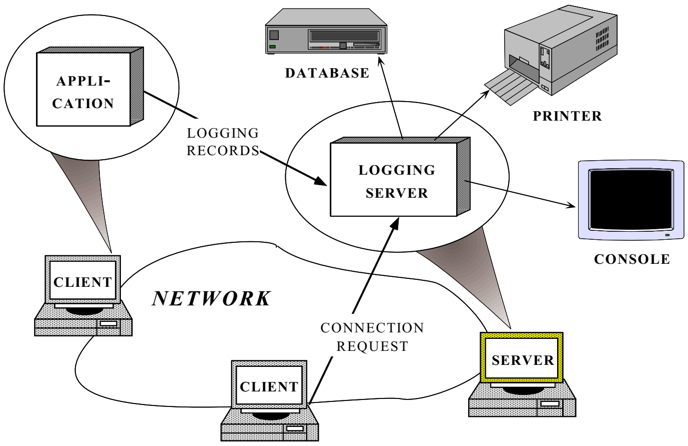


Figure 1: Distributed Logging Service

One way to develop a logging server is to use multi-threaded Active Objects [1]. In this approach, the server creates an active object for every connected client. Each active object contains a thread that blocks on a `read` system call. A thread unblocks when it receives a logging message from its associated client. At this point, the logging record is processed within the thread. The thread then re-blocks awaiting subsequent input from `read`. The key participants in the thread-based logging server are illustrated in Figure 2.

Using multi-threaded active objects to implement event handling in the logging server has several drawbacks:

- it may require complex concurrency control schemes;
- it may lead to poor performance on uni-processors [2];
- it may not be available on an OS platform.

Often, a more convenient and portable way to develop a logging server is to use the *Reactor pattern*. The Reactor pattern is useful for managing a single-threaded event loop that performs concurrent event demultiplexing and event handler dispatching in response to events.

[1] 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 basically the same.

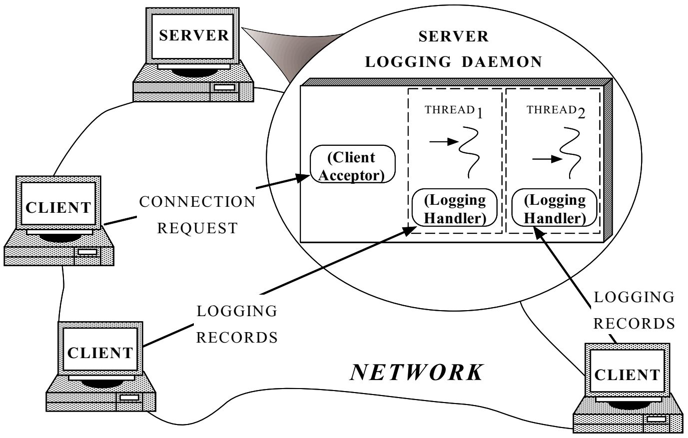


Figure 2: Multi-thread-based Logging Server

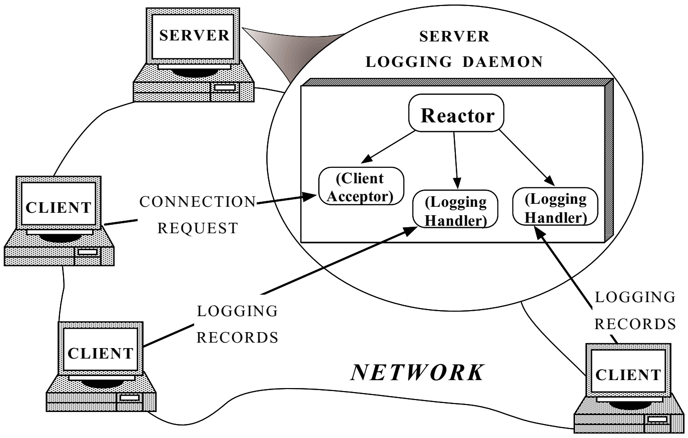
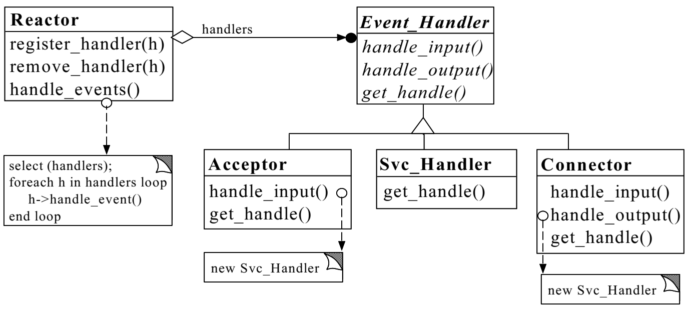


Figure 3: Reactor-based Logging Server

The Reactor pattern provides several major benefits for event-driven applications. It facilitates the development of flexible applications developed using reusable components. In particular, the pattern helps to decouple application-independent mechanisms from application-specific functionality. The application-independent mechanisms are reusable components that demultiplex events and dispatch pre-registered event handlers. The application-specific functionality is performed by user-defined methods in the event handlers.

In addition, the Reactor pattern facilitates application extensibility. For example, application-specific event handlers may evolve independently of the event demultiplexing mechanisms provided by the underlying OS platform. By using the Reactor pattern, developers are able to concentrate on application-specific functionality. In turn, the lower-level event demultiplexing and handler dispatching mechanisms are performed automatically by the Reactor.

The following figure uses OMT notation [3] to illustrate the structure of a logging server designed according to the Reactor pattern:



The `Event Handler` base class provides a standard interface for dispatching event handlers. The `Reactor` uses this interface to callback application-specific methods automatically when certain types of events occur. There are two subclasses of the `Event Handler` base class in the logging server: `Logging Handler` and `Client Acceptor`. Objects of these subclasses handle events arriving on client I/O handles. The `Logging Handler` event handler is responsible for receiving and processing logging records. The `Client Acceptor` event handler is a factory object. It accepts a new connection request from a client, dynamically allocates a new `Logging Handler` object to process logging records from this client, and registers the new `Logging Handler` event handler with the `Reactor` object.

Figure 3 illustrates a run-time view of the key participants in a Reactor-based logging server. Note that the `Client Acceptor` and the `Logging Handler` event handlers shown in this figure all execute within a single thread of control. The `Reactor` manages the event loop within this thread.

4 Applicability

Use the Reactor pattern when:

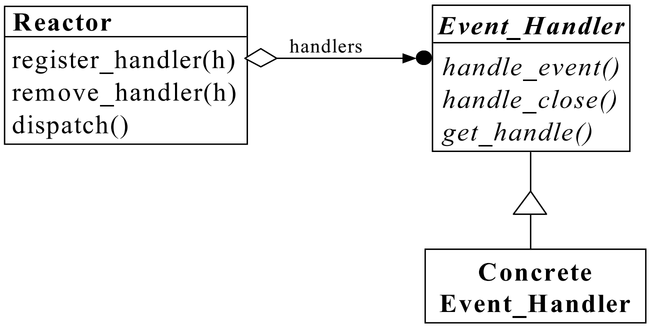
- one or more events may arrive concurrently from different sources, and blocking or continuously polling for incoming events on any individual source of events is inefficient;
- an event handler possesses the following characteristics:
  - it exchanges fixed-sized or bounded-sized messages with its peers *without* requiring blocking I/O;
  - it processes each message it receives within a relatively short period of time;
- using multi-threading to implement event demultiplexing is either:
  - *infeasible* – due to lack of multi-threading support on an OS platform;
  - *undesirable* – due to poor performance on uniprocessors or due to the need for overly complex concurrency control schemes;

– *redundant* – due to the use of multi-threading at a higher level within an application's architecture;[2]

- the functionality, portability, and extensibility of application-specific event handlers will benefit by being decoupled from the application-independent mechanisms that perform event demultiplexing and event handler dispatching.

5 Structure

The structure of the Reactor pattern is illustrated in the following OMT class diagram:



6 Participants

The key participants in the Reactor pattern include the following:

- **Reactor** (`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.
- **Event Handler** (`Event Handler`)
  - Specifies an interface used by the `Reactor` to dispatch callback methods defined by objects that are pre-registered to handle certain events.
- **Concrete Event Handler** (`Client Acceptor`, `Logging Handler`)
  - Implements the callback method(s) that process events in an application-specific manner.

[2] For example, the `handle_event` method of an `Event Handler` may spawn a separate thread and then handle one or more incoming events within this thread.

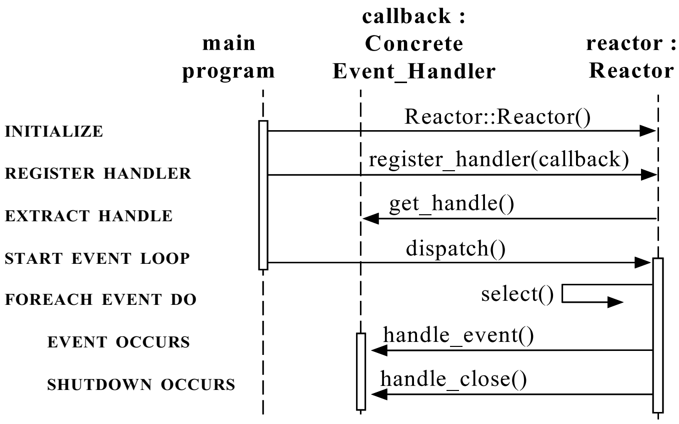


Figure 4: Interaction Diagram for the Reactor Pattern

7 Collaborations

- The `Reactor` triggers `Event Handler` methods in response to events. These events are 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 its `get_handle` method. When the `Reactor` registers an `Event Handler` subclass object, it obtains the object's handle by invoking the `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).
- When events occur, the `Reactor` uses the handles activated by the events as keys to locate and dispatch the appropriate `Event Handler` methods. This collaboration is structured using method callbacks and is depicted in Figure 4. The `handle_event` method is called by the `Reactor` to perform application-specific functionality in response to an event. Likewise, the `Reactor` invokes the `handle_close` method to perform application-specific cleanup operations before it removes an `Event Handler` subclass object.

8 Consequences

The Reactor pattern offers the following benefits:

- It decouples application-independent mechanisms from application-specific functionality. The application-independent mechanisms become reusable components. They know how to demultiplex events and dispatch the appropriate callback methods defined by `Event Handlers`. In contrast, the application-specific functionality knows how to perform a particular type of service.

- It helps to improve the modularity, reusability, and configurability of event-driven application software. For example, the pattern decouples the functionality in the logging server into two separate classes: one for establishing connections and another for receiving and processing logging records. This decoupling enables the reuse of the connection establishment class for different types of connection-oriented services (such as file transfer, remote login, and video-on-demand). Therefore, to modify or extend the functionality of the logging server, only the implementation of the logging class must change.
- It improves application portability by allowing its interface to be reused independently of the underlying OS system calls that perform event demultiplexing. These system calls detect and report the occurrence of one or more events that may occur simultaneously on multiple sources of events. Sources of events may include I/O ports, timers, synchronization objects, signals, etc. On UNIX platforms, the event demultiplexing system calls are called `select` and `poll` [4]. In the Windows NT WIN32 API, the `WaitForMultipleObjects` system call performs event demultiplexing [5].
- It provides applications with a 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 process or thread. Often, this eliminates the need for more complicated synchronization or locking within an application process.

The Reactor pattern has the following drawbacks:

- Its event handlers are not preempted while they are executing. Therefore, event handlers generally should not perform blocking I/O. Neither should they perform long-duration operations (such as bulk data transfer of a multi-megabyte file) on an individual I/O handle. Otherwise, the responsiveness of services offered on other I/O handles will decrease significantly.

  To perform long-duration services, it may be necessary to spawn a separate process or thread. This separate process or thread will complete its tasks in parallel with the `Reactor`'s main event-loop.
- The Reactor pattern can be difficult to debug since its flow of control oscillates between the lower-level demultiplexing code and the higher-level method callbacks on application-specific event handlers. This increases the difficulty of "single-stepping" through the run-time behavior of a `Reactor` (and its registered `Event Handlers`) within a debugger since the application writer may have no knowledge of how to understand the lower-level demultiplexing code. This is similar to the problems encountered trying to debug a compiler lexical analyser and parser written with LEX and YACC.
- On certain OS platforms, it may be necessary to allocate more than one `Reactor` object. For example, both UNIX and Windows NT restrict the number of I/O handles and events that may be waited for by a single system call (such as `select` or `WaitForMultipleObjects`). This limitation may be overcome by allocating separate processes or separate threads, each of which runs its own `Reactor` event loop.

9 Implementation

The Reactor pattern may be implemented in many ways. This section discusses several topics related to implementing the Reactor pattern.

- *Event demultiplexing* – A `Reactor` maintains a table of objects that are derived from the `Event Handler` base class. Public methods in the `Reactor`'s interface register and remove these objects from this table at run-time. The `Reactor` also provides a means to dispatch the `handle_event` method on an `Event Handler` object in response to events it has registered for.

  The `Reactor`'s dispatching mechanism is typically used as the main event loop of an event-driven application. Its `dispatch` method may be implemented using an OS event demultiplexing system call (such as `select`, `poll`, or `WaitForMultipleObjects`).

  The `Reactor`'s `dispatch` method blocks on the OS event demultiplexing system call until one or more events occur. When events occur, the `Reactor` returns from the event demultiplexing system call. It then dispatches the `handle_event` method on any `Event Handler` object(s) that are registered to handle these events. This callback method executes user-defined code and returns control to the `Reactor` when it completes.
- *Synchronization* – The `Reactor` may be used in a multi-threaded application. In this case, critical sections within the `Reactor` must be serialized to prevent race conditions when modifying or activating shared variables (such as the table holding the `Event Handler` subclass objects). A common technique for preventing race conditions involves mutual exclusion mechanisms such as semaphores or mutex variables [6].

  To prevent deadlock, mutual exclusion mechanisms should use *recursive locks*. Recursive locks are an efficient mechanism for preventing deadlock on locks that are held by the same thread across `Event Handler` method callbacks within the `Reactor`. A recursive lock may be re-acquired by the thread that owns the lock *without* blocking the thread. This property is important since the Reactor's `dispatch` method performs callbacks on application-specific `Event Handler` objects. Application callback code may subsequently re-enter the `Reactor` object using its `register_handler` and `remove_handler` methods.

- *I/O semantics* – The I/O semantics of the underlying OS significantly affect the implementation of the Reactor pattern. The standard I/O mechanisms on UNIX systems provide "reactive" semantics [4]. For example, the `select` and `poll` system calls indicate the subset of I/O handles that may be read from or written to synchronously without blocking.

  Implementing the Reactor pattern using reactive I/O is straightforward. In UNIX, `select` or `poll` indicate which handle(s) have become ready for I/O. The `Reactor` object then "reacts" by invoking the `Event Handler` `handle_event` callback method, which performs the I/O operation and the associated application-specific processing.

  In contrast, Windows NT provides "proactive" I/O semantics [5]. Proactive I/O operations proceed asynchronously and do not cause the caller to block. An application may subsequently use the WIN32 `WaitForMultipleObjects` system call to determine when its outstanding asynchronous I/O operations have completed.

  Variations in the I/O semantics of different operating systems may cause the class interfaces and class implementations of the Reactor pattern to vary across platforms. [7] describes the interfaces and implementations of several versions of the Reactor pattern that were ported from BSD and System V UNIX to Windows NT. This porting experience indicates that implementing the Reactor pattern using proactive I/O is more complicated than using reactive I/O. Unlike the reactive I/O scenario described above, proactive I/O operations must be invoked *immediately* by the `Reactor`, rather than waiting until it becomes *possible* to perform an operation. Therefore, additional information (such as a data buffer or an I/O handle) must be supplied to the `Reactor` by the `Event Handler` *before* an I/O system call is invoked. This reduces the `Reactor`'s flexibility and efficiency somewhat since the size of the data buffer must be specified in advance.

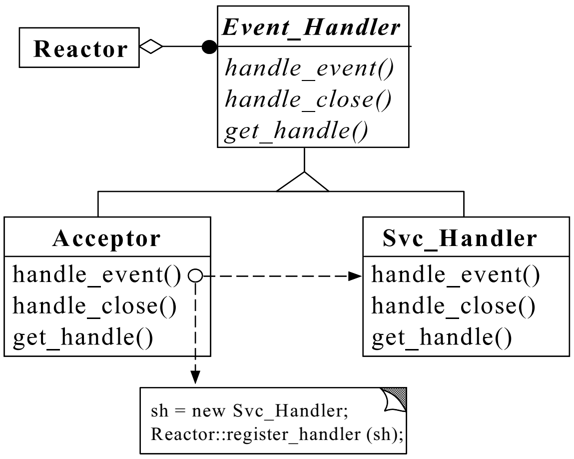
- *Event handler flexibility and extensibility* – It is possible to develop highly extensible event handlers that may be configured into a `Reactor` at installation-time or at run-time. This enables applications to be updated and extended without modifying, recompiling, relinking, or restarting the applications at run-time [8]. Achieving this degree of flexibility and extensibility requires the use of object-oriented language features (such as templates, inheritance, and dynamic binding [9]), object-oriented design techniques (such as the Factory Method or Abstract Factory design patterns [10]), and advanced operating system mechanisms (such as explicit dynamic linking and multi-threading [2]).

Regardless of the underlying OS I/O semantics, the Reactor pattern is applicable for event-driven applications that must process multiple event handlers triggered concurrently by various types of synchronous or asynchronous events. Although differences between the I/O semantics of OS platforms may preclude direct reuse of implementations or interfaces, the Reactor pattern may still be reused. In general, the ability to reuse an abstract architecture independently from its concrete realization is an important contribution of the design pattern paradigm.

10 Sample Code

The following code illustrates an example of the Reactor object behavioral pattern. The example implements portions of the logging server described in Section 3. This example also illustrates the use of an object creational pattern called the Acceptor [11]. The Acceptor pattern decouples the act of establishing a connection passively from the service(s) provided once a connection is established. This pattern enables the application-specific portion of a service to vary independently of the mechanism used to establish the connection. The Acceptor pattern is useful for simplifying the development of connection-oriented network services (such as file transfer, remote login, distributed logging, and video-on-demand applications).

The structure of the Acceptor pattern is illustrated in the following OMT class diagram:



In the example below, the general `Client Handler` class in the OMT class diagram shown above is represented by the `Logging Handler` class.

The C++ code illustrated in this section implements the `Client Acceptor` and `Logging Handler` classes, as well as the main driver function for the logging server. This implementation of the `Reactor` runs on UNIX platforms. A functionally equivalent version of this code that illustrates the Windows NT `Reactor` interface appears in [7].

10.1 The Client Acceptor Class

The Client Acceptor class establishes connections with client applications. In addition, it provides a factory for creating Logging Handler objects, which receive and process logging records from clients. The Logging Handler class is described in Section 10.2 below.

The Client Acceptor class inherits from the Event Handler base class. This enables a Client Acceptor to be registered with a Reactor object. When a client connection request arrives, the Reactor dispatches the Client Acceptor's handle_event method. This method invokes the accept method of the SOCK Acceptor object, which establishes the new connection.

The SOCK Acceptor object is a concrete factory object that enables the Client Acceptor to listen and accept connection requests on a communication port. When a connection arrives from a client, the SOCK Acceptor accepts the connection and produces a SOCK Stream object. Henceforth, the SOCK Stream object is used to transfer data reliably between the client and the logging server.

```
// Global per-process Reactor object.
extern Reactor reactor;

// Handles connection requests from clients.

class Client_Acceptor : public Event_Handler
{
public:

  // Initialize the acceptor endpoint.
  Client_Acceptor (const INET_Addr &addr)
    : acceptor_ (addr) { }

  // Callback method that accepts a new
  // SOCK_Stream connection, creates a
  // Logging_Handler object to handle logging
  // records sent using the connection, and
  // registers the object with the Reactor.

  virtual void handle_event (void)
  {
    SOCK_Stream new_connection;

    this->acceptor_.accept (new_connection);

    Logging_Handler *cli_handler =
      new Logging_Handler (new_connection);
    reactor.register_handler (cli_handler);
  }

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

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

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

  virtual void handle_close (void)
  {
    this->acceptor_.close ();
  }
private:
  // Factory that accepts client connections.
  SOCK_Acceptor acceptor_;
};
```

The SOCK Acceptor and SOCK Stream classes used to implement the logging server are part of the SOCK SAP C++ wrapper library for BSD and Windows sockets [12]. SOCK SAP encapsulates the SOCK STREAM semantics of the socket interface within a portable and type-secure object-oriented interface. In the Internet domain, SOCK STREAM sockets are implemented using the TCP transport protocol [4]. TCP provides a reliable, bi-directional bytestream-oriented transport service between two user processes.

10.2 The Logging Handler Class

The logging server uses the Logging Handler class shown below to receive logging records sent from client applications. In the code, the Logging Handler class inherits from Event Handler. This enables a Logging Handler object to be registered with the Reactor. When a logging record arrives, the Reactor automatically dispatches the handle_event method of the associated Logging Handler object. This object then receives and processes the logging record.

```
// Receive and process logging records
// sent by a client application.

class Logging_Handler : public Event_Handler
{
public:

  // Initialize the client stream.
  Logging_Handler (SOCK_Stream &cs)
    : client_stream_ (cs) {}

  // Callback method that handles the
  // reception of logging records
  // from client applications.

  virtual void handle_event (void)
  {
    Log_Record log_record;

    this->client_stream_.recv (log_record);
    // Print logging record to output.
    log_record.print ();
  }

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

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

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

  virtual void handle_close (void)
  {
```

```
    this->client_stream_.close ();
  }

private:
  // Receives logging records from a client.
  SOCK_Stream client_stream_;
};
```

The Client Acceptor and Logging Handler code shown above "hard-code" the interprocess communication (IPC) classes (*i.e.,* SOCK Acceptor and SOCK Stream, respectively) used to communicate between clients and the logging server. To remove the reliance on the particular class of IPC mechanisms, this example could be generalized to use the Abstract Factory or Factory Method patterns described in [10].

10.3 The Logging Server Main Function

The following code illustrates the main entry point into the logging server. This code creates a global Reactor object and a local Client Acceptor object. The Client Acceptor object is initialized with the network address and port number of the logging server. The program then registers the Client Acceptor object with the Reactor and enters the Reactor's main event-loop. There, the Reactor uses the select or poll OS event demultiplexing system call to block awaiting connection requests and logging records to arrive from clients.

```
// Global per-process Reactor object.
Reactor reactor;

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

int
main (void)
{
  // Logging server address and port number.
  INET_Addr addr (PORT);

  // Initialize logging server endpoint.
  Client_Acceptor ca (addr);

  // Register logging server object with
  // Reactor to set up the callback scheme.
  reactor.register_handler (&ca, READMASK);

  // Main event loop that handles client
  // logging records and connection requests.
  reactor.dispatch ();

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

The interaction diagram in Figure 5 illustrates the collaboration between the objects participating in the logging server example. Once the Reactor object is initialized, it becomes the primary focus of the control flow within the logging server. All subsequent activity is triggered by callback methods on the Client Acceptor and Logging Handler objects registered with, and controlled by, the Reactor.

11 Known Uses

The Reactor pattern has been used in a number of object-oriented frameworks. It is provided by the InterViews window system distribution [13] as the Dispatcher class category. The Dispatcher is used to define an application's main event loop and to manage connections to one or more physical GUI displays. The ADAPTIVE Service eXecutive (ASX) framework [2] uses the Reactor pattern as the central event demultiplexer/dispatcher in an object-oriented toolkit for experimenting with high-performance parallel communication protocol stacks. A freely available subset of the ASX framework described in this paper may be obtained via anonymous ftp from ics.uci.edu in the files gnu/C++_wrappers.tar.Z and gnu/C++_wrappers_doc.tar.Z.

The Acceptor pattern is used in the inetd and listen superservers. Inetd and listen are multi-service daemon management frameworks that utilize a master dispatcher process to monitor 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 dispatcher process demultiplexes the request to the appropriate pre-registered service handler. This handler performs the service and returns any results to the client requestor.

The Reactor and Acceptor patterns have been used in a number of commercial products. These products include the Bellcore Q.port ATM signaling software product, the Ericsson EOS family of telecommunication switch monitoring applications, and the network management portion of the Motorola Iridium global personal communications system.

12 Related Patterns

The Reactor pattern may be viewed as a variation on the Observer pattern [10]. In the Observer pattern, dependents are updated automatically when a subject changes. In the Reactor pattern, handlers are informed automatically when events occur. In general, the Reactor pattern is used to demultiplex multiple sources of input to their associated event handlers. In contrast, the Observer is usually associated with only a single source of events.

The Active Object pattern decouples method execution from method invocation in order to simplify synchronized access to a shared resource by methods invoked in different threads of control. The Reactor pattern is often used in place of the Active Object pattern [14] when threads are not available or when the overhead and complexity of synchronization is undesirable.

A Reactor provides a Facade [10] for event demultiplexing. A Facade is an interface that shields applications from complex object relationships within a subsystem.

The virtual methods provided by the Event Handler base class are Template Methods [10]. These template methods are used by the Reactor to trigger callbacks to the

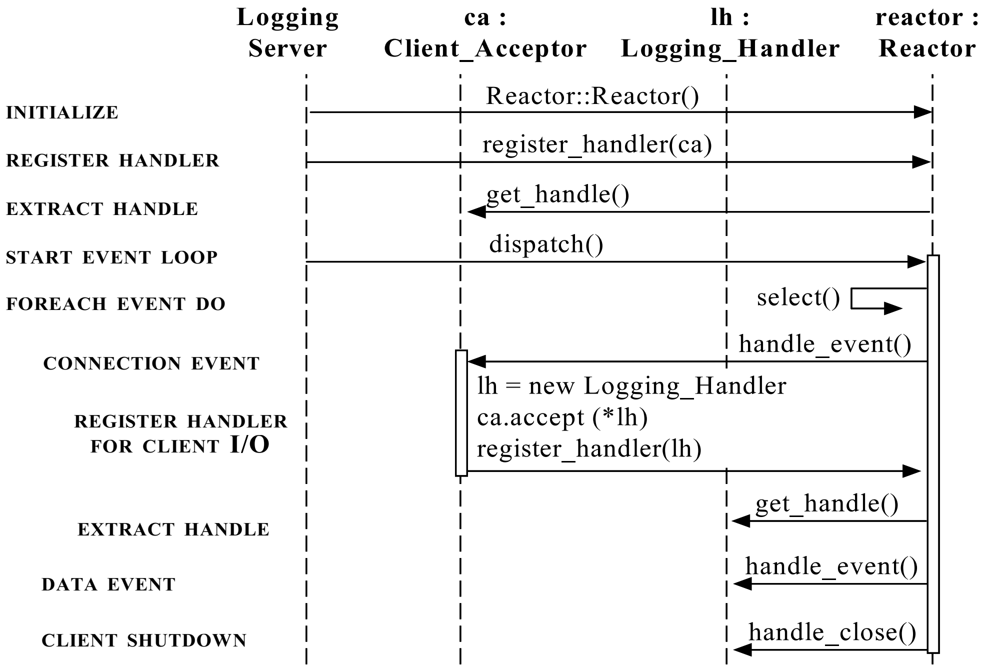


Figure 5: Interaction Diagram for Logging Server

appropriate application-specific processing functions in response to events.

An `Event Handler` (such as the `Logging Handler` class described in Section 10) may be created using a Factory Method [10]. This allows an application to decide which type of `Event Handler` subclass to create.

A Reactor may be implemented as a Singleton [10]. This is useful for centralizing event demultiplexing and dispatching into a single location within an application.

A Reactor may be used as the basis for demultiplexing messages and events that flow through a "pipes and filters" architecture [15].

The `Client Acceptor` is essentially an Abstract Factory [10] that produces concrete product objects (`Logging Handler`.

References

[1] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching," in *Proceedings of the 1^{st} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–10, August 1994.

[2] 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.

[3] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.

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

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

[6] A. D. Birrell, "An Introduction to Programming with Threads," Tech. Rep. SRC-035, Digital Equipment Corporation, January 1989.

[7] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the 9^{th} European Conference on Object-Oriented Programming*, (Aarhus, Denmark), August 1995.

[8] 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.

[9] Bjarne Stroustrup, *The C++ Programming Language, 2^{nd} Edition*. Addison-Wesley, 1991.

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

[11] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Active and Passive Establishment of Network Connections," in *Workshop on Pattern Languages of Object-Oriented Programs at ECOOP '95*, (Aarhus, Denmark), August 1995.

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

[13] M. A. Linton and P. R. Calder, "The Design and Implementation of InterViews," in *Proceedings of the USENIX C++ Workshop*, November 1987.

[14] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Submitted to the proceedings of the 2^{nd} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.

[15] R. Meunier, "The Pipes and Filters Architecture," in *Proceedings of the 1^{st} Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), August 1994.