Reactor

An Object Behavioral Pattern for
Concurrent Event Demultiplexing and Dispatching

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

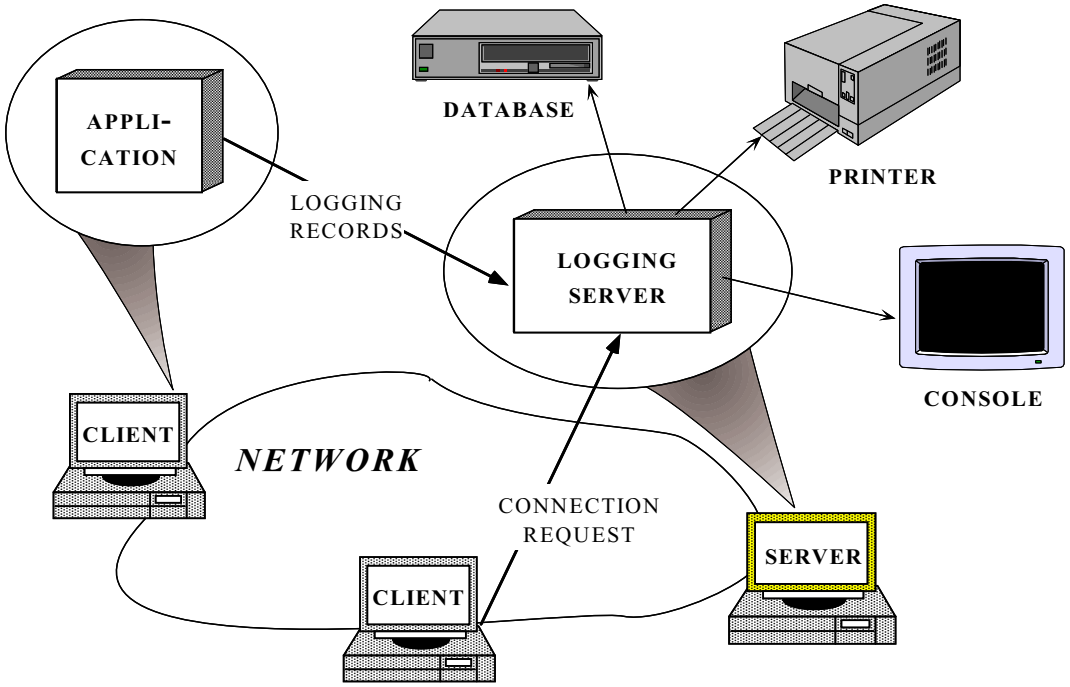
An earlier version of this paper appeared in the 1^{st} Annual Conference on the Pattern Languages of Programs, Monicello, Illinois, August 4 – 6, 1994.

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 event handlers.

2 Motivation

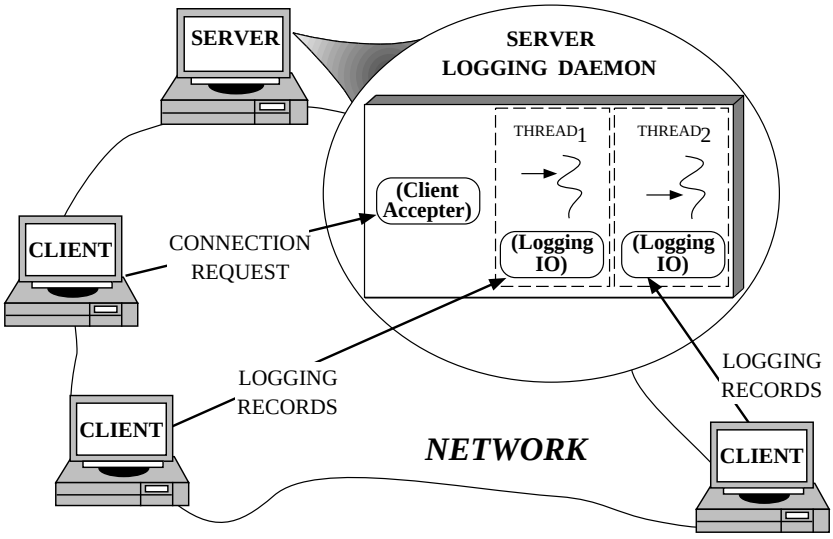
To illustrate the Reactor pattern, consider the event-driven server for a distributed logging facility shown below:



Applications use this system to log information (such as error notifications, debugging statements, and status updates) in a distributed environment. In this facility, logging records are sent to a central logging server. The logging server may output the logging records onto a console, a printer, a file, or a network management information base, etc.

In the distributed logging architecture, a logging server processes logging records sent by multiple clients. These records may arrive concurrently on multiple I/O handles[1] (*i.e.*, one handle for each client). The server also listens for new connections on a designated I/O handle. This handle is used to accept connection requests from new clients. Since input may arrive from multiple sources concurrently, a single-threaded server may not block indefinitely reading from an I/O handle. In particular, using a blocking `read` on one handle may significantly delay the responsive time for other clients.

One way to develop a logging server is to use multi-threading [1]. In this approach, the server spawns a separate thread for each client connection. Each thread 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 or connection request is processed within the thread. The thread then re-blocks awaiting subsequent input. The participants in a thread-based logging server are illustrated in the following figure:



Using multi-threading to implement event handling in the logging server has several drawbacks:

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

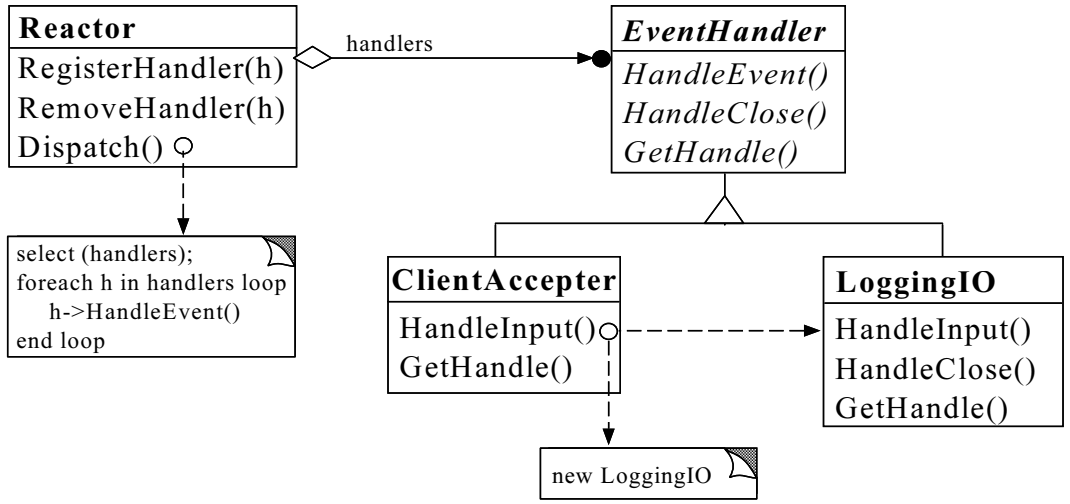
Often, a more appropriate way to develop a logging server is to use the *Reactor pattern*. The Reactor pattern is useful for managing a single-threaded event loop that performs event demultiplexing and event handler dispatching in response to events.

The Reactor pattern provides several benefits. First, it facilitates the development of flexible and extensible event-driven applications. In particular, it helps to decouple application-independent mechanisms from application-specific functionality. The application-independent mechanisms demultiplex events and dispatch pre-registered event handlers. The application-specific functionality is performed by methods defined for these objects. By using the Reactor pattern, developers are able to concentrate on application-specific functionality,

[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*. However, the underlying concepts are basically the same.

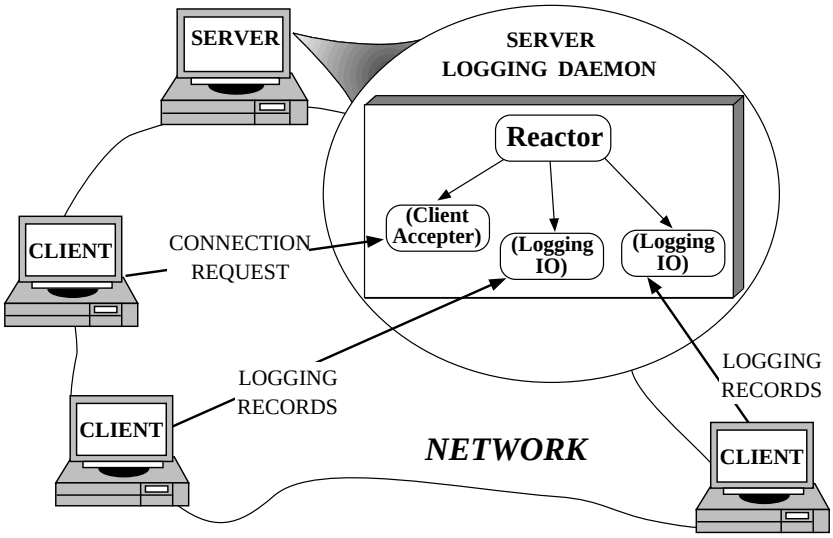
rather than on the lower-level event demultiplexing and handler dispatching mechanisms. Second, the event handlers may evolve independently of the event demultiplexing mechanisms provided by the underlying OS platform.

The following figure uses OMT notation [3] to illustrate the structure of the Reactor pattern:



The `EventHandler` base class provides a standard interface for dispatching event handlers. The `Reactor` uses this interface to callback application-specific operations when certain types of events occur. In the logging server there are two subclasses of the `EventHandler` base class: `LoggingIO` and `ClientAccepter`. These subclasses process events arriving on client I/O handles. The `LoggingIO` event handler is responsible for receiving and processing logging records. The `ClientAccepter` event handler is a factory object. It accepts a new connection request from a client, dynamically allocates a new `LoggingIO` object to process logging records from this client, and registers the new event handler with a `Reactor` object.

The following figure presents a run-time view of the participants in a Reactor-based logging server:



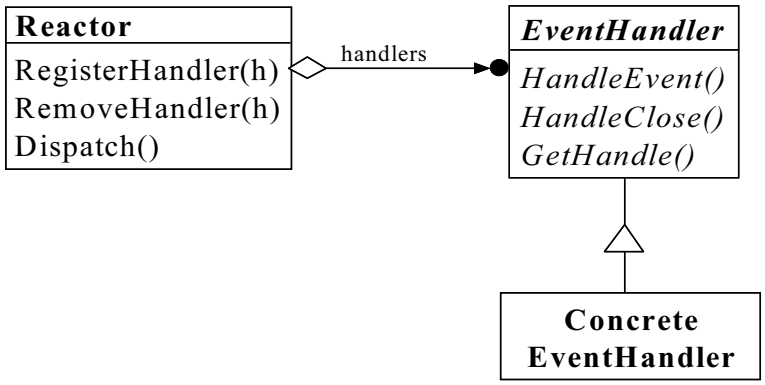
Note that the `ClientAccepter` and the `LoggingIO` event handlers shown in this figure all execute within a single thread of control.

3 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 the messages it receives within a relatively short time period;
- using multi-threading to implement event demultiplexing is either:
  - *infeasible* – due to lack of multi-threading on an OS platform;
  - *undesirable* – due to poor performance on uni-processors 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]
- you want to decouple the application-specific portions of your event handlers from the application-independent mechanisms that perform event demultiplexing and event handler dispatching.

4 Structure



5 Participants

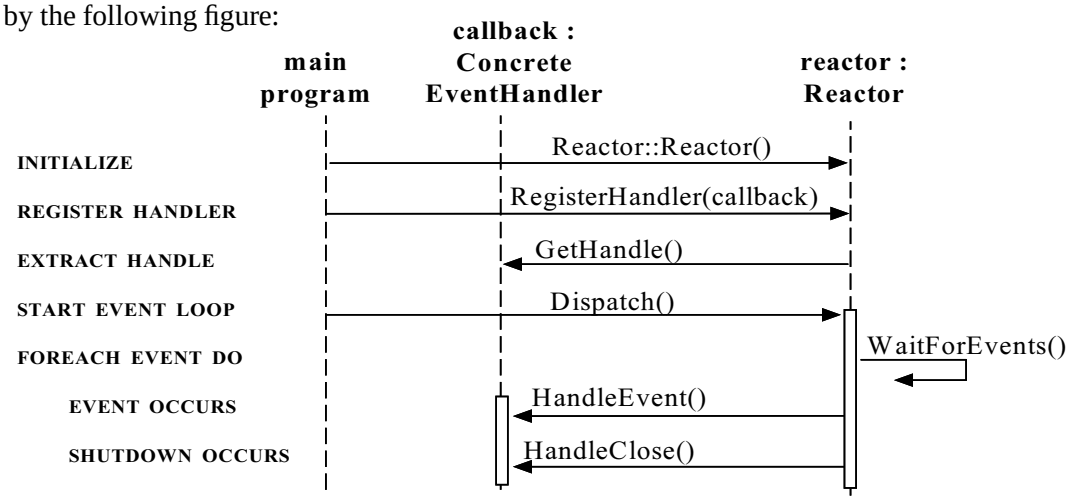
- **Reactor** (`Reactor`)
  - Defines an interface for registering, removing, and dispatching `EventHandler` objects. An implementation of this interface provides a set of application-independent mechanisms. These mechanisms perform event demultiplexing and dispatching of application-specific event handlers in response to events.
- **EventHandler** (`EventHandler`)
  - Specifies an interface for dispatching callback methods defined by objects that are registered to handle events.

[2] For example, the `HandleEvent` method of an `EventHandler` may spawn a separate thread and then handle an incoming event within this thread.

- **ConcreteEventHandler** (`LoggingIO, ClientAccepter`)
  - Implements the callback method(s) that process events in an application-specific manner.

6 Collaborations

- Event handlers are triggered by the occurrence of events. These events are associated with OS handles that are bound to sources of events (such as I/O ports or synchronization objects). To bind the `Reactor` together with these handles, a subclass of `EventHandler` must define the `GetHandle` method. When an `EventHandler` subclass object is registered with the `Reactor`, the `Reactor` invokes the `GetHandle` method to obtain the `EventHandler`'s handle. The `Reactor` uses this handle internally to wait for events to occur on the handle. When events occur, the `Reactor` uses this handle as a key to locate and dispatch the appropriate event handler.
- The application-independent code in a `Reactor` dispatches methods on application-specific event handlers in response to events. This collaboration is structured using method callbacks and is depicted by the following figure:



The `HandleClose` method is invoked by the `Reactor` to perform application-specific cleanup operations immediately before removing an `EventHandler` object.

7 Consequences

- The Reactor pattern decouples application-independent mechanisms from application-specific functionality. The application-independent mechanisms know how to demultiplex events and dispatch the appropriate pre-registered event handlers. Likewise, the application-specific functionality knows how to perform a particular service.
- The Reactor pattern helps to improve the modularity, reusability, and configurability of application software. For example, the `Reactor` decouples the functionality of connection establishment and logging record reception in the logging server into two separate classes. This decoupling enables the reuse of the `ClientAccepter` class for different types of connection-oriented services (such as file transfer, remote login, and video-on-demand). Therefore, only the behavior of the `LoggingIO` class needs to be reimplemented to modify or extend the functionality of the logging server.
- The Reactor improves application portability since its interface may be reused independently of the underlying OS mechanisms (*i.e.,* `select`, `poll`, and `WaitForMultipleObjects`) that perform

event demultiplexing.[3]

- The Reactor pattern provides applications with a rudimentary form of concurrency control. It serializes the invocation of event handlers at the level of event demultiplexing and dispatching within a process or thread. Often, this reduces the need for more complicated synchronization or locking within an application process.
- One consequence of the previous bullet is that the `Reactor` does not preempt event handlers 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 then completes the task in parallel with the Reactor's main dispatch loop.
- 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. This limitation may be overcome by allocating separate processes or separate threads, each of which runs its own `Reactor`.
- The flow of control of the Reactor pattern is complicated by the fact that it performs method callbacks to application-specific event handlers. This makes it somewhat difficult to single-step through the run-time behavior of a `Reactor` and its registered `EventHandlers`.

8 Implementation

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

- *Event Demultiplexing* – A `Reactor` maintains a set of objects that are derived from the `EventHandler` base class. It provides methods for registering and removing these `EventHandler` objects from its set of handlers at run-time. It also provides an interface for dispatching the `HandleEvent` method on the event handler object associated with certain event(s).

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

  The `Reactor`'s `Dispatch` method blocks on the event demultiplexing system call until one or more events occur. When events occur, the `Reactor` unblocks from the event demultiplexing system call. It then dispatches the appropriate callback method(s) associated with `EventHandler` objects that are registered to handle these events.
- *Synchronization* – The Reactor pattern may be used in a multi-threaded application. In this case, it is necessary to protect critical sections within the `Reactor` to prevent race conditions on shared variables (such as the table holding the `EventHandler` subclass objects). A common technique for preventing race conditions involves mutual exclusion mechanisms (such as semaphores or mutex variables [1]).

  Moreover, these mutual exclusion mechanisms should be implemented using *recursive locks* [4] to prevent deadlock. A recursive lock may be re-acquired by the thread that owns the lock *without*

[3] `Select` and `poll` are available in variants of UNIX and `WaitForMultipleObjects` is a Windows NT WIN32 API system call. These system calls all detect and report the occurrence of one or more events that may occur simultaneously on multiple sources of events. These sources of events may include I/O ports, timers, synchronization objects, signals, etc.

blocking the thread. This property is important since the Reactor's `Dispatch` method performs callbacks on application-specific `EventHandler` objects. These `EventHandler` objects may subsequently re-enter the `Reactor` object using its `RegisterHandler` and `RemoveHandler` methods. Therefore, recursive locks are an efficient mechanism for preventing deadlock on locks that are held across `EventHandler` method callbacks within the `Reactor`.

- *I/O Semantics* – The I/O semantics of the underlying OS significantly affect the implementation of the Reactor pattern. For example, the standard `read` and `write` mechanisms on UNIX systems provide "reactive" I/O semantics [5]. In UNIX, 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 straight-forward. `Select` or `poll` are used to indicate the handle(s) that have become ready for I/O. The `Reactor` object then "reacts" by invoking the appropriate `EventHandler` callback methods, which may perform I/O.

  In contrast, Windows NT provides "proactive" I/O semantics [6]. Proactive I/O operations proceed asynchronously and do not cause the caller to block. Subsequently, an application may use the WIN32 `WaitForMultipleObjects` system call to determine when its outstanding asynchronous operations have completed.

  Implementing the Reactor pattern using proactive I/O is more complicated than using reactive I/O. Unlike the reactive I/O scenario described above, I/O operations must be invoked *immediately*, rather than waiting until it becomes *possible* to perform an operation. Therefore, additional information (such as a data buffer or an event handle) must be supplied to the `Reactor` by the event handler *before* an I/O system call is invoked.

  Variations in the I/O semantics of different operating systems 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 platforms to a Windows NT platform.

- *Event Handler Flexibility and Extensibility* – It is possible to develop highly extensible event handlers that are configured into a `Reactor` at installation-time or at run-time. Achieving this degree of flexibility requires the use of object-oriented language features (such as templates, inheritance, and dynamic binding [8]), object-oriented design techniques (such as the Factory Method or Abstract Factory design patterns [9]), and advanced operating system mechanisms (such as explicit dynamic linking and multi-threading [10]).

9 Sample Code

The following code illustrates an example of the Reactor object behavioral pattern. The example implements a portion of the logging server described in Section 2. This example also illustrates the use of an object creational pattern called the Accepter (described in [7]). The Accepter pattern decouples the act of establishing a connection from the service(s) provided once a connection is established. This pattern is useful for simplifying the development of connection-oriented network services (such as file transfer, remote login, distributed logging, and video-on-demand). The Accepter pattern enables the application-specific portion of a service to be modified independently of the mechanism used to establish the connection.

The code shown below implements the `ClientAccepter` class. This class performs the steps necessary to accept connection requests from client applications. In addition, it provides a factory for creating `LoggingIO` objects (described below), which receive and process logging records from clients. The `ClientAccepter` class inherits from `EventHandler`. This enables a `ClientAccepter` to

be registered with a Reactor object. The Reactor automatically dispatches the ClientAccepter's HandleEvent method when a client connection request arrives. As shown in the code, the HandleEvent method invokes the Accept method of the SockListener object to accept a new connection.

The SOCKListener object is a concrete factory object that enables the ClientAccepter to listen and accept connection requests on a I/O port. When a connection arrives from a client, the SOCKListener accepts the connection and produces a SOCKStream object. Henceforth, the SOCKStream object is used to transfer data reliably between the client and the server processes.

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

// Handles connection requests from a remote client.

class ClientAccepter : public EventHandler
{
public:

  // Initialize the accepter endpoint.
  ClientAccepter (INETAddr &addr): listener_ (addr) { }

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

  virtual void HandleEvent (void)
  {
    SOCKStream newConnection;

    this->listener_->Accept (newConnection);

    LoggingIO *cliHandler = new LoggingIO (newConnection);
    reactor.RegisterHandler (cliHandler);
  }

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

  virtual HANDLE GetHandle (void) const
  {
    return this->listener_->GetHandle ();
  }

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

  virtual void HandleClose (void)
  {
    this->listener_->Close ();
  }

private:

  SOCKListener listener_; // Accepts new client connections
};
```

The SOCKListener, SOCKStream, and INETAddr classes used in the implementation of the logging server are part of the SOCK_SAP C++ wrapper library for BSD sockets [11]. SOCK_SAP encapsulates the SOCK_STREAM semantics of the socket interface within a type-secure object-oriented interface. In the Internet domain, SOCK_STREAM sockets are implemented using the TCP transport protocol [5].

The logging server uses the LoggingIO class shown below to receive logging records sent from client applications. In the code, the LoggingIO class inherits from EventHandler. This enables a LoggingIO

object to be registered with the `Reactor`. When logging records arrive, the `Reactor` automatically dispatches the `HandleEvent` method of the associated `LoggingIO` object. This object then receives and processes the logging records.

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

class LoggingIO : public EventHandler
{
public:

  // Initialize the clientStream.

  LoggingIO (SOCKStream &cs): clientStream_ (cs) {}

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

  virtual void HandleEvent (void)
  {
    LogRecord logRecord;

    this->clientStream_->Recv (logRecord);
    logRecord.Print (); // Print logging record to output device.
  }

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

  virtual HANDLE GetHandle (void) const
  {
    return this->clientStream_->GetHandle ();
  }

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

  virtual void HandleClose (void)
  {
    this->clientStream_->Close ();
  }

private:

  SOCKStream clientStream_; // Receives logging records from a client
};
```

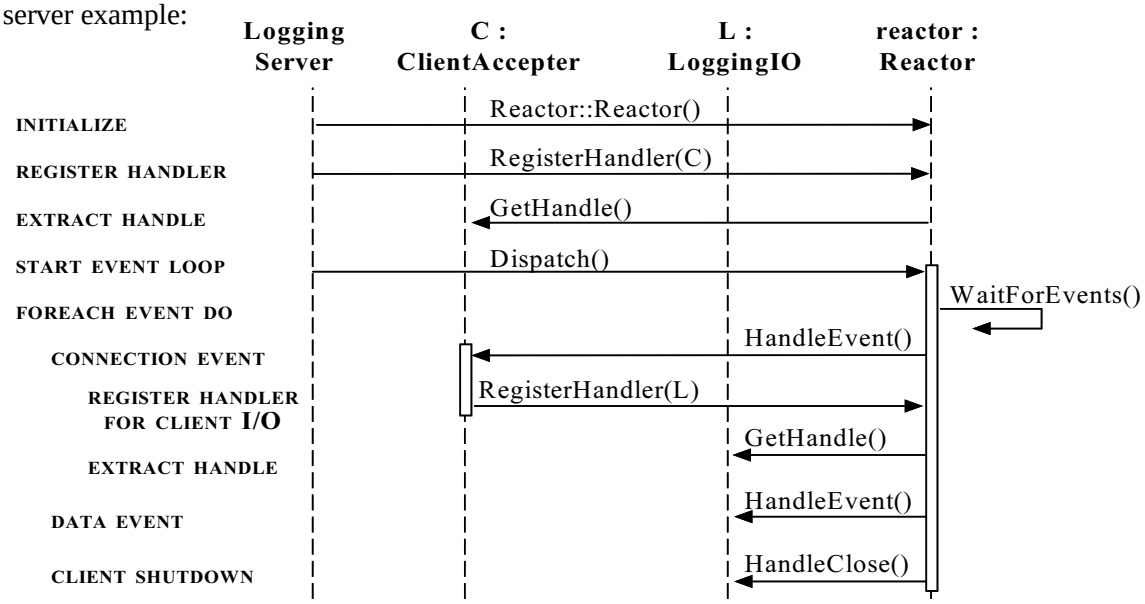
The `ClientAccepter` and `LoggingIO` code shown above "hard-code" the interprocess communication classes (*i.e.,* `SOCKListener` and `SOCKStream`, respectively) used to communicate between clients and the logging server. To remove the reliance on the particular class of communication mechanisms, this example could be generalized to use the Abstract Factory or Factory Method patterns described in [9].

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

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

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

int
main (void)
{
  // Network and port address of the logging server.
  INETAddr addr (PORT);

  // Initialize logging server object endpoint.
  ClientAccepter accepter (addr);

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

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

  return 0;
}
```

The following interaction diagram illustrates the collaboration between the objects participating in the logging server example:



Note that 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 `EventHandler` objects registered with, and controlled by, the `Reactor`.

10 Known Uses

The Reactor pattern has been used in a number of object-oriented frameworks. It is provided by the InterViews window system distribution [12] 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 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. The Reactor pattern has also been used in commercial products including the AT&T Q.port ATM signaling software product, the Ericsson EOS family of telecommunication switch monitoring applications, and the network management portion of the Motorola Iridium mobile personal communications system.

11 Related Patterns

A `Reactor` provides a Facade [9] for event demultiplexing. A Facade is an interface that shields applications from more complex object relationships within a subsystem.

The virtual methods provided by the `Event_Handler` base class are Template Methods [9]. These template methods are used by the `Reactor` to trigger callbacks to the appropriate application-specific processing functionality in response to events.

An `EventHandler` (such as the `LoggingIO` class described in Section 9) may be created using a Factory Method [9]. This allows a subclass to decide which type of `EventHandler` subclass to create.

A Reactor may be implemented as a Singleton [9]. 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 [13].

References

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

[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] D. C. Schmidt, "Transparently Parameterizing Synchronization Mechanisms into a Concurrent Distributed Application," *C++ Report*, vol. 6, July/August 1994.

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

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

[7] D. C. Schmidt and P. Stephenson, "Achieving Reuse Through Design Patterns," in *Proceedings of the* 3^{rd} *C++ World Conference*, (Austin, Texas), SIGS, Nov. 1994.

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

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

[10] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, to appear 1995.

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

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

[13] 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.