

# Reactor

## An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events

Douglas C. Schmidt

schmidt@cs.wustl.edu

Department of Computer Science

Washington University, St. Louis, MO<sup>1</sup>

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

The Reactor design pattern handles service requests that are delivered concurrently by multiple clients. This pattern simplifies event-driven applications by integrating the synchronous demultiplexing of events and the dispatching of their corresponding event handlers. In addition, the Reactor pattern also decouples the generic demultiplexing and dispatching code from the application-specific event handler code.

### 2 Also Known As

Dispatcher, Notifier

### 3 Motivation

This section provides the context and motivation for using the Reactor pattern.

#### 3.1 Context and Forces

The Reactor pattern should be applied when an application must handle events from multiple clients concurrently without incurring the programming complexity, performance overhead, and non-portability of multi-threading.

To illustrate the Reactor pattern, consider the event-driven server for a distributed logging service shown in Figure 1. Client applications use the logging service to record information about their status in a distributed environment. This status information commonly includes error notifications, debugging traces, and performance reports. Logging records are sent to a central logging server, which can write the records to various output devices, such as a console, a printer, a file, or a network management database.

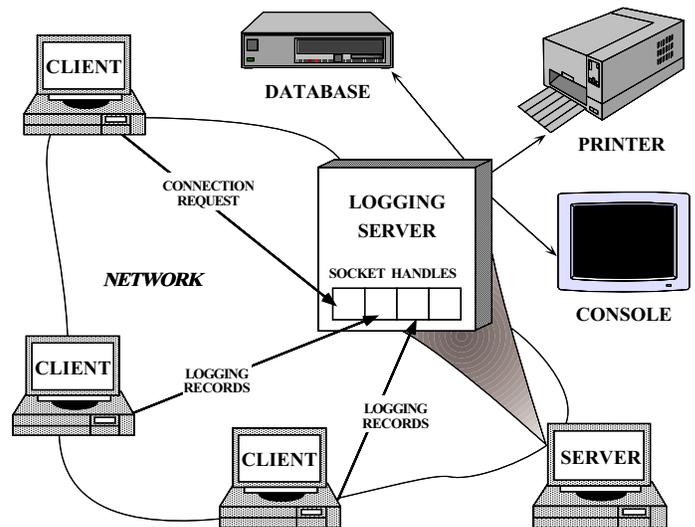


Figure 1: Distributed Logging Service

The logging server shown in Figure 1 handles logging records and connection requests sent by clients. Logging records and connection requests can arrive concurrently on multiple *socket handles*. A socket handle identifies network communication resources managed within the OS.

The logging server communicates with clients using a connection-oriented protocol, such as TCP [1]. The server listens on a “passive-mode” socket handle for connection requests to arrive from new clients. A passive-mode socket is a factory that creates a separate “data-mode” socket handle for each newly connected client. Once clients are connected, they can send logging records concurrently to the server.

Server applications in a distributed system must handle multiple clients that send requests for their services. Before invoking a specific service, however, the server application must demultiplex and dispatch each incoming request to its corresponding service handler. Developing an effective server mechanisms for demultiplexing and dispatching client requests requires the resolution of the following forces:

- **Concurrency:** A server must not block indefinitely handling any single source of events at the exclusion of other event sources since this may significantly delay the respon-

<sup>1</sup>This work was supported in part by a grant by Siemens.

ness to multiple clients;

- **Efficiency:** A server must minimize latency, maximize throughput, and avoid utilizing the CPU(s) unnecessarily;
- **Programming simplicity:** The design of a server should simplify the use of efficient concurrency strategies;
- **Adaptability:** Integrating new or improved services, such as changing message formats or adding server-side caching, should incur minimal modifications and maintenance costs for existing code;
- **Portability:** Porting a server to a new OS platform should not require significant effort;
- **Separation of concerns:** Implementing event handlers for new services should not require modifications to the generic event demultiplexing and dispatching mechanisms.

### 3.2 Common Traps and Pitfalls with Conventional Multi-threaded Solutions

Perhaps the most intuitive way to develop a concurrent logging server is to use multiple threads that can process multiple clients concurrently, as shown in Figure 2. This approach

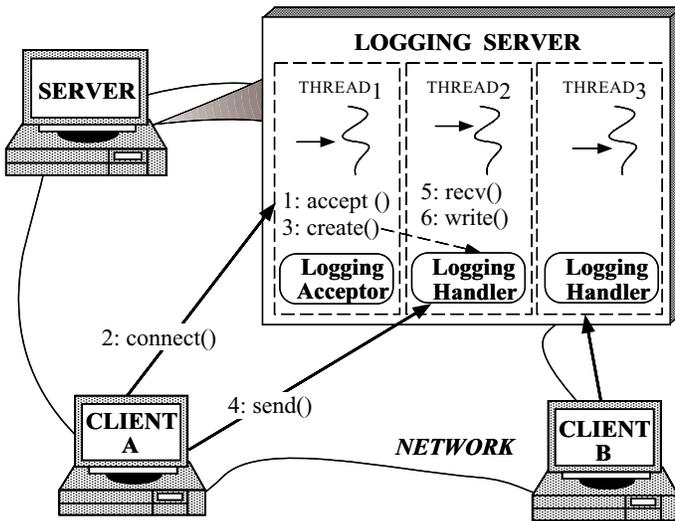


Figure 2: Multi-threaded Logging Server

synchronously accepts network connections and spawns a “thread-per-connection” to handle client logging records.

However, using multi-threading to implement the processing of logging records in the server fails to resolve the following forces:

- **Efficiency:** Threading may lead to poor performance due to context switching, synchronization, and data movement [2];
- **Programming simplicity:** Threading may require complex concurrency control schemes;

- **Portability:** Threading is not available on all OS platforms.

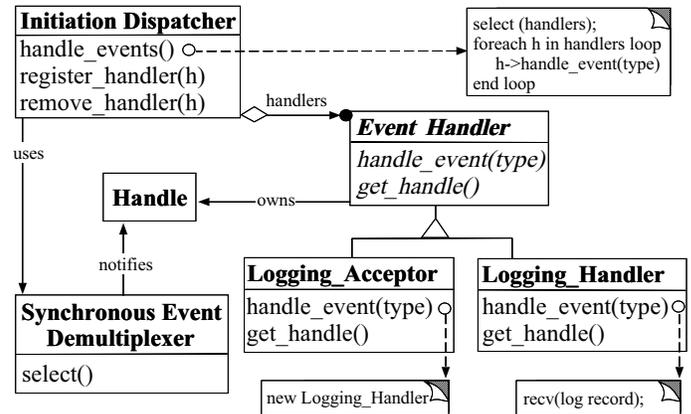
As a result of these drawbacks, multi-threading is often not the most efficient nor the least complex solution to develop a concurrent logging server.

For completeness, a C++ code example that uses synchronous threading to implement the logging server appears in Appendix A.

### 3.3 Solution: Concurrency Through Reactive Operations

Often, a more convenient, efficient, and portable way to develop a logging server is to use the *Reactor pattern*. This pattern integrates the synchronous demultiplexing of events and the dispatching of their corresponding event handlers. It also decouples the implementation of application-specific services from the general-purpose event demultiplexing and dispatching mechanisms.

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 handlers that have registered to process certain types of events. The Initiation Dispatcher uses this interface to call back to a hook method on an application-specific Event Handler when it is possible to initiate an operation without blocking. Concrete Event Handlers, such as the Logging Handler and Logging Acceptor, can be customized to process events associated with socket handles. A C++ code example that applies the Reactor pattern to the logging server appears in Section 8.6.

## 4 Applicability

Use the Reactor pattern when:

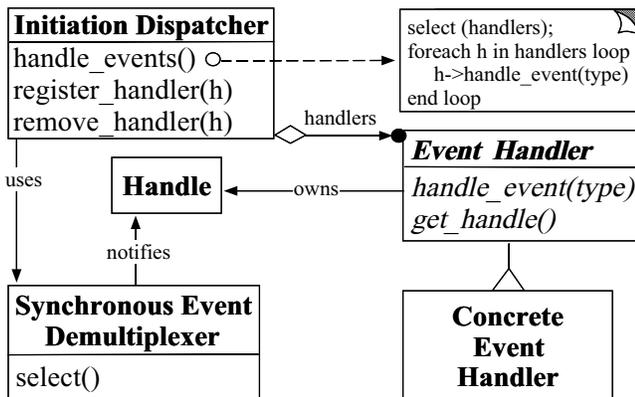
- One or more events may arrive concurrently from different sources of events, and blocking or continuously

polling for incoming events on any individual source 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 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;<sup>2</sup>
- 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 and Participants

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



The key participants in the Reactor pattern include the following:

### Handles

- Handles identify resources that are managed by an OS. These resources commonly include network connections, open files, timers, and synchronization objects. Handles are used in the logging server to

<sup>2</sup>For example, the `handle_event` method of an Event Handler may become an Active Object [4] by spawning a separate thread and handling one or more incoming events within this thread.

identify socket endpoints so that the Synchronous Event Demultiplexer can wait for READ events to occur on them.

### Synchronous Event Demultiplexer

- The Synchronous Event Demultiplexer blocks awaiting events to occur on a set of Handles. It returns when it is possible to initiate an operation on a Handle without blocking. A common demultiplexer for I/O events is `select` [1]. `select` indicates which Handles can have operations invoked on them synchronously without blocking the application process.

### Initiation Dispatcher

- Defines an interface for registering, removing, and dispatching Event Handlers. The Synchronous Event Demultiplexer informs the Initiation Dispatcher when to call back application-specific event handlers in response to certain types of events. Common events include connection acceptance events, data input and output events, and timeout events.

### Event Handler

- Specifies an interface used by the Initiation Dispatcher to call back to hook methods [5] defined by Event Handlers that pre-register to process certain types of events.

### Concrete Event Handler

- Implements the hook method(s) that process events in an application-specific manner. There are two Concrete Event Handlers in the logging server: Logging Handler and Logging Acceptor. The Logging Handler is responsible for receiving and processing logging records. The Logging Acceptor creates and connects Logging Handlers that process logging records from clients.

## 6 Collaborations

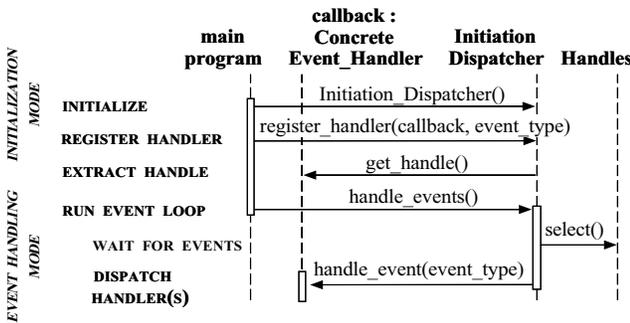
### 6.1 General Collaborations

The following collaborations generally occur in the Reactor pattern:

- When a Concrete Event Handler is registered with the Initiation Dispatcher, the application indicates the type of event(s) this Event Handler wants the Initiation Dispatcher to notify it about when the events occur on the associated Handle.
- The Initiation Dispatcher combines the Handle of all the registered Event Handlers and uses the Synchronous Event Demultiplexer to wait for events to occur on these Handles.

- The Synchronous Event Demultiplexer notifies the Initiation Dispatcher when a Handle corresponding to an event source becomes “ready.” For instance, the TCP protocol layer uses the select synchronous event demultiplexing operation to notify the Initiation Dispatcher operation that a socket is “ready for reading.”
- The Initiation Dispatcher triggers Event Handler methods in response to events on the ready Handles. When events occur, the Initiation Dispatcher uses the Handles activated by the event sources as “keys” to locate and dispatch the appropriate Event Handler hook methods.
- The Initiation Dispatcher calls back to the handle\_event hook method of the Event Handler to perform application-specific functionality in response to an event. The type of event that occurred can be passed as a parameter to the method (an alternative approach is described in Section 8.4).

The following interaction diagram illustrates the collaboration between application code and participants in the Reactor pattern:

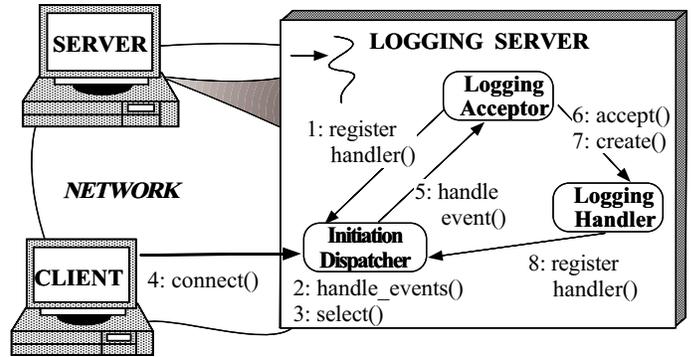


## 6.2 Collaboration Scenarios

The collaborations within the Reactor pattern for the logging server can be illustrated with two scenarios. These scenarios show how a logging server designed using reactive event dispatching handles connections requests and logging data from multiple clients.

### 6.2.1 Client Connects to a Reactive Logging Server

The first scenario shows the steps taken when a client connects to the logging server.

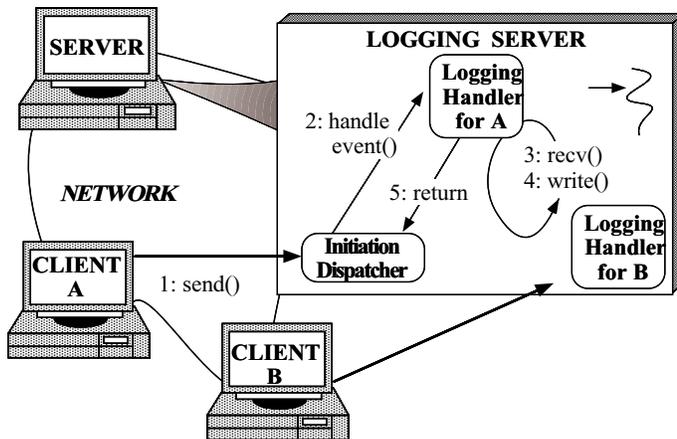


This sequence of steps can be summarized as follows:

1. The logging server registers the Logging Acceptor with the Initiation Dispatcher to accept new connections;
2. The logging server invokes the handle\_events method of the Initiation Dispatcher;
3. The Initiation Dispatcher invokes the synchronous event demultiplexing select operation to wait for connection requests or logging data to arrive;
4. A client connects to the logging server;
5. The Logging Acceptor is notified by the Initiation Dispatcher of the new connection request;
6. The Logging Acceptor accepts the new connection;
7. The Acceptor creates a Logging Handler to service the new client;
8. Logging Handler registers its socket handle with the Initiation Dispatcher and instructs the dispatcher to notify it when the socket becomes “ready for reading.”

### 6.2.2 Client Sends Logging Record to a Reactive Logging Server

The second scenario shows the sequence of steps that the reactive logging server takes to service a logging record.



This sequence of steps can be summarized as follows:

1. The client sends a logging record;
2. The Initiation Dispatcher notifies the associated Logging Handler when a client logging record is queued on its socket handle by OS;
3. The record is received in a non-blocking manner (steps 2 and 3 repeat until the logging record has been completely received);
4. The Logging Handler processes the logging record and writes it to the standard output.
5. The Logging Handler returns control to the Initiation Dispatcher's event loop.

## 7 Consequences

### 7.1 Benefits

The Reactor pattern offers the following benefits:

**Separation of concerns:** The Reactor pattern decouples application-independent demultiplexing and dispatching mechanisms from application-specific hook method functionality. The application-independent mechanisms become reusable components that know how to demultiplex events and dispatch the appropriate hook methods defined by Event Handlers. In contrast, the application-specific functionality in a hook method knows how to perform a particular type of service.

**Improve modularity, reusability, and configurability of event-driven applications:** The pattern decouples server functionality into separate classes. For instance, there are two separate classes in the logging server: 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.

**Improves application portability:** The Initiation Dispatcher's interface can be reused independently of the 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. Common sources of events may include I/O handles, timers, and synchronization objects. On UNIX platforms, the event demultiplexing system calls are called `select` and `poll` [1]. In the Windows NT Win32 API, the `WaitForMultipleObjects` system call performs event demultiplexing [6].

**Provides coarse-grained concurrency control:** The Reactor pattern serializes the invocation of event handlers at the level of event demultiplexing and dispatching within a process or thread. Serialization at the Initiation Dispatcher level often eliminates the need for more complicated synchronization or locking within an application process.

### 7.2 Liabilities

The Reactor pattern has the following liabilities:

**Restricted applicability:** The Reactor pattern can only be applied efficiently if the OS supports Handles. It is possible to emulate the semantics of the Reactor pattern using multiple threads within the Initiation Dispatcher, e.g. one thread for each Handle. However, this design is typically very inefficient since it serializes all Event Handlers, thereby increasing synchronization and context switching overhead without enhancing parallelism.

**Non-preemptive:** In a single-threaded application process, Event Handlers are not preempted while they are executing. This implies that an Event Handler should not perform blocking I/O on an individual Handle since this will block the entire process and decrease the responsiveness for clients connected to other Handles. Therefore, for long-duration operations, such as transferring multi-megabyte medical images [7], the Active Object pattern [8] may be more effective. An Active Object uses multi-threading or multi-processing to complete its tasks in parallel with the Initiation Dispatcher's main event-loop.

**Hard to debug:** Applications written with the Reactor pattern can be hard to debug since the inverted flow of control oscillates between the framework infrastructure and the method callbacks on application-specific handlers. This increases the difficulty of "single-stepping" through the runtime behavior of a framework within a debugger since application developers may not understand or have access to the framework code. This is similar to the problems encountered trying to debug a compiler lexical analyzer and parser written with LEX and YACC. In these applications, debugging is straightforward when the thread of control is within the user-defined action routines. Once the thread of control returns to the generated Deterministic Finite Automata (DFA) skeleton, however, it is hard to follow the program logic.

## 8 Implementation

This section describes how to implement the Reactor pattern in C++. The implementation described below is influenced by the reusable components provided in the ACE communication software framework [2].

### 8.1 Select the Synchronous Event Demultiplexer Mechanism

The Initiation Dispatcher uses a Synchronous Event Demultiplexer to wait synchronously until one or more events occur. This is commonly implemented using an OS event demultiplexing system call like `select`. The `select` call indicates which Handle(s) are ready to perform I/O operations without blocking the application process. In general, the Synchronous Event Demultiplexer can be selected from existing OS mechanisms, rather than developed by implementers of the Reactor pattern.

### 8.2 Develop an Initiation Dispatcher

The following are the steps necessary to develop the Initiation Dispatcher:

**Implement the Event Handler table:** A Initiation Dispatcher maintains a table of Concrete Event Handlers. Therefore, the Initiation Dispatcher should provide methods to register and remove the handlers from this table at run-time.

**Implement the event loop entry point:** The entry point into the event loop of the Initiation Dispatcher should be provided by a `handle_events` method. This method controls the Handle demultiplexing provided by the Synchronous Event Demultiplexer, as well as performing Event Handler dispatching. When events occur, the Initiation Dispatcher returns from the synchronous event demultiplexing call and “reacts” by dispatching the Event Handler’s `handle_event` hook method for each handle that is “ready.” This hook method executes user-defined code and returns control to the Initiation Dispatcher when it completes.

The following C++ class illustrates the core methods on the Initiation Dispatcher’s public interface:

```
enum Event_Type
// = TITLE
//   Types of events handled by the
//   Initiation_Dispatcher.
//
// = DESCRIPTION
//   These values are powers of two so
//   their bits can be efficiently ``or'd``
//   together to form composite values.
{
    ACCEPT_EVENT = 01,
    READ_EVENT = 02,
    WRITE_EVENT = 04,
    TIMEOUT_EVENT = 010,
```

```
    SIGNAL_EVENT = 011,
    CLOSE_EVENT = 012
};

class Initiation_Dispatcher
// = TITLE
//   Demultiplex and dispatch Event_Handlers
//   in response to client requests.
{
public:
// Register an Event_Handler of a particular
// Event_Type.
int register_handler (Event_Handler *eh,
                    Event_Type et);

// Remove an Event_Handler of a particular
// Event_Type.
int remove_handler (Event_Handler *eh,
                  Event_Type et);

// Entry point into the reactive event loop.
int handle_events (Time_Value *timeout = 0);
};
```

#### Implement the necessary synchronization mechanisms:

If the Reactor pattern is used in an application with only one thread of control it is possible to eliminate all synchronization. In this case, the Initiation Dispatcher serializes the Event Handler `handle_event` hooks within the application’s process.

However, the Initiation Dispatcher can also serve as a central event dispatcher in multi-threaded applications. In this case, critical sections within the Initiation Dispatcher must be serialized to prevent race conditions when modifying or activating shared variables (such as the table holding the Event Handlers). A common technique for preventing race conditions uses mutual exclusion mechanisms like semaphores or mutex variables.

To prevent deadlock, mutual exclusion mechanisms can use *recursive locks* [9]. Recursive locks are an efficient means to prevent deadlock when locks are held by the same thread across Event Handler hook methods within the Initiation Dispatcher. 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 `handle_events` hook method calls back on application-specific Event Handlers. Therefore, application hook method code may subsequently re-enter the Initiation Dispatcher using its `register_handler` and `remove_handler` methods.

### 8.3 Determine the Type of the Dispatching Target

Two types of Event Handlers can be associated with a Handle to serve as the target of an Initiation Dispatcher’s dispatch operation:

**Event Handler objects:** A common way to associate an Event Handler with a Handle is to make the Event Handler an object. For instance, the Reactor pattern implementation shown in Section 5 registers Event Handler

subclass objects with an Initiation Dispatcher. Using an object as the dispatching target makes it convenient to subclass Event Handlers in order to reuse and extend existing components. In addition, objects integrate the state and methods of a service.

**Event Handler functions:** Another way to associate an Event Handler with a Handle is to register a function with the Initiation Dispatcher. Using functions as the dispatching target makes it convenient to register callbacks without having to define a new class that inherits from Event Handler. A hybrid approach using the Adapter pattern [10] can be employed to support both objects and functions.

## 8.4 Define the Event Handling Interface

Assuming that we use Event Handler objects rather than functions, the next step is to define the interface of the Event Handler. There are two approaches:

**A single-method interface:** The OMT diagram in Section 5 illustrates an implementation of the Event Handler base class interface that contains a single method, called `handle_event`, which is used by the Initiation Dispatcher to dispatch events. In this case, the type of the event is passed as a parameter to the method.

The following C++ abstract base class illustrates the single-method interface:

```
class Event_Handler
// = TITLE
// Abstract base class that serves as the
// target of the Initiation_Dispatcher.
{
public:
// Hook method that is called back by the
// Initiation_Dispatcher to handle events.
virtual int handle_event (Event_Type et) = 0;

// Hook method that returns the underlying
// I/O Handle.
virtual Handle get_handle (void) const = 0;
};
```

The advantage of the single-method interface is that it is possible to add new types of events without changing the interface. However, this approach encourages the use of switch statements in the subclass's `handle_event` method, which limits its extensibility.

**A multi-method interface:** Another way to implement the Event Handler interface is to define separate virtual hook methods for each type of event (such as `handle_input`, `handle_output`, or `handle_timeout`).

The following C++ abstract base class illustrates the single-method interface:

```
class Event_Handler
{
public:
```

```
// Hook methods that are called back by
// the Initiation_Dispatcher to handle
// particular types of events.
virtual int handle_accept (void) = 0;
virtual int handle_input (void) = 0;
virtual int handle_output (void) = 0;
virtual int handle_timeout (void) = 0;
virtual int handle_close (void) = 0;

// Hook method that returns the underlying
// I/O Handle.
virtual Handle get_handle (void) const = 0;
};
```

The benefit of the multi-method interface is that it is easy to extend since subclassing does not involve switch statements. However, it requires the framework developer to anticipate the set of Event Handler methods in advance.

Both approaches described above are examples of the hook method pattern described in [5] and the Factory Call-back pattern described in [11]. The intent of these patterns is to provide well-defined hooks that can be specialized by applications and called back by lower-level dispatching code.

## 8.5 Determine the Number of Initiation Dispatchers in an Application

Many applications can be structured using just one instance of the Reactor pattern. In this case, the Initiation Dispatcher can be implemented as a Singleton [10]. This design is useful for centralizing event demultiplexing and dispatching into a single location within an application.

However, some operating systems limit the number of Handles that can be waited for within a single thread of control. For instance, Win32 allows `select` and `WaitForMultipleObjects` to wait for no more than 64 Handles in a single thread. In this case, it may be necessary to create multiple threads, each of which runs its own instance of the Reactor pattern.

Note that Event Handlers are only serialized *within* an instance of the Reactor pattern. Therefore, multiple Event Handlers in multiple threads can run in parallel. This may necessitate additional synchronization mechanisms if Event Handlers in different threads access shared state.

## 8.6 Implement the Concrete Event Handlers

The following code implements the Concrete Event Handlers for the logging server described in Section 3.1. These handlers provide *passive connection establishment* (Logging Acceptor) and *data reception* (Logging Handler).

**The Logging Acceptor class:** This class is an example of the Acceptor pattern [12]. The Acceptor pattern decouples the task of service initialization from the tasks performed after a service is initialized. This pattern enables the application-specific portion of a service, such as the

Logging Handler, to vary independently of the mechanism used to establish the connection.

A Logging Acceptor passively accepts connections from client applications and creates Logging Handler objects, which receive and process logging records from clients. The key methods and data members in the Logging Acceptor class are defined below:

```
class Logging_Acceptor : public Event_Handler
    // = TITLE
    //   Handles client connection requests.
{
public:

    // Initialize the acceptor_ endpoint and
    // register with the Initiation Dispatcher.
    Logging_Acceptor (const INET_Addr &addr);

    // Factory method that accepts a new
    // SOCK_Stream connection and creates a
    // Logging_Handler object to handle logging
    // records sent using the connection.
    virtual void handle_event (Event_Type et);

    // Get the I/O Handle (called by the
    // Initiation Dispatcher when
    // Logging_Acceptor is registered).
    virtual HANDLE get_handle (void) const
    {
        return acceptor_.get_handle ();
    }

private:
    // Socket factory that accepts client
    // connections.
    SOCK_Acceptor acceptor_;
};
```

The Logging Acceptor class inherits from the Event Handler base class. This enables an application to register the Logging Acceptor with an Initiation Dispatcher.

The Logging Acceptor also contains an instance of SOCK Acceptor. This is a concrete factory that enables the Logging Acceptor to accept connection requests on a passive mode socket that is listening to 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.

The SOCK Acceptor and SOCK Stream classes used to implement the logging server are part of the C++ socket wrapper library provided by ACE [13]. These socket wrappers encapsulate 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.

The constructor for the Logging Acceptor registers itself with the Initiation Dispatcher Singleton [10] for ACCEPT events, as follows:

```
Logging_Acceptor::Logging_Acceptor
    (const INET_Addr &addr)
```

```
    : acceptor_ (addr)
{
    // Register acceptor with the Initiation
    // Dispatcher.
    Initiation_Dispatcher::instance ()->
        register_handler (this, ACCEPT_EVENT);
}
```

Henceforth, whenever a client connection arrives, the Initiation Dispatcher calls back to the Logging Acceptor's handle\_event method, as shown below:

```
void
Logging_Acceptor::handle_event (Event_Type et)
{
    // Can only be called for an ACCEPT event.
    assert (et == ACCEPT_EVENT);

    SOCK_Stream new_connection;

    // Accept the connection.
    acceptor_.accept (new_connection);

    // Create a new Logging Handler.
    Logging_Handler *handler =
        new Logging_Handler (new_connection);
}
```

The handle\_event method invokes the accept method of the SOCK Acceptor to passively establish a SOCK Stream. Once the SOCK Stream is connected with the new client, a Logging Handler is allocated dynamically to process the logging requests. As shown below, the Logging Handler registers itself with the Initiation Dispatcher, which will then demultiplex all the logging records of its associated client to it.

**The Logging Handler class:** The logging server uses the Logging Handler class shown below to receive logging records sent by client applications:

```
class Logging_Handler : public Event_Handler
    // = TITLE
    //   Receive and process logging records
    //   sent by a client application.
{
public:

    // Initialize the client stream.
    Logging_Handler (SOCK_Stream &cs);

    // Hook method that handles the reception
    // of logging records from clients.
    virtual void handle_event (Event_Type et);

    // Get the I/O Handle (called by the
    // Initiation Dispatcher when
    // Logging_Handler is registered).
    virtual HANDLE get_handle (void) const
    {
        return this->peer_stream_.get_handle ();
    }

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

Logging Handler inherits from Event Handler, which enables it to be registered with the Initiation Dispatcher, as shown below:

```

Logging_Handler::Logging_Handler
(SOCK_Stream &cs)
: peer_stream_ (cs)
{
// Register with the dispatcher for
// READ events.
Initiation_Dispatcher::instance ()->
register_handler (this, READ_EVENT);
}

```

Once it's created, a Logging Handler registers itself for READ events with the Initiation Dispatcher Singleton. Henceforth, when a logging record arrives, the Initiation Dispatcher automatically dispatches the handle\_event method of the associated Logging Handler, as shown below:

```

void
Logging_Handler::handle_event (Event_Type et)
{
if (et == READ_EVENT) {
Log_Record log_record;

this->peer_stream_.recv (log_record);

// Write logging record to standard output.
log_record.write (STDOUT);
}
else if (et == CLOSE_EVENT) {
this->peer_stream_.close ();
delete (void *) this;
}
}

```

When a READ event occurs, the Initiation Dispatcher calls back to Logging\_Handler's handle\_event method. This method receives, processes, and writes the logging record to the standard output. Likewise, when the client closes down the connection the Initiation Dispatcher passes a CLOSE event, which informs the Logging Handler to shut down its SOCK Stream and delete itself.

## 8.7 Implement the Server

The logging server contains a single main function.

**The logging server main function:** This function implements a single-threaded concurrent logging server that waits in an event loop for an Initiation Dispatcher to notify it to initiate accept and recv operations via callbacks to the appropriate Event Handlers. The main entry point into the logging server is defined as follows:

```

// Server port number.
const u_short PORT = 10000;

int
main (void)
{
// Logging server port number.
INET_Addr server_addr (PORT);

// Initialize logging server endpoint.
Logging_Acceptor ca (server_addr);

// Main event loop that handles client

```

```

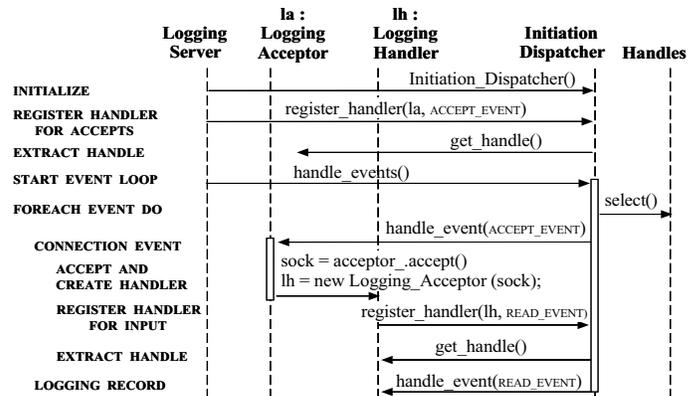
// logging records and connection requests.
for (;;)
Initiation_Dispatcher::instance ()->
handle_events ();

/* NOTREACHED */
return 0;
}

```

The main program creates a Logging Acceptor, initializes it with the port number of the logging server, and enters its main event-loop. Subsequently, the Initiation Dispatcher Singleton uses the select event demultiplexing system call to synchronously wait for connection requests and logging records to arrive from clients.

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



Once the Initiation Dispatcher object is initialized, it becomes the primary focus of the control flow within the logging server. All subsequent activity is triggered by hook methods on the Logging Acceptor and Logging Handler objects registered with, and controlled by, the Initiation Dispatcher.

When data arrives on the network connection, the Initiation Dispatcher calls back the Logging Acceptor, which accepts the network connection and creates a Logging Handler. This Logging Handler then registers with the Initiation Dispatcher for READ events. Thus, when a client sends a logging record, the Initiation Dispatcher calls back to the Logging Handler to process the incoming record from that client connection in the logging server's single thread of control.

## 9 Known Uses

The Reactor pattern has been used in many object-oriented frameworks, including the following:

- **InterViews:** The Reactor pattern is implemented by the InterViews [14] window system distribution, where it is known as the Dispatcher. The InterViews Dispatcher is used to define an application's main event

loop and to manage connections to one or more physical GUI displays.

- **ACE Framework:** The ACE framework [15] uses the Reactor pattern as its central event demultiplexer and dispatcher.

The Reactor pattern has been used in many commercial projects, including:

- **CORBA ORBs:** The ORB Core layer in many single-threaded implementations of CORBA [16] (such as VisiBroker and Orbix) use the Reactor pattern demultiplex and dispatch ORB requests to servants.

- **Ericsson EOS Call Center Management System:** This system uses the Reactor pattern to manage events routed by Event Servers [17] between PBXs and supervisors in Call Center Management system.

- **Project Spectrum:** The high-speed medical image transfer subsystem of project Spectrum [7] uses the Reactor pattern in a medical imaging system.

## 10 Related Patterns

The Reactor pattern is related to the Observer pattern [10], where all dependents are informed when a single subject changes. In the Reactor pattern, a single handler is informed when an event of interest to the handler occurs on a source of events. The Reactor pattern is generally used to demultiplex events from multiple sources to their associated event handlers, whereas an Observer is often associated with only a single source of events.

The Reactor pattern is related to the Chain of Responsibility (CoR) pattern [10], where a request is delegated to the responsible service provider. The Reactor pattern differs from the CoR pattern since the Reactor associates a specific Event Handler with a particular source of events, whereas the CoR pattern searches the chain to locate the first matching Event Handler.

The Reactor pattern can be considered a *synchronous* variant of the asynchronous Proactor pattern [18]. The Proactor supports the demultiplexing and dispatching of multiple event handlers that are triggered by the *completion* of *asynchronous* events. In contrast, the Reactor pattern is responsible for demultiplexing and dispatching of multiple event handlers that are triggered when it is possible to *initiate* an operation *synchronously* without blocking.

The Active Object pattern [8] decouples method execution from method invocation 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 when threads are not available or when the overhead and complexity of threading is undesirable.

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

## 11 Concluding Remarks

The Reactor pattern provides several benefits for event-driven applications. It simplifies the development of flexible applications based on reusable components by decoupling (1) application-independent demultiplexing and dispatching mechanisms from (2) application-specific functionality performed by user-defined methods in the `Event Handler`. Likewise, the Reactor pattern facilitates application extensibility by allowing `Event Handlers` to evolve independently of the event demultiplexing mechanisms provided by the OS.

## Acknowledgements

Thanks to Frank Buschmann and John Vlissides for helpful comments on this paper.

## References

- [1] W. R. Stevens, *UNIX Network Programming, First Edition*. Englewood Cliffs, NJ: Prentice Hall, 1990.
- [2] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6<sup>th</sup> 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] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.
- [5] W. Pree, *Design Patterns for Object-Oriented Software Development*. Reading, MA: Addison-Wesley, 1994.
- [6] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.
- [7] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.
- [8] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Proceedings of the 2<sup>nd</sup> Annual Conference on the Pattern Languages of Programs*, (Monticello, Illinois), pp. 1–7, September 1995.
- [9] D. C. Schmidt, "An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit," Tech. Rep. WUCS-95-31, Washington University, St. Louis, September 1995.
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
- [11] S. Berczuk, "A Pattern for Separating Assembly and Processing," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.
- [12] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

- [13] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," in *Proceedings of the 2<sup>nd</sup> Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.
- [14] M. A. Linton and P. R. Calder, "The Design and Implementation of InterViews," in *Proceedings of the USENIX C++ Workshop*, November 1987.
- [15] D. C. Schmidt, "The ACE Framework." Available from [www.cs.wustl.edu/~schmidt/ACE.html](http://www.cs.wustl.edu/~schmidt/ACE.html), 1997.
- [16] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.
- [17] 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.
- [18] T. Harrison, I. Pyarali, D. C. Schmidt, and T. Jordan, "Proactor – An Object Behavioral Pattern for Dispatching Asynchronous Event Handlers," in *The 4<sup>th</sup> Pattern Languages of Programming Conference*, September 1997.

## A Implementing the Logging Server with Synchronous Multi-threading

This section shows how to use synchronous I/O with multiple threads to develop the logging server shown in Figure 2. The sequence of steps to service a logging record from a client can be summarized as follows:

1. The Logging Acceptor runs in thread<sub>1</sub> and blocks in the accept socket call waiting for client connection requests;
2. A client connects to the server;
3. The Logging Acceptor accepts the new connection and creates a new Logging Handler to service the client – this handler is run in the newly-created thread<sub>2</sub>;
4. The client sends a logging record;
5. The client's logging record is synchronously received from the network connection by thread<sub>2</sub>;
6. The record is written to the standard output.

Steps 4, 5, and 6 are repeated until the client closes down the connection. The code below shows an implementation of the Logging Handler using synchronous multi-threading.

```
class Logging_Handler : public Task
// = TITLE
//   Handle client logging records.
//
// = DESCRIPTION
//   The Logging_Handler class runs as
//   an Active Object, i.e., with its
//   own thread of control. The Task
//   base clas defines activate () and
//   the svc() hook for Active Objects.
{
public:
// This method is called by the
```

```
// Logging_Acceptor when a new
// client connects to the server.
Logging_Handler (Socket_Stream &client)
{
    Logging_Record record;

    // Store reference to client.
    client_ = client;

    // Become an Active Object (which
    // spawns a thread and calls the svc()
    // hook).
    activate ();
}

// Hook method called by activate().
void svc (void)
{
    Logging_Record record;

    // Synchronously read the logging
    // record from the network connection
    // and write it to standard output.
    while (client_>recv (record) != DONE)
        record.write (STDOUT);
}

private:
// Socket endpoint.
Socket_Stream client_;

// ...
};
```

The primary advantage of synchronous threading is the simplification of application code. In particular, operations performed by the logging server to service client A's request are mostly independent of the operations required to service client B's request. Thus, it is easy to service different requests in separate threads because the amount of state shared between the threads is low, which minimizes the need for synchronization. Moreover, executing application logic in separate threads allows developers to utilize intuitive sequential commands such as blocking accept and recv operations.