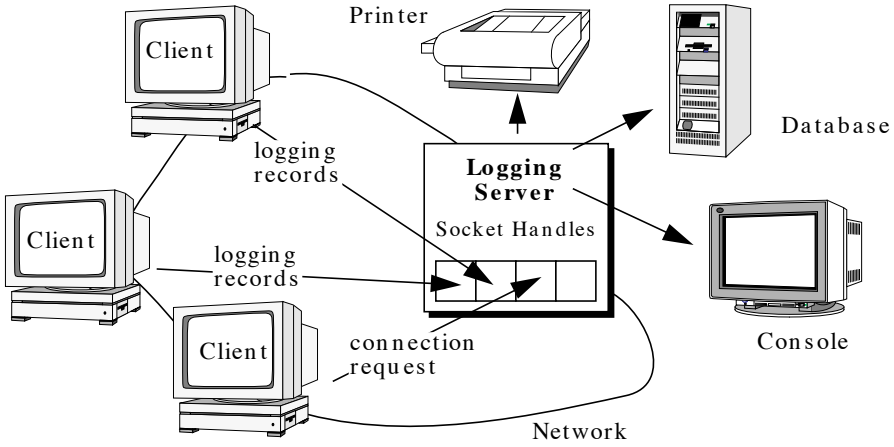
Reactor

The *Reactor* design pattern handles service requests that are delivered concurrently to an application by one or more clients.

Also known as Dispatcher, Notifier

Example Consider an event-driven server for a distributed logging service. 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 diagnostics. 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.



Clients communicate with the logging server using a connection-oriented protocol, such as TCP [Ste90]: clients and the logging service are bound to a connection endpoint designated by an IP address and a TCP port number on the clients and logging server. The port number uniquely identifies the clients and the logging service, respectively. The logging service is typically used by multiple clients, each maintaining its own connection with the logging server. Thus the logging records and connection requests which these clients issue can arrive *concurrently* at the logging server.

However, using multi-threading, to implement the processing of logging records in the server in a 'thread-per-connection' fashion fails to resolve the following aspects:

- *Efficiency*. Threading may lead to poor performance due to context switching, synchronization, and data movement [Sch94].
- *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 neither the most efficient nor the least complex solution to develop a concurrent logging server. Yet we must handle client requests concurrently.

Context A server application in a distributed system that receives service requests from one or more clients concurrently.

Problem Server applications in a distributed system must handle one or more clients that send them service requests. Each such request is typically associated with a specific operating system event. For instance, in our logging server example, the request for processing logging records within the logging service is indicated by a `READ` event. Before invoking a specific service, the server application must therefore demultiplex and dispatch each incoming event to its corresponding service provider. Resolving this problem effectively requires the resolution of the following five *forces*:

- The server must be available to handle input events even if it is waiting for other events to occur. In particular, a server must not block indefinitely handling any single source of events at the exclusion of other event sources since this may significantly delay its responsiveness to other clients.
- A server must minimize latency, maximize throughput, and avoid utilizing the CPU(s) unnecessarily.
- The design of a server should simplify the use of suitable concurrency strategies.
- 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. For instance, implementing application services should not require

modifications to the generic event demultiplexing and dispatching mechanisms.

- Porting a server to a new operating system platform should not require significant effort.

Solution Integrate the synchronous demultiplexing of events with their dispatching to the service providers that handle these events. In addition, decouple these general-purpose event demultiplexing and dispatching mechanisms from the application-specific dispatching of events to services within the service providers.

For each service the application offers, introduce a separate *event handler* that processes certain types of events. Event handlers register with an *initiation dispatcher*, which uses a *synchronous event demultiplexer* to wait for events to occur. When events occur, the synchronous event demultiplexer notifies the initiation dispatcher, which synchronously calls back to the event handler associated with the event. The event handler then dispatches the event to the method that implements the requested service.

Structure The key participants in the Reactor pattern include the following:

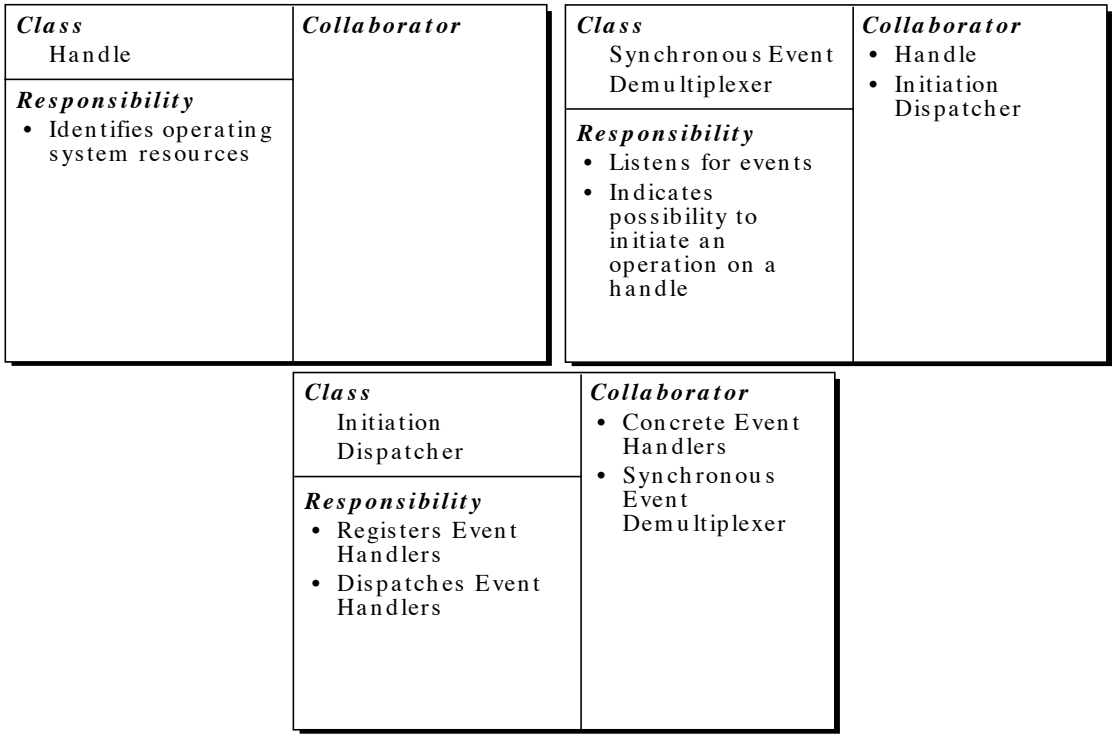
Handles identify resources that are managed by an operating system. These resources commonly include, among others, network connections, open files, timers, and synchronization objects.

➥ Handles are used in the logging server to identify socket endpoints so that a synchronous event demultiplexer can wait for events to occur on them. The two types of events the logging server is interested in are connection events and read events, which represent incoming client connections and logging data, respectively. The logging server maintains a separate connection for each client. Every connection is represented in the server by a socket handle. ❑

A *synchronous event demultiplexer* blocks awaiting events to occur on a set of handles. The blocking does not impede the progress of the process in which the synchronous event demultiplexer resides, since it only blocks when no events are queued at the 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` [Ste90], which is an event demultiplexing system call provided by the UNIX and

Win32 OS platforms. The `select` call indicates which handles can have operations invoked on them synchronously without blocking.

An *initiation dispatcher* defines an interface for registering, removing, and dispatching event handler objects. Ultimately, the synchronous event demultiplexer is responsible for waiting until events occur. When it detects new events, it informs the initiation dispatcher to call back application-specific event handlers. Common events include connection acceptance events, data input and output events, and timeout events.



An *event handler* specifies an interface consisting of a hook method [Pree95] that abstractly represents the dispatching operation for service-specific events.

Concrete event handlers derive from the abstract event handler. Each implements the methods for a specific service that the application offers. In addition, concrete event handlers implement the inherited event dispatching hook method, which is responsible for processing incoming events sent to the service from clients. Applications register

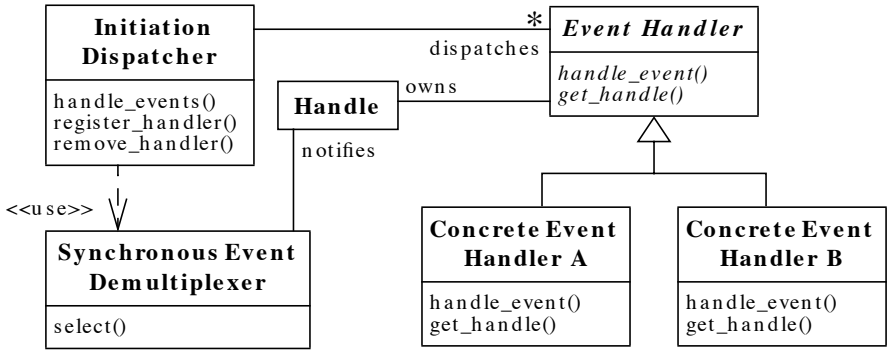
concrete event handlers with the initiation dispatcher to process certain types of events. When these events arrive, the initiation dispatcher calls back the hook method of the appropriate concrete event handler.

Class Event Handler	***Collaborator***
Responsibility • Defines an interface for processing events	• Handle

Class Concrete Event Handler	***Collaborator***
Responsibility • Processes events in a 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 uses the Acceptor-Connector pattern (129) to create and connect logging handlers that process subsequent logging records from clients.❑

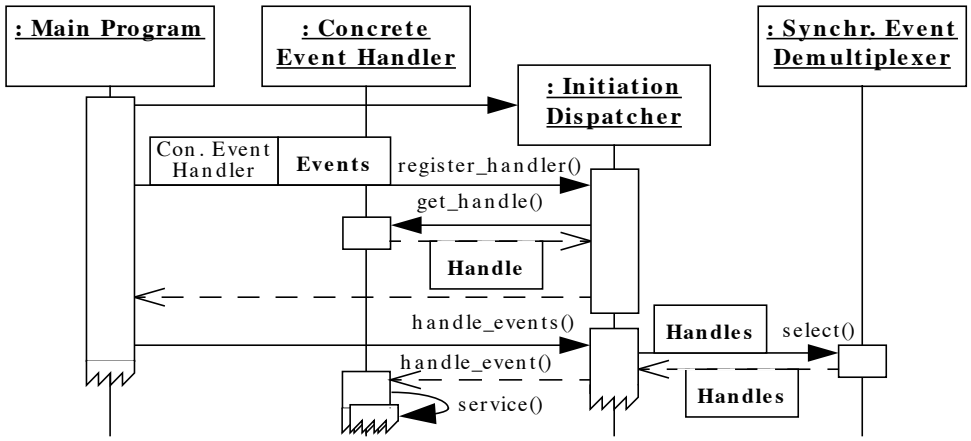
The structure of the participants in the Reactor pattern is illustrated in the following UML class diagram:



Dynamics The following collaborations occur in the Reactor pattern:

- An application registers a concrete event handler with the initiation dispatcher. At this point, the application indicates the type of event(s) this event handler wants the initiation dispatcher to notify it about when the event(s) occur on the associated handle.

- The initiation dispatcher requests each event handler to pass back its internal handle. This handle identifies the event handler to the operating system.
- After all event handlers are registered, the main program calls `handle_events()` to start the initiation dispatcher's event loop. At this point, the initiation dispatcher combines the handles from each registered event handler and uses the synchronous event demultiplexer to wait for events to occur on these handles. For instance, the TCP protocol layer uses the `select` synchronous event demultiplexing operation to wait for client logging record events to arrive on connected socket handles.
- The synchronous event demultiplexer notifies the initiation dispatcher when a handle corresponding to an event source becomes 'ready,' for example, that a TCP socket is 'ready for reading.'
- The initiation dispatcher triggers the event handler hook method 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's hook method. The type of event that occurred can be passed as a parameter to the method and used internally by this method to perform additional service-specific demultiplexing and dispatching. An alternative dispatching approach is described in the Implementation section.

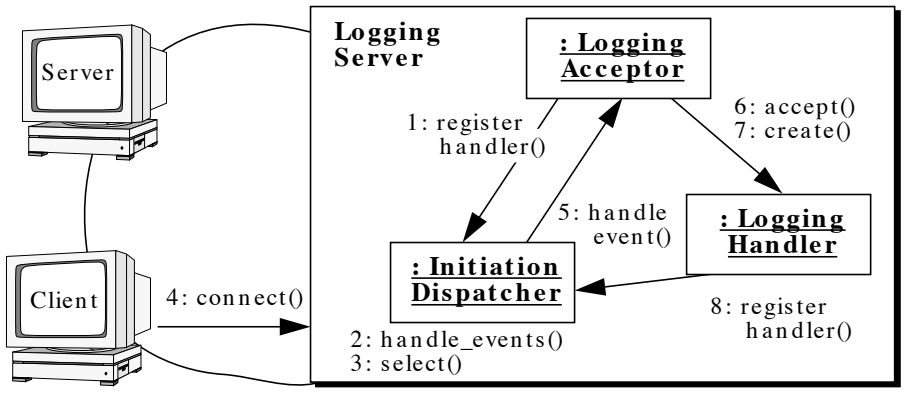


➥ The collaborations within the Reactor pattern for the logging server can be illustrated with two scenarios. In general, clients that

want to log data must first send a connection request to the server. The server waits for these connection requests using a *handle factory*, called logging acceptor, that listens on an address known to clients. When a connection request arrives, the handle factory creates a new logging server to serve this client's requests and establishes a socket connection between the client and the server. Once clients are connected, they can send logging records concurrently to the server using the created socket connection.

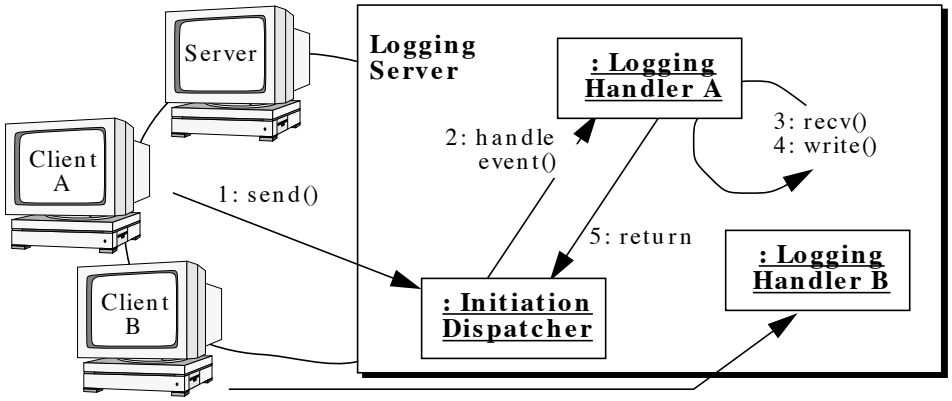
The first scenario shows the steps taken when a client connects to the logging server:

- The logging server (1) registers the logging acceptor with the initiation dispatcher to handle connection requests.
- The logging server invokes the `handle_events()` method (2) of the initiation dispatcher.
- The initiation dispatcher invokes the synchronous event demultiplexing `select()` (3) operation to wait for connection requests or logging data to arrive.
- A client connects (4) to the logging server.
- The logging acceptor is notified by the initiation dispatcher (5) of the new connection request.
- The logging acceptor accepts (6) the new connection and creates (7) a logging handler to service the new client.
- The logging handler registers (8) its socket handle with the initiation dispatcher and instructs the dispatcher to notify it when the socket becomes 'ready for reading.'



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

- A client sends (1) a logging record.
- The initiation dispatcher notifies (2) the associated logging handler when a client logging record is queued on its socket handle by the operating system.
- The record is read from the socket (3) in a non-blocking manner. Steps 2 and 3 repeat until the logging record has been received completely.
- The logging handler processes the logging record and writes (4) it to the standard output.
- The logging handler returns (5) control to the initiation dispatcher's event loop.



❑

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 [Sch94].

1 *Select the synchronous event demultiplexer mechanism*. The initiation dispatcher of the Reactor pattern uses a synchronous event demultiplexer to wait synchronously until one or more events occur. This is commonly implemented using an operating system event demultiplexing system call like `select`. The `select` call indicates which handles are ready to perform I/O operations without blocking the operating system process in which the application-specific service handlers reside. In general, the synchronous event demultiplexer is

based upon existing operating system mechanisms, rather than developed by implementers of the Reactor pattern.

2 *Develop an initiation dispatcher*, according to the following steps:

2.1 *Implement the event handler table*. An initiation dispatcher maintains a table of concrete event handlers. The data stored in this table consists of a set of <handle, event handler, event registrations> tuples. The handles are keys that identify their associated event handlers by indexing into the table entries. Note that the 'type of event(s)' each event handler is registered for is also stored in the table. The table can be implemented in various ways, such as using hashing or linear search. If handles are represented as a contiguous range of small integral values, they can be used to directly index into the table.

2.2 *Implement the event handler registration functionality*. The initiation dispatcher provides methods to register and remove event handlers from event handler table at run-time. The registration method usually expects two parameters: one that identifies the event handler and another that indicates the type of event(s) that the event handler can handle. The registration method calls back to the event handler to obtain its associated handle. With this information, the registration method creates a new entry into the event handler table. Once the registration method terminates, the event handler that called it is registered with the initiation dipatcher's event demultiplexing and dispatching mechanism. The method for removing an event handler from the initiation dipatcher's event demultiplexing and dispatching mechanism removes the corresponding entry in the event handler table when it is no longer registered for any events.

2.3 *Implement the event loop entry point*. The initiation disptacher provides a method that represents the entry point into its event loop. This method, which we call `handle_events()`, controls the handle demultiplexing provided by the synchronous event demultiplexer, as well as performing event handler dispatching. Often, the main event loop of the entire application is controlled by this entry point.

When events occur, the initiation dispatcher returns from the synchronous event demultiplexing call and 'reacts' by dispatching the event handlers that are represented by each handle that is now ready.

➥ The following C++ class illustrates the core methods on the initiation dispatcher's public interface:

```
enum Event_Type {
     // Types of events handled by the
     // Initiation_Dispatcher. 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 = 020,
     CLOSE_EVENT = 040
};

class Initiation_Dispatcher {
     // 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);
};
```

❑

2.4 *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 dispatching of 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 state, 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 self-deadlock, mutual exclusion mechanisms can use recursive locks [Sch95a]. Recursive locks help 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()` method calls back on application-specific concrete event handlers. Application hook method code may subsequently re-enter the initiation dispatcher via its `register_handler()` and `remove_handler()` methods.

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 dispatching logic. Implementations of the Reactor pattern can select either one or both of the following dispatching alternatives:

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 the Structure section registers concrete event handler 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 into a single component.

Event handler functions. Another way to associate an event handler with a handle that was not described in earlier sections 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`.

The Adapter pattern [GHJV95] can be employed to support both objects and functions simultaneously. For instance, an adapter could be defined using an event handler object that holds a pointer to an event handler function. When the `handle_event()` method was invoked on the event handler adapter object, it could automatically forward the call to the event handler function that it holds.

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 the Structure section illustrates an implementation of the `Event_Handler` base class interface that contains a single method, known as `handle_event()`,

which is used by the initiation dispatcher to dispatch events. In this case, the type of the event that has occurred is passed as a parameter to the method.

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

```
class Event_Handler {
     // 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 multi-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 a multi-method interface is that it is easy to selectively override methods in the base class and avoid further demultiplexing via `switch` or `if` statements in the hook method. However, it requires the framework developer to anticipate the set of event handler methods in advance. For instance, the various `handle_*` methods in the `Event_Handler` interface above are tailored for I/O events available through the UNIX `select` mechanism. However, this interface is not broad enough to encompass all the types of events handled via the Win32 `WaitForMultipleObjects` mechanism [Sch95b].

Both approaches described above are examples of the Hook Method pattern described in [Pree95] and Steve Berczuk's Factory Callback pattern described in [PLoP94]. 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.

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 [GHJV95]. 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 configuration may necessitate the use of additional synchronization mechanisms if event handlers in different threads access shared state.

6 *Implement the concrete event handlers*. The concrete event handlers are typically created by application developers to perform specific services in response to particular events. The developers must determine what processing to perform when the corresponding hook method is invoked by the initiation dispatcher.

➥ The following code implements the concrete event handlers for our logging server example. The logging acceptor provides *passive connection establishment*, and the logging handler provides *data reception*.

The `Logging_Acceptor` class is an example of the acceptor component in the Acceptor-Connector pattern (129). This pattern decouples the task of service initialization from the tasks performed after a service is initialized. The 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 connection requests from client applications and creates client-specific 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 {
     // 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 Wrapper Facades (15) from the C++ socket wrapper library provided by the ACE communication framework [Sch97]. They 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 TCP.

The constructor for the logging acceptor registers itself with the initiation dispatcher singleton [GHJV95] for ACCEPT events, as follows:

```
Logging_Acceptor::Logging_Acceptor
     (const INET_Addr &addr) : acceptor_ (addr) {
     // Register acceptor with the Initiation
     // Dispatcher, which "double dispatches" the
     // Logging_Acceptor::get_handle() method to
     // obtain the HANDLE.
     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.

```
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 object to passively establish a SOCK_Stream object. Once the SOCK_Stream object is connected with the new client, a logging handler is allocated dynamically on the logging server to process the logging requests:

```
class Logging_Handler : public Event_Handler {
     // 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_;
};
```

Class Logging_Handler inherits from Event_Handler, which enables it to be registered with the initiation dispatcher:

```
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 with the initiation dispatcher singleton to receive READ_EVENTs. Henceforth, when a logging record arrives, the initiation dispatcher automatically dispatches the handle_event() method of the associated logging handler:

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

          this->peer_stream_.recv ((void *) log_record,
                                   sizeof 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 the logging handler's `handle_event()` hook method. This method receives, processes, and writes the logging record to the standard output (`STDOUT`). 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. ❑

7 *Implement the server.*

➥ The logging server contains a single `main` function. This function implements a single-threaded concurrent logging server that waits in the initiation dispatcher's `handle_events()` event loop. As requests arrive from clients, the initiation dispatcher invokes the appropriate concrete event handler hook methods, which accept connections and receive and process logging records. The main entry point into the logging server is defined as follows:

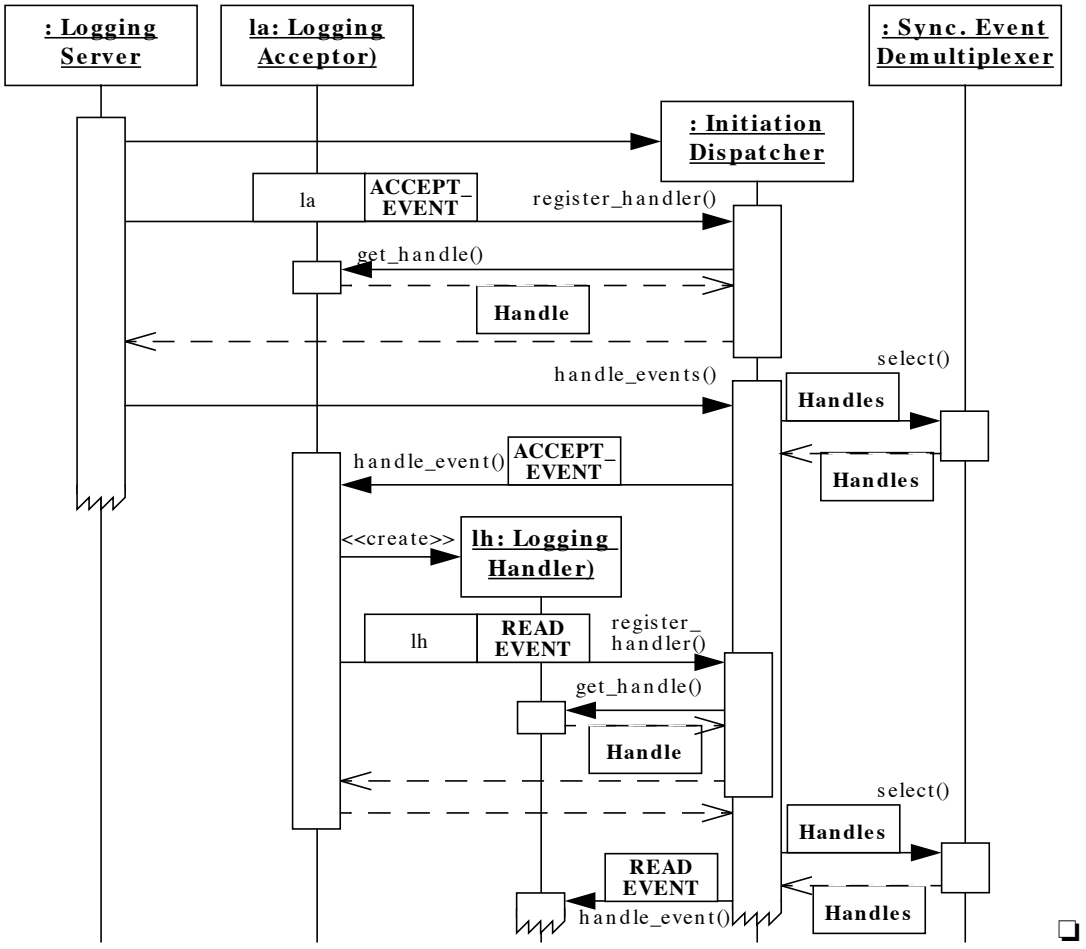
```
// Server port number.
const u_short PORT = 10000;

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

    // Initialize logging server endpoint and register
    // with the Initiation_Dispatcher.
    Logging_Acceptor la (server_addr);

    // Main event loop that handles client
    // logging records and connection requests.
    for (;;)
        Initiation_Dispatcher::instance ()->
                                    handle_events ();
    /* NOTREACHED */ return 0;
}
```

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



❑

The main program creates a logging acceptor, whose constructor initializes it with the port number of the logging server. The program then 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.

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 a connection request 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_EVENT`s. Thus, when a client sends a logging record, the initiation dispatcher calls back to the client's logging handler to process the incoming record from that client connection in the logging server's single thread of control.

Known uses **InterViews** [LC87]. The Reactor pattern is implemented by the InterViews 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 [Sch97]. The ACE framework uses the Reactor pattern as its central event demultiplexer and dispatcher.

The **ORB Core** layer in many single-threaded implementations of CORBA [OMG98b], such as VisiBroker, Orbix, and TAO [POSA3], use the Reactor pattern to 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 [SchSu94] between PBXs and supervisors in Call Center Management system.

Project Spectrum. The high-speed medical image transfer subsystem of project Spectrum [PHS96] uses the Reactor pattern in a medical imaging system.

Consequences 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.

Modularity, reusability, and configurability of event-driven applications. The pattern decouples application functionality into several 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 development of generic event handler components, such as acceptors or connectors (129), that are loosely coupled together through a Reactor. This separation of concerns helps promote greater software reuse. For instance, the logging server's `Acceptor` class can establish connections for many different connection-oriented services, such as file transfer, remote login, and video-on-demand. As a result modifying or extending the functionality of the logging server only affects the implementation of the logging handler class.

Portability. The initiation dispatcher's interface can be reused independently of the operating 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` [Ste90]. In the Win32 API, event demultiplexing is performed by the `WaitForMultipleObjects` system [Cus93].

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.

The Reactor pattern has the following **liabilities**:

Restricted applicability. The Reactor pattern can only be applied efficiently if the operating system supports handles. It is possible to emulate the semantics of the Reactor pattern using multiple threads within the initiation dispatcher, for example, one thread for each handle. Whenever there are events available on a handle, its associated thread will read the event and place it on a queue that is processed sequentially by the initiation dispatcher. However, this design is typ-

ically very inefficient since it serializes all the 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 impede the responsiveness for clients connected to other handles. Therefore, for long-duration operations, such as transferring multi-megabyte medical images [PHS96], the Active Object pattern (89) 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 run-time 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.

See Also The Reactor pattern is related to the Observer pattern [GHJV95], 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 pattern [GHJV95], where a request is delegated to the responsible service provider. The Reactor pattern differs from the Chain of Responsibility since the Reactor associates a specific event handler with a particular

source of events, whereas the Chain of Responsibility 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 (85). 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 (89) 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 for event demultiplexing. A Facade [GHJV95] is an interface that shields applications from complex object relationships within a subsystem

Credits John Vlissides, the shepherd of the [PLoP94] version of Reactor, and Doug Lea provided many useful suggestions for documenting the original Reactor concept in pattern form..