**Reactor**

An Object Behavioral Pattern for

Demultiplexing and Dispatching Handles for Synchronous Events

Douglas C. Schmidt

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.

CLIENT

DATABASE

**1 Intent**

The Reactor design pattern handles service requests that are delivered concurrently to an application by one or more clients. Each service in an application may consist of serveral methods and is represented by a separate event han-

CONNECTION REQUEST

NETWORK

LOGGING SERVER

SOCKET HANDLES

PRINTER

CONSOLE

dler that is responsible for dispatching service-specific re- quests. Dispatching of event handlers is performed by an ini- tiation dispatcher, which manages the registered event han- dlers. Demultiplexing of service requests is performed by a synchronous event demultiplexer.

CLIENT

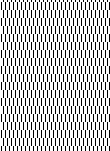
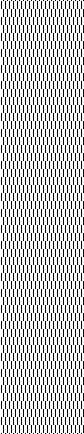
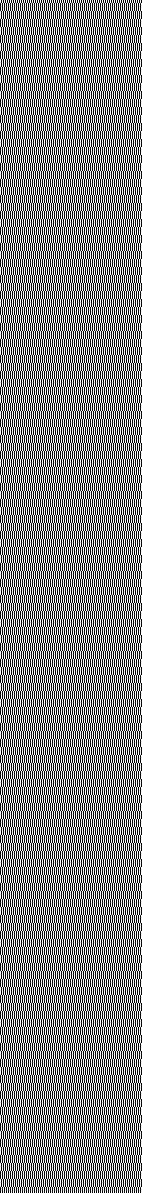
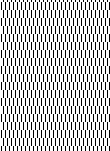
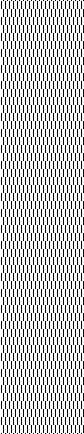
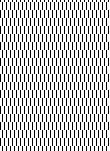
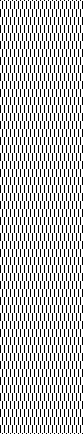
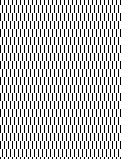
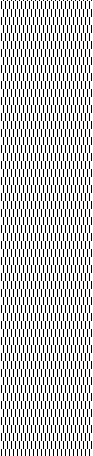
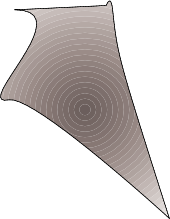
LOGGING RECORDS

CLIENT

LOGGING RECORDS

SERVER

**2 Also Known As**



Dispatcher, Notifier

**3 Example**

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 informa- tion about their status in a distributed environment. This sta- tus information commonly includes error notifications, de- bugging 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.

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 *handles*. A handle identifies network communica- tion resources managed within an OS.

The logging server communicates with clients using a connection-oriented protocol, such as TCP [1]. Clients that want to log data must first send a connection request to the

This work was supported in part by a grant by Siemens.

Figure 1: Distributed Logging Service

server. The server waits for these connection requests using a *handle factory* that listens on an address known to clients. When a connection request arrives, the handle factory es- tablishes a connection between the client and the server by creating a new handle that represents an endpoint of the con- nection. This handle is returned to the server, which then waits for client service requests to arrive on the handle. Once clients are connected, they can send logging records concur- rently to the server. The server receives these records via the connected socket handles.

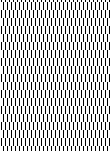
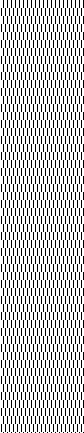
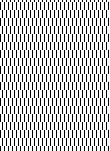
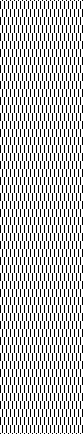
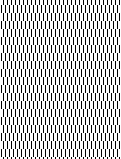
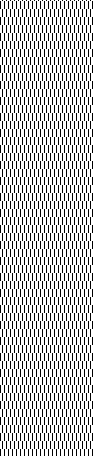
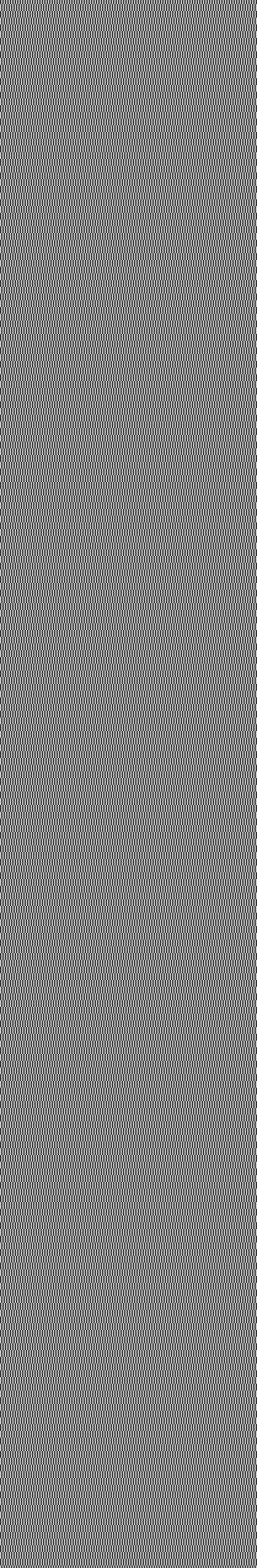
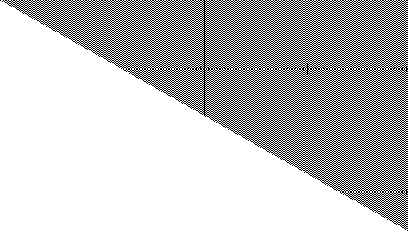
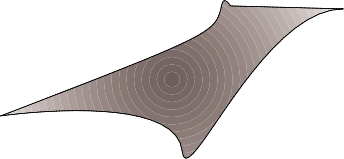
Perhaps the most intuitive way to develop a concurrent logging server is to use multiple threads that can process multiple clients concurrently, as shown in Figure 2. This approach synchronously accepts network connections and spawns a “thread-per-connection” to handle client logging records.

However, using multi-threading to implement the process- ing of logging records in the server fails to resolve the fol- lowing forces:

**Efficiency:** Threading may lead to poor performance due to context switching, synchronization, and data movement [2];

•

**Adaptability:** Integrating new or improved services,



•

LOGGING SERVER

such as changing message formats or adding server-side caching, should incur minimal modifications and mainte-

SERVER

THREAD1

1: accept ()

3: create()

THREAD2 THREAD3

5: recv()

6: write()

nance costs for existing code. For instance, implementing new application services should not require modifications to the generic event demultiplexing and dispatching mech- anisms.

**Portability:** Porting a server to a new OS platform

•

2: ccnnect()

CLIENT A

Logging

Acceptor

4: send()

Logging

Handler

NETWORK

Logging

Handler

CLIENT B

should not require significant effort.

**6 Solution**

Integrate the synchronous demultiplexing of events and the dispatching of their corresponding event handlers that pro- cess the events. In addition, decouple the application- specific dispatching and implementation of services from the general-purpose event demultiplexing and dispatching

Figure 2: Multi-threaded Logging Server

**Programming simplicity:** Threading may require com- plex concurrency control schemes;

•

**Portability:** Threading is not available on all OS plat-

mechanisms.

For each service the application offers, introduce a sep- arate Event Handler that processes certain types of events. All Event Handlers implement the same inter- face. Event Handlers register with an Initiation Dispatcher, which uses a Synchronous Event

•

forms.

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.

**4 Context**

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

**5 Problem**

Demultiplexer to wait for events to occur. When events

occur, the Synchronous Event Demultiplexer

notifies the Initiation Dispatcher, which syn- chronously 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.

**7 Structure**

The key participants in the Reactor pattern include the fol- lowing:

**Handles（能够引发事件的资源句柄）**

Identify resources that are managed by an OS.

•

Server applications in a distributed system must handle mul- tiple clients that send them service requests. Before invok- ing a specific service, however, the server application must demultiplex and dispatch each incoming request to its corre- sponding service provider. Developing an effective server mechanisms for demultiplexing and dispatching client re- quests requires the resolution of the following forces:

**Availability:** The server must be available to handle in-

These resources commonly include network connec- tions, open files, timers, synchronization objects, etc. 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 in- terested in are *connection* events and *read* events, which represent incoming client connections and logging data, respectively. The logging server maintains a separate

•

coming requests even if it is waiting for other requests to ar- rive. In particular, a server must not block indefinitely han- dling any single source of events at the exclusion of other event sources since this may significantly delay the respon- seness to other clients.

**Efficiency:** A server must minimize latency, maximize

connection for each client. Every connection is repre- sented in the server by a socket handle.

**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 demulti-

•

•

throughput, and avoid utilizing the CPU(s) unnecessarily.

**Programming simplicity:** The design of a server should

plexer for I/O events is select [1], which is an event demultiplexing system call provided by the UNIX and

Win32 OS platforms. The select call indicates which

•

simplify the use of suitable concurrency strategies.

Handles can have operations invoked on them syn- chronously without blocking the application process.

**Initiation Dispatcher**

Defines an interface for registering, removing, and dispatching Event Handlers. Ultimately, the Synchronous Event Demultiplexer is respon- sible for waiting until new events occur. When it detects new events, it informs the Initiation Dispatcher to call back application-specific event handlers. Common events include connection accep- tance events, data input and output events, and timeout events.

•

**Event Handler**

Specifies an interface consisting of a hook method [3]

•

that abstractly represents the dispatching operation for service-specific events. This method must be imple- mented by application-specific services.

**Concrete Event Handler**

Implements the hook method, as well as the meth-

•

ods to process these events in an application-specific manner. Applications register Concrete Event Handlers with the Initiation Dispatcher to process certain types of events. When these events ar- rive, the Initiation Dispatcher calls back the hook method of the appropriate Concrete Event Handler.

There are two Concrete Event Handlers in the logging server: Logging Handler and Logging Acceptor. The Logging Handler is responsi- ble for receiving and processing logging records. The Logging Acceptor creates and connects Logging Handlers that process subsequent logging records from clients.

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

**8 Dynamics**

**8.1 General Collaborations**

The following collaborations occur in the Reactor pattern:

When an application registers a Concrete Event Handler 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 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 OS.

•

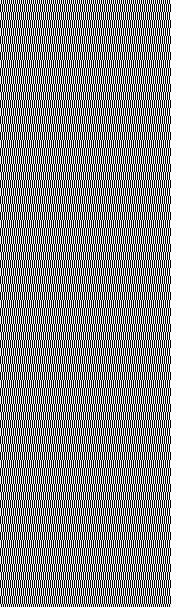
After all Event Handlers are registered, an applica- tion calls handle events to start the Initiation Dispatcher’s event loop. At this point, the Initiation Dispatcher combines the Handle from each registered Event Handler and uses the Synchronous Event Demultiplexer to wait for events to occur on these Handles. For in- stance, the TCP protocol layer uses the select syn- chronous event demultiplexing operation to wait for client logging record events to arrive on connected socket Handles.

•

The Synchronous Event Demultiplexer no- tifies the Initiation Dispatcher when a Handle corresponding to an event source becomes “ready,” *e.g.*, that a TCP socket is “ready for reading.”

•

The Initiation Dispatcher triggers Event Handler hook method in response to events on the ready Handles. When events occur, the Initiation Dispatcher uses the Handles ac- tivated by the event sources as “keys” to locate and dispatch the appropriate Event Handler’s hook method.



•

Initiation Dispatcher handle\_events() register\_handler(h) remove\_handler(h)

handlers

N

select (handlers);

foreach h in handlers loop

h.handle\_event(type)

end loop

Event Handler

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

•

uses

Handle owns

notifies

handle\_event(type)

get\_handle()

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 Section 9.4.

Synchronous Event

Demultiplexer

select()

Concrete Event Handler

The following interaction diagram illustrates the collabo- ration between application code and participants in the Re- actor pattern:

main program

EVENT HANDLING INITIALIZATION

callback : Concrete Event\_Handler

Initiation

Dispatcher

Handles

7. The Logging Acceptor creates (7) a Logging

Handler to service the new client;

8. Logging Handler registers (8) its socket handle

INITIALIZE REGISTER HANDLER EXTRACT HANDLE

MODE

RUN EVENT LOOP

Initiaticn\_Dispatcher() register\_handler(callback, event\_type) get\_handle()

handle\_events()

with the Initiation Dispatcher and instructs the dispatcher to notify it when the socket becomes “ready for reading.”

WAIT FOR EVENTS DISPATCH

MODE

HANDLER(S)

handle\_event(event\_type)

select()

**8.2.2 Client Sends Logging Record to a Reactive Log- ging Server**

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

**8.2 Collaboration Scenarios**

The collaborations within the Reactor pattern for the logging server can be illustrated with two scenarios. These scenarios

SERVER LOGGING SERVER Logging Handler

show how a logging server designed using reactive event dis-

patching handles connection requests and logging data from

2: handle

event()

for A

3: recv()

multiple clients.

**8.2.1 Client Connects to a Reactive Logging Server**

NETWORK

CLIENT

5: return

Initiation

Dispatcher

4: write()

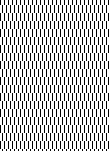
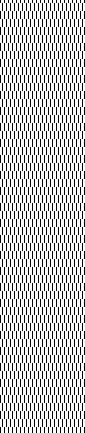
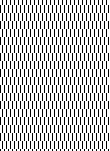
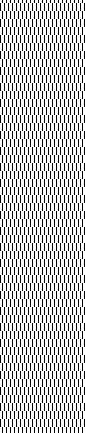
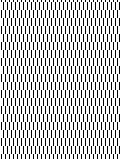
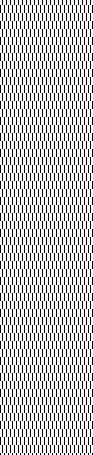
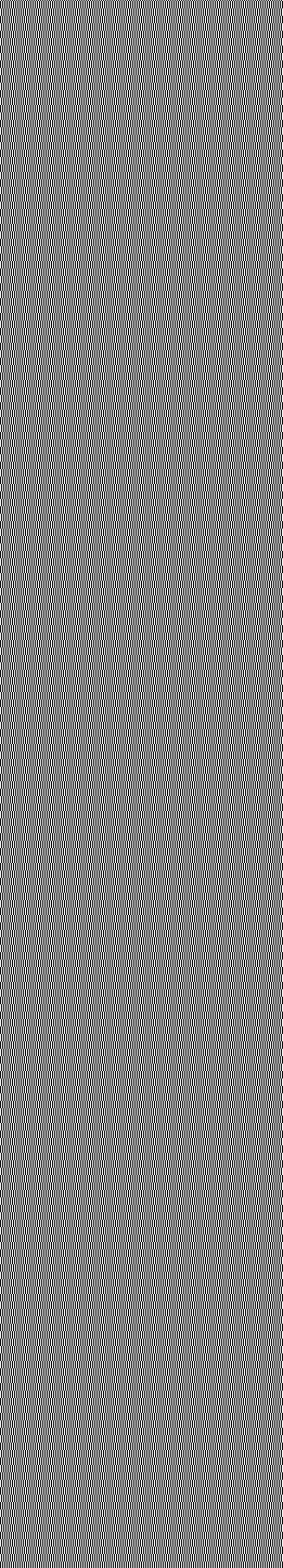
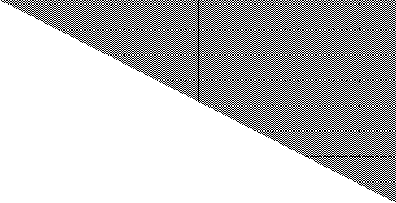
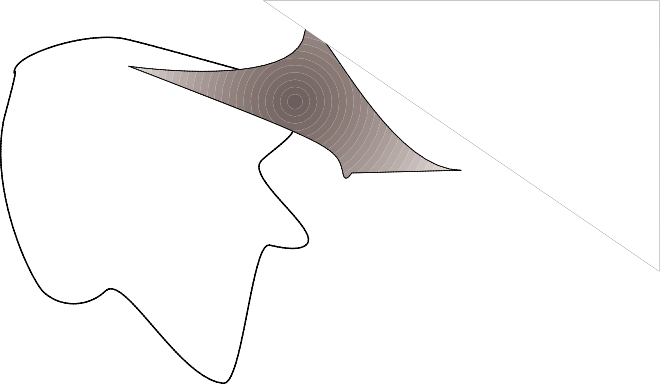
Logging Handler for B

The first scenario shows the steps taken when a client con- nects to the logging server.

A 1: send()

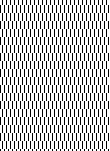
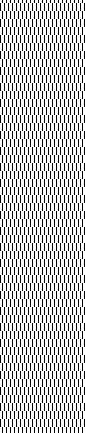
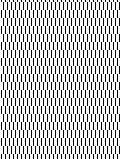
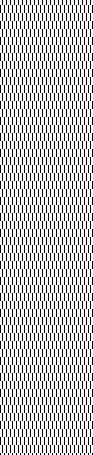
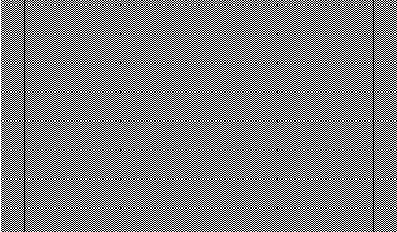
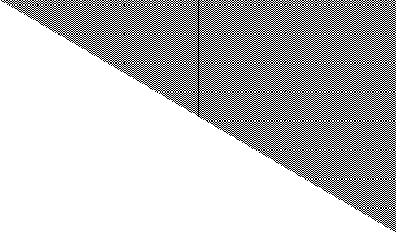
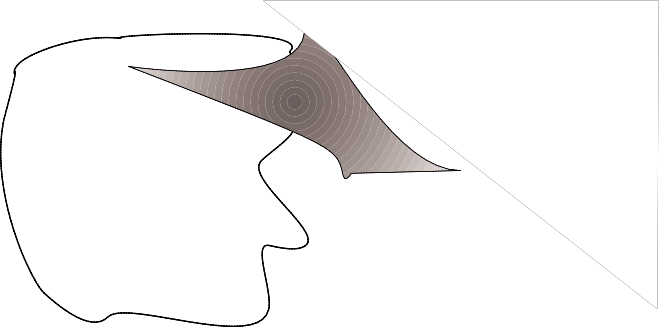
CLIENT B

SERVER LOGGING SERVER



Logging

This sequence of steps can be summarized as follows:



1: register handler()

Acceptor

6: accept()

7: create()

1. The client sends (1) a logging record;

2. The Initiation Dispatcher notifies (2) the as-

NETWORK

Initiation

Dispatcher

5: handle

event()

Logging

Handler

8: register

sociated Logging Handler when a client logging record is queued on its socket handle by OS;

CLIENT

4: ccnnect()

2: handle\_events()

3: select()

handler()

3. The record is received (3) in a non-blocking manner

(steps 2 and 3 repeat until the logging record has been received completely);

4. The Logging Handler processes the logging

This sequence of steps can be summarized as follows:

1. The logging server (1) registers the Logging Acceptor with the Initiation Dispatcher to handle connection requests;

2. The logging server invokes the handle events

method (2) of the Initiation Dispatcher;

3. The Initiation Dispatcher invokes the syn- chronous event demultiplexing select (3) operation to wait for connection requests or logging data to arrive;

4. A client connects (4) to the logging server;

5. The Logging Acceptor is no- tified by the Initiation Dispatcher (5) of the new connection request;

6. The Logging Acceptor accepts (6) the new con- nection;

record and writes (4) it to the standard output.

5. The Logging Handler returns (5) control to the

Initiation Dispatcher’s event loop.

**9 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 communi- cation software framework [2].

**9.1 Select the Synchronous Event Demulti- plexer Mechanism**

The Initiation Dispatcher uses a Synchronous

Event Demultiplexer to wait synchronously until one

or more events occur. This is commonly implemented us- ing 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 OS process in which the application-specific service handlers reside. In general, the Synchronous Event Demultiplexer is based upon existing OS mechanisms, rather than devel- oped by implementers of the Reactor pattern.

**9.2 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 provides methods to register and remove the handlers from this table at run-time. This table can be implemented in var- ious ways, *e.g.*, using hashing, linear search, or direct index- ing if handles are represented as a contiguous range of small integral values.

**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. 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 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 = 020,

CLOSE\_EVENT = 040

};

class Initiation\_Dispatcher

// = TITLE

// Demultiplex and dispatch Event\_Handlers

// in response to client requests.

{

public:

// Register an Event\_Handler of a particular

// Event\_Type (e.g., READ\_EVENT, ACCEPT\_EVENT,

// etc.).

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 synchroniza- tion. In this case, the Initiation Dispatcher serial- izes 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 applica- tions. In this case, critical sections within the Initiation Dispatcher must be serialized to prevent race conditions when modifying or activating shared state variables (such as the table holding the Event Handlers). A common tech- nique for preventing race conditions uses mutual exclusion mechanisms like semaphores or mutex variables.

To prevent self-deadlock, mutual exclusion mechanisms can use *recursive locks* [4]. Recursive locks hold 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 Event Handlers. Application hook method code may subsequently re-enter the Initiation Dispatcher via its register handler and remove handler methods.

**9.3 Determine the Type of the Dispatching**

**Target**

Two different types of Event Handlers can be as- sociated with a Handle to serve as the target of an Initiation Dispatcher’s dispatching logic. Imple- mentations of the Reactor pattern can choose 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 imple- mentation shown in Section 7 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 into a single component.

**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 call- backs without having to define a new class that inherits from Event Handler.

The Adapter pattern [5] 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.

**9.4 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 Sec- tion 7 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 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

// = TITLE

// Abstract base class that serves as the

// target of the Initiation\_Dispatcher.

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 selectively override methods in the base class and avoid further demultiplexing, *e.g.,* via switch or if state- ments, in the hook method. However, it requires the frame- work developer to anticipate the set of Event Handler methods in advance. For instance, the various handle \* methods in the Event Handler interface above are tai- lored 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 [6].

Both approaches described above are examples of the hook method pattern described in [3] and the Factory Call- back pattern described in [7]. The intent of these patterns is to provide well-defined hooks that can be specialized by ap- plications and called back by lower-level dispatching code.

**9.5 Determine the Number of Initiation Dis- patchers in an Application**

Many applications can be structured using just one instance of the Reactor pattern. In this case, the Initiation

{

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 in- terface. 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:

Dispatcher can be implemented as a Singleton [5]. 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 neces- sary 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 syn- chronization mechanisms if Event Handlers in different threads access shared state.

**9.6 Implement the Concrete Event Handlers**

The concrete event handlers are typically created by appli- cation developers to perform specific services in response to

particular events. The developers must determine what pro- cessing to perform when the corresponding hook method is invoked by the initiation dispatcher.

The following code implements the Concrete Event Handlers for the logging server described in Section 3. 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 component of the Acceptor-Connector pattern [8]. The Acceptor-Connector 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 connec- tions 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 be- low:

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 reg- ister 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 log- ging 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 [9]. These socket wrappers encapsulate the SOCK Stream semantics of the socket in- terface within a portable and type-secure object-oriented in- terface. In the Internet domain, SOCK Stream sockets are implemented using TCP.

The constructor for the Logging Acceptor registers itself with the Initiation Dispatcher Singleton [5] for ACCE PT 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, 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 dy- namically to process the logging requests. As shown below, the Logging Handler registers itself with the Initiation Dispatcher, which will 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 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 RE AD 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;

**9.7 Implement the Server**

The logging server contains a single main function.

**The logging server main function:** This function imple- ments 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 ac- cept 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;

}

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

The following interaction diagram illustrates the collabo-

peer\_stream\_.recv ((void \*) log\_record, sizeofratlioongb\_ertewceoerndth)e;objects participating in the logging server

// Write logging record to standard output. log\_record.write (STDOUT);

}

example:

else if (et == CLOSE\_EVENT) {

peer\_stream\_.close ();

delete (void \*) this;

Logging

Server

la : Logging Acceptor

lh : Logging Handler

Initiation

Dispatcher

Handles

} INITIALIZE

} REGISTER HANDLER

FOR ACCEPTS

EXTRACT HANDLE

Initiaticn\_Dispatcher()

register\_handler(la, ACCEPT\_EVENT)

get\_handle()

When a RE AD event occurs on a socket Handle,

the Initiation Dispatcher calls back to the

handle event method of the Logging Handler. This method receives, processes, and writes the logging

record to the standard output (ST DOUT). Likewise, when the client closes down the connection the Initiation Dispatcher passes a CL OSE event, which informs the Logging Handler to shut down its SOCK Stream and delete itself.

START EVENT LOOP FOREACH EVENT DO

CONNECTION EVENT

ACCEPT AND CREATE HANDLER

REGISTER HANDLER FOR INPUT

EXTRACT HANDLE LOGGING RECORD

handle\_events()

handle\_event(ACCEPT\_EVENT)

scck = acceptcr\_.accept()

lh = new Lcgging\_Acceptcr (scck);

register\_handler(lh, READ\_EVENT)

get\_handle()

handle\_event(READ\_EVENT)

select()

Once the Initiation Dispatcher object is initial- ized, 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 RE AD events. Thus, when a client sends a logging record, the Initiation Dispatcher calls back to the client’s Logging Handler to process the in- coming record from that client connection in the logging server’s single thread of control.

**10 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 [10] 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 [11] uses the Reactor pattern as its central event demultiplexer and dis- patcher.

•

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

**CORBA ORBs:** The ORB Core layer in many single- threaded implementations of CORBA [12] (such as VisiBro- ker, Orbix, and TAO [13]) use the Reactor pattern to demul- tiplex 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 [14] between PBXs and supervisors in a Call Center Management system.

•

**Project Spectrum:** The high-speed medical image trans-

**Separation of concerns:** The Reactor pattern decou- ples application-independent demultiplexing and dispatch- ing mechanisms from application-specific hook method functionality. The application-independent mechanisms be- come 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 par- ticular type of service.

**Improve modularity, reusability, and configurability of event-driven applications:** The pattern decouples appli- cation functionality into separate classes. For instance, there are two separate classes in the logging server: one for es- tablishing connections and another for receiving and pro- cessing 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, modifying or ex- tending the functionality of the logging server only affects the implementation of the logging handler class.

**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 multi- ple sources of events. Common sources of events may in- clude I/O handles, timers, and synchronization objects. On UNIX platforms, the event demultiplexing system calls are called select and poll [1]. In the Win32 API [16], the WaitForMultipleObjects system call performs event demultiplexing.

**Provides coarse-grained concurrency control:** The Re- actor 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 com- plicated synchronization or locking within an application process.

**11.2 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 pos-

•

fer subsystem of project Spectrum [15] uses the Reactor pat- tern in a medical imaging system.

**11 Consequences**

**11.1 Benefits**

The Reactor pattern offers the following benefits:

sible to emulate the semantics of the Reactor pattern using

multiple threads within the Initiation Dispatcher, *e.g.* 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 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 pro- cess, 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 respon- siveness for clients connected to other Handles. There- fore, for long-duration operations, such as transferring multi- megabyte medical images [15], the Active Object pattern [17] 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 pat- tern can be hard to debug since the inverted flow of con- trol oscillates between the framework infrastructure and the method callbacks on application-specific handlers. This in- creases the difficulty of “single-stepping” through the run- time behavior of a framework within a debugger since appli- cation 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 writ- ten 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 re- turns to the generated Deterministic Finite Automata (DFA)

skeleton, however, it is hard to follow the program logic.

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 Fa- cade [5] for event demultiplexing. A Facade is an interface that shields applications from complex object relationships within a subsystem.

**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* th *USENIX C++ Technical Conference*, (Cambridge, Mas-

s6achusetts), USENIX Association, April 1994.

[3] W. Pree, *Design Patterns for Object-Oriented Software De- velopment*. Reading, MA: Addison-Wesley, 1994.

[4] 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.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Pat- terns: Elements of Reusable Object-Oriented Software*. Read- ing, MA: Addison-Wesley, 1995.

[6] D. C. Schmidt and P. Stephenson, “Experiences Using Design

Patterns to Evolve System Software Across Diverse OS Plat-

forms,” in *Proceedings of the*

th *European Conference on*

**12 See Also**

The Reactor pattern is related to the Observer pattern [5], 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 demul- tiplex 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 Responsibil- ity (CoR) pattern [5], where a request is delegated to the re- sponsible 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

*Object-Oriented Programming*,g (Aarhus, Denmark), ACM,

August 1995.

[7] S. Berczuk, “A Pattern for Separating Assembly and Process- ing,” in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley,

1995.

[8] 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.

[9] 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* nd *Conference on Object-Oriented Technologies and*

*System*2*s*, (Toronto, Canada), USENIX, June 1996.

[10] M. A. Linton and P. R. Calder, “The Design and Implemen- tation of InterViews,” in *Proceedings of the USENIX C++ Workshop*, November 1987.

[11] D. C. Schmidt, “The ACE Framework.” Available from

Handler.

[www.cs.wustl.edu/](http://www.cs.wustl.edu/)

schmidt/ACE.html, 1997.

The Reactor pattern can be considered a *synchronous* vari- ant of the asynchronous Proactor pattern [18]. The Proac- tor supports the demultiplexing and dispatching of multiple event handlers that are triggered by the *completion* of *asyn- chronous* events. In contrast, the Reactor pattern is respon- sible 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 [17] decouples method execu- tion 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

[12] Object Management Group, *The Common Object Request*

*Broker: Architecture and Specification*, 2.0 ed., July 1995.

[13] D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Brokers,” *Com- puter Communications*, vol. 21, pp. 294–324, Apr. 1998.

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

[15] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High- Performance Electronic Medical Imaging,” *USENIX Comput- ing Systems*, vol. 9, November/December 1996.

[16] H. Custer, *Inside Windows NT*. Redmond, Washington: Mi- crosoft Press, 1993.

[17] R. G. Lavender and D. C. Schmidt, “Active Object: an Object

Behavioral Pattern for Concurrent Programming,” in *Proceed-*

*ings of the*

nd *Annual Conference on the Pattern Languages*

*of Program*2*s*, (Monticello, Illinois), pp. 1–7, September 1995.

[18] T. Harrison, I. Pyarali, D. C. Schmidt, and T. Jordan, “Proac- tor – An Object Behavioral Pattern for Dispatching Asyn-

chronous Event Handlers,” in *The*

th *Pattern Languages of*

*Programming Conference (Washing* *ton University technical report #WUCS-97-34)*, September 1997.