Patterns in C – Part 5: REACTOR

By Adam Petersen <adampetersen75@yahoo.se>

This final part of the series will step outside the domain of standard C and investigate a pattern for event-driven applications. The REACTOR pattern decouples different responsibilities and allows applications to demultiplex and dispatch events from potentially many clients.

The case of many Clients

In order to simplify maintenance of large systems, the diagnostics of the individual subsystems are gathered in one, central unit. Each subsystem connects to the diagnostics server using TCP/IP. As TCP/IP is a connection-oriented protocol, the clients (the different subsystems) have to request a connection at the server. Once a connection is established, a client may send diagnostics messages at any time.

The brute-force approach is to scan for connection requests and diagnostics messages from the clients one by one as illustrated in the activity diagram in Figure 1.

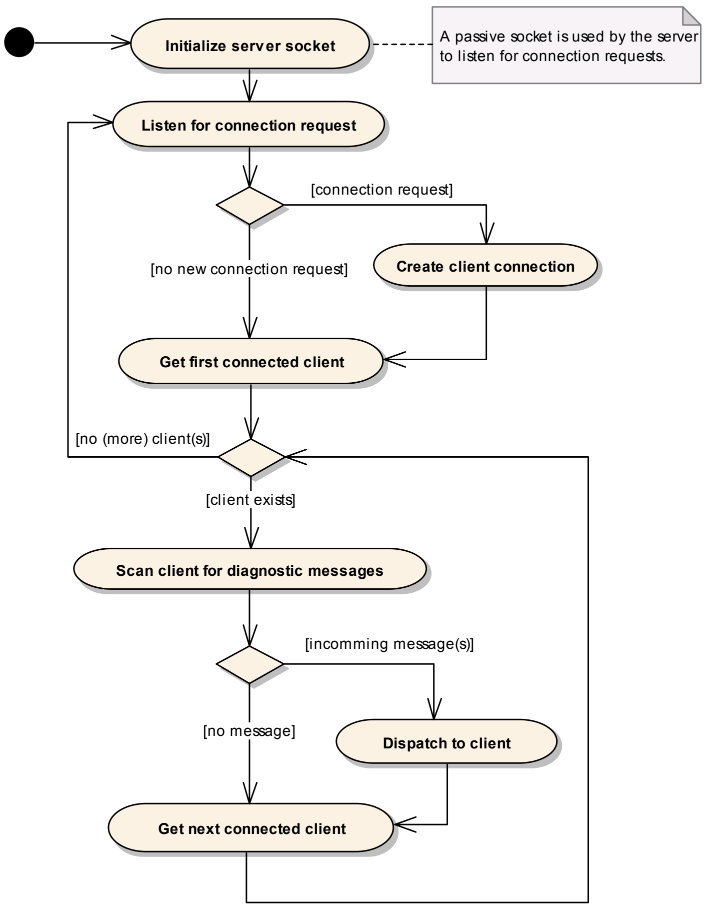
Initialize server socket
A passive socket is used by the server to listen for connection requests.
Listen for connection request
[connection request]
[no new connection request]
Create client connection
Get first connected client
[no (more) client(s)]
[client exists]
Scan client for diagnostic messages
[incomming message(s)]
[no message]
Dispatch to client
Get next connected client

Figure 1: Eternal loop to scan for different events

Even in this strongly simplified example, there are several potential problems. By intertwining the application logic with the networking code and the code for event dispatching, several unrelated responsibilities have been built into one module. Such a design is likely to lead to serious maintenance, testability, and scalability problems by violating a fundamental design principle.

The Single Responsibility Principle

The single responsibility principle states that "a class should have only one reason to change" [1]. The goal of this principle is close to that of the open-closed principle treated below: both strive to protect existing code from modifications. When violating the single responsibility principle, a module gets more reasons to change and modifications to it become more likely. Worse, the different responsibilities absorbed by a single module may become coupled and interact with each other making modifications and testing of the module more complicated.

The single responsibility principle is basically about cohesion. It is useful and valuable on many levels of abstraction, not at least in a procedural context; simply replacing the word "class" with "function" enables us to analyze algorithms like the one above with respect to this principle.

Violation of the Open-Closed Principle

By violating the single responsibility principle, the module in the example above will be hard to maintain; it is code that one never wants to dig into in the future. Unfortunately, on collision course with that wish is the fact that the event loop above violates the open-closed principle [1]; new functionality cannot be added without modifying existing code. Related to our example, a diagnostics server typically allows a technician to connect and query stored information. Introducing that functionality would double the logic in the event loop. Clearly, this design makes the code expensive to modify.

From a Performance Perspective

To make things worse, the solution above fails to scale in terms of performance as well. As all events are scanned serially, even in case timeouts are used, valuable time is wasted doing nothing.

The potential performance problem above may be derived from the failure of taking the concurrent nature of the problem into account. One approach to address this problem is by introducing multiple threads. The diagnostics server keeps its event loop, but the loop is now reduced to scan for connection requests. As soon as a connection is requested, a new thread is allocated exclusively for handling messages on that new connection.

The multithreading approach fails to address the overall design issue as it still violates both the single responsibility principle and the open-closed principle. Although the code for scanning and dispatching diagnostics messages is moved out of the event loop, adding a new server-port still requires modifications to existing code.

From a design perspective threads didn't improve anything. In fact, even with respect to performance, this solution may due to context switches and synchronization actually perform worse than the initial single-threaded approach.

The sheer complexity inherent in designing and implementing multithreaded applications is a further argument for discarding this solution.

Problem Summary

Summarizing the experience so far, the design fails as it assumes three different responsibilities. This problem is bound to be worse as the design violates the open-closed principle, making modifications to existing code more likely.

Summarizing the ideal solution, it should scale well, encapsulate and decouple the different responsibilities, and be able to serve multiple clients simultaneously without introducing the liabilities of multithreading. The REACTOR pattern realizes this solution by encapsulating each service of the application logic in event handlers and separating out the code for event demultiplexing.

The REACTOR Pattern

The intent of the REACTOR pattern is: "The REACTOR architectural pattern allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients" [2].

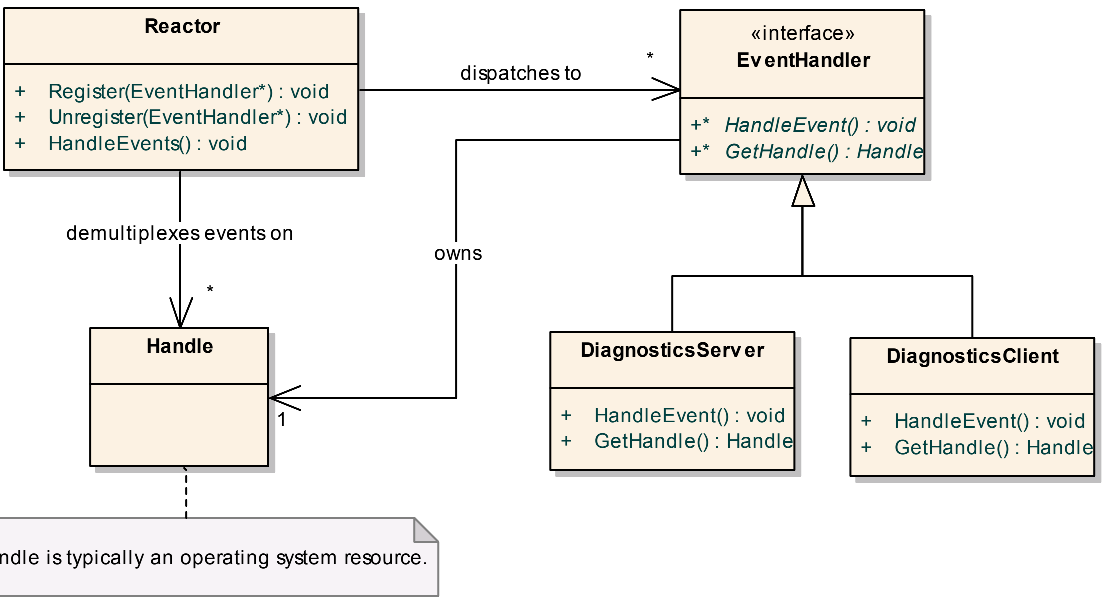

Figure 2: Structure of the REACTOR pattern

The roles of the involved participants are:

- **`EventHandler`**: An **`EventHandler`** defines an interface to be implemented by modules reacting to events. Each **`EventHandler`** own its own **`Handle`**.
- **`Handle`**: An efficient implementation of the REACTOR pattern requires an OS that supports handles (examples of **`Handles`** include system resources like files, sockets, and timers).
- **`DiagnosticsServer`** and **`DiagnosticsClient`**: These two are concrete event handlers, each one encapsulating one responsibility. In order to be able to receive event notifications, the concrete event handlers have to register themselves at the **`Reactor`**.
- **`Reactor`**: The **`Reactor`** maintains registrations of **`EventHandlers`** and fetches the associated **`Handles`**. The **`Reactor`** waits for events on its set of registered **`Handles`** and invokes the corresponding **`EventHandler`** as a **`Handle`** indicates an event.

Event Detection

In its description of REACTOR, *Pattern-Oriented Software Architecture* [2] defines a further collaborator, the **`Synchronous Event Demultiplexer`**. The **`Synchronous Event Demultiplexer`** is called by the **`Reactor`** in order to wait for events to occur on the registered **`Handles`**.

A synchronous event demultiplexer is often provided by the operating system. This example will use **`poll()`** (**`select()`** and Win32's **`WaitForMultipleObjects()`** are other functions available on common operating systems), which works with any descriptor.

The code interacting with **`poll()`** will only be provided as a sketch, because the POSIX specific details are outside the scope of this article. The complete sample code, used in this article, is available from my homepage [6].

Implementation Mechanism

The collaboration between an **`EventHandler`** and the **`Reactor`** is similar to the interaction between an observer and its subject in the design pattern OBSERVER [5]. This relationship between these two patterns indicates that the techniques used to realize OBSERVER in C [4] may serve equally well to implement the REACTOR.

In order to decouple the Reactor from its event handlers and still enable the Reactor to notify them, each concrete event handler must correspond to a unique instance. In our OBSERVER implementation, the FIRST-CLASS ADT pattern [3] was put to work to solve this problem. As all concrete event handlers have to be abstracted as one, general type realizing the **`EventHandler`** interface, **`void*`** is chosen as "the general type" to be registered at the Reactor (please refer to the previous part in this series - Reference [4] - for the rationale and technical reasons behind the **`void*`** abstraction). These decisions enable a common interface for all event handlers:

Listing 1 : Interface of the event handlers, `EventHandler.h`

```
/* The type of a handle is system specific – this example
uses UNIX I/O handles, which are plain integer values. */
typedef int Handle;

/* All interaction from Reactor to an event handler goes
through function pointers with the following signatures: */
typedef Handle (*getHandleFunc)(void* instance);
typedef void (*handleEventFunc)(void* instance);

typedef struct
{
  void* instance;
  getHandleFunc getHandle;
  handleEventFunc handleEvent;
} EventHandler;
```

Having this interface in place allows us to declare the registration functions of the Reactor.

Listing 2 : Registration interface of the Reactor, `Reactor.h`

```
#include "EventHandler.h"

void Register(EventHandler* handler);
void Unregister(EventHandler* handler);
```

The application specific services have to implement the **`EventHandler`** interface and register themselves using the interface above in order to be able to react to events.

Listing 3 : Implementation of a concrete event handler, `DiagnosticsServer.c`

```
#include "EventHandler.h"

struct DiagnosticsServer
{
   Handle listeningSocket;
   EventHandler eventHandler;

   /* Other attributes here... */
};

/* Implementation of the EventHandler interface. */
static Handle getServerSocket(void* instance)
{
   const DiagnosticsServerPtr server = instance;
   return server->listeningSocket;
}

static void handleConnectRequest(void* instance)
{
   DiagnosticsServerPtr server = instance;

   /* The server gets notified as a new connection
      request arrives. Add code for accepting the
      new connection and creating a client here... */
}

DiagnosticsServerPtr createServer(unsigned int tcpPort)
{
   DiagnosticsServerPtr newServer = malloc(sizeof *newServer);

   if(NULL != newServer) {
      /* Code for creating the server socket here.
         The real code should look for a failure, etc. */
      newServer->listeningSocket = createServerSocket(tcpPort);

      /* Successfully created -> register the listening socket. */
      newServer->eventHandler.instance = newServer;
      newServer->eventHandler.getHandle = getServerSocket;
      newServer->eventHandler.handleEvent = handleConnectRequest;

      Register(&newServer->eventHandler);
   }
   return newServer;
}

void destroyServer(DiagnosticsServerPtr server)
{
   /* Before deleting the server we have to unregister at the Reactor. */
   Unregister(&server->eventHandler);

   free(server);
}
```

Reactor Registration Strategy

When implementing the concrete event handlers as a FIRST-CLASS ADT, the functions for creating and destructing the ADT serves well to encapsulate the registration handling. The advantage is the combination of loose dependencies with information hiding as a client does not even have to know about the usage and interactions with the Reactor.

Another attractive property is that the internals of the server, in our example the handle, is encapsulated within the **`getServerSocket`** function. Sure, we are giving the Reactor a way to fetch it, but the Reactor is considered a well-trusted collaborator and we are actively giving it access by registering our event handler. There is no way for any other module to mistakenly fetch the handle and corrupt the associated resource.

Reactor Implementation

The details of the Reactor implementation are platform specific as they depend upon the available synchronous event demultiplexers. In case the operating system provides more than one synchronous event demultiplexer (e.g. **`select()`** and **`poll()`**), a concrete Reactor may be implemented for each one of them and the linker used to chose either one of them depending on the problem at hand. This technique is referred to as *link-time polymorphism*.

Each Reactor implementation has to decide upon the number of reactors required by the surrounding application. In the most common case, the application can be structured around one, single Reactor. In this case, the interface in Listing 2 (`Reactor.h`) will serve well. An application requiring more than one Reactor should consider making the Reactor itself a FIRST-CLASS ADT. This second variation complicates the clients slightly as references to the Reactor ADT have to be maintained and passed around in the system.

Independent of the system specific demultiplexing mechanism, a Reactor has to maintain a collection of registered, concrete event handlers. In its simplest form, this collection may simply be an array. This approach serves well in case the maximum number of clients is known in advance.

Listing 4: Implementation of a Reactor using `poll()`, `PollReactor.c`

```
#include "Reactor.h"
#include <poll.h>
/* Other include files omitted... */

/* Bind an event handler to the struct used to interface poll(). */
typedef struct
{
   EventHandler handler;
   struct pollfd fd;
} HandlerRegistration;

static HandlerRegistration registeredHandlers[MAX_NO_OF_HANDLES];

/* Add a copy of the given handler to the first free position in registeredHandlers. */
static void addToRegistry(EventHandler* handler);

/* Identify the event handler in the registeredHandlers and remove it. */
static void removeFromRegistry(EventHandler* handler);

/* Implementation of the Reactor interface used for registrations.*/

void Register(EventHandler* handler)
{
   assert(NULL != handler);
   addToRegistry(handler);
}

void Unregister(EventHandler* handler)
{
   assert(NULL != handler);
   removeFromRegistry(handler);
}
```

Invoking the Reactor

The reactive event loop is the core of the Reactor and its responsibilities are to control the demultiplexing and dispatch the detected events to the registered, concrete event handlers. The event loop is contained within the **`HandleEvents()`** function and is typically invoked from the **`main()`** function.

Listing 5: Client code driving the reactor

```
int main(void){
   const unsigned int serverPort = 0xC001;
   DiagnosticsServerPtr server = createServer(serverPort);

   if(NULL == server) {
      error("Failed to create the server");
   }

   /* Enter the eternal reactive event loop. */
   for(;;){
      HandleEvents();
   }
}
```

Before investigating the implementation, which include file should provide the declaration of the **`HandleEvents()`** function? Unfortunately, adding it to the file in Listing 2 (`Reactor.h`) would clearly make that interface less cohesive; the registration functions models a different responsibility than the event loop. A solution is to create a separate interface for the event loop. This interface is intended solely for the compilation unit invoking the event loop.

Listing 6: Interface to the event loop, `ReactorEventLoop.h`

```
void HandleEvents(void);
```

Despite its simplicity, this separation solves the cohesiveness problem and shields clients from functions they do not use. This technique of providing separate interfaces to separate clients is known as the interface-segregation principle [1].

Implementing the Event Loop

With the interface in place, we can move on and implement the event loop itself. The listing below extends Listing 4.

Listing 7: Example of a reactive event loop, `PollReactor.c`

```
#include “ReactorEventLoop.h”
/* The code from Listing 4 go here (omitted)... */

/* Add a copy of all registered handlers to the given array. */
static size_t buildPollArray(struct pollfd* fds);

/* Identify the event handler corresponding to the given descriptor in the registeredHandlers. */
static EventHandler* findHandler(int fd);

static void dispatchSignalledHandles(const struct pollfd* fds,
                                     size_t noOfHandles)
{
   /* Loop through all handles. Upon detection of a handle signalled by poll,
      its corresponding event handler is fetched and invoked. */
   size_t i = 0;

   for(i = 0; i < noOfHandles; ++i) {
      /* Detect all signalled handles and invoke their corresponding event handlers. */
      if((POLLRDNORM | POLLERR) & fds[i].revents) {
         EventHandler* signalledHandler = findHandler(fds[i].fd);

         if(NULL != signalledHandler){
            signalledHandler-> handleEvent(signalledHandler->instance);
         }
      }
   }
}

/* Implementation of the reactive event loop. */
void HandleEvents(void)
{
   /* Build the array required to interact with poll().*/
   struct pollfd fds[MAX_NO_OF_HANDLES] = {0};

   const size_t noOfHandles = buildPollArray(fds);

   /* Inoke the synchronous event demultiplexer.*/
   if(0 < poll(fds, noOfHandles, INFTIM)){
      /* Identify all signalled handles and invoke the event handler associated with each one. */
      dispatchSignalledHandles(fds, noOfHandles);
   }
   else{
      error("Poll failure");
   }
}
```

The example above lets each element in the collection maintain a binding between the registered event handler and the structure used to interact with **`poll()`**. One alternative approach is to keep two separate lists and ensure consistency between them. *Pattern-Oriented Software Architecture* [2] describes another, system specific alternative: in a UNIX implementation using **`select()`**, the “array is indexed by UNIX I/O handle values, which are unsigned integers ranging from 0 to FD_SETSIZE-1”.

Returning to the example, by grouping the registration and the poll-structure together, the array used to interact with **`poll()`** has to be built each time the reactive event loop is entered. In case the performance penalty is acceptable, this is probably a better choice as it enables a simpler handling of registrations and unregistrations during the event loop.

Handling new registrations

In my previous article [4], I discussed strategies for managing changed registrations during the notification of observers. The alternative of forbidding changed registrations is, unlike the OBSERVER pattern, not an option for a REACTOR implementation. In the example used in this article, the server reacts to the notification by creating a new client, which must be registered shall it ever be activated again. This leaves only one option for a REACTOR implementer: ensure that it works.

One solution is to maintain a separate array to interact with the synchronous event demultiplexer as illustrated above. This array is never modified in the event loop. However, this solution has the consequence that handles unregistered during the current event loop may be marked as signalled in the separate array. The code simply has to check for this case and ignore such handles, as illustrated by the function **`dispatchSignalledHandles`** in Listing 7 above.

The code uses the handle alone as identification. In cases resources are disposed and created during the same event loop, there is, depending on platform, a possibility that the handle ID's are re-used; a signalled handle in the copy may belong to an unregistered event handler, but due to a new registration using the re-cycled handle ID, the new event handler may be erroneously invoked. If this is an issue, the problem may be prevented by introducing further book-keeping data. For example, a second array containing the identities of the handles unregistered during the current event loop makes it possible to identify the case described above and thus avoid it.

More than one Type of Event

The design in the example above does only allow applications to register for one type of event (read-events). The event type is even hardcoded in the Reactor and it is a simple solution sufficient for applications without any need for further event detection. The REACTOR pattern, however, is not limited to one type of event. The pattern scales well to support different types of events.

Pattern-Oriented Software Architecture [2] describes two general strategies for dispatching event notifications:

- *Single-method interface*: An event handler is notified about all events through one, single function. The type of event (typically in the form of an **`enum`**) is passed as a parameter to the function. The disadvantage of this approach is that it sets the stage for conditional logic, which soon gets hard to maintain.
- *Multi-method interface*: In this case, the event handler declares separate functions for each supported event (e.g. **`handleRead`**, **`handleWrite`**, **`handleTimeout`**). As the Reactor has the knowledge of what event occurred, it invokes the corresponding function immediately, thus avoiding placing the burden on the event handler to re-create the event from a parameter.

Comparision of REACTOR and OBSERVER

Although the mechanisms used to implement them are related, there are differences between these two patterns. The main difference is in the notification mechanism. As a Subject changes its state in an OBSERVER implementation, all its dependents (observers) are notified. In a REACTOR implementation, this relationship is one to one – a detected event leads the **`Reactor`** to notify exactly one dependent (**`EventHandler`**).
One typical liability of the OBSERVER pattern is that the cohesion of the subject is lowered; besides serving its central purpose, a subject also takes on the responsibility of managing and notifying observers. With this respect, a Reactor differs significantly as its whole raison d'être is to dispatch events to its registered handlers.

Consequences

The main consequences of applying the REACTOR pattern are:

1. *The benefits of the single-responsibility principle*. Using the REACTOR pattern, each of the responsibilities above is encapsulated and decoupled from each other. The design results in increased cohesion, which simplifies maintenance and minimizes the risk of feature interaction.
   As the platform dependent code for event detection is decoupled from the application logic, unit testing is greatly simplified (it is straightforward to simulate events through the **`EventHandler`** interface).

2. *The benefits of the open-closed principle*. The design now adheres to the open-closed principle. New responsibilities, in the form of new event handlers, may be added without affecting the existing event handlers.

3. *Unified mechanism for event handling*. Even if the REACTOR pattern is centred on handles, it may be extended for other tasks. *Pattern-Oriented Software Architecture* [2] describes different strategies for integrating the demultiplexing of I/O events with timer handling. Extending the Reactor with timer support is an attractive alternative to typical platform specific solutions based upon signals or threads. This extension builds upon the possibility to specify a timeout value when invoking the synchronous event demultiplexer (for example, **`poll()`** allows a timeout to be specified with a resolution of milliseconds). Although it will possibly not suit a hard real-time system, a Reactor based timer mechanism is easier to implement and use than a signal or thread based solution as it tends to avoid re-entrance problems and race-conditions.

4. *Provides an alternative to multithreading*. Using the REACTOR pattern, blocking operations in the concrete event handlers can typically be avoided and consequently also multithreading. As discussed above, a multithreaded solution does not only add significant complexity; it may also prove to be less efficient in terms of run-time performance. However, as the Reactor implies a non pre-emptive multitasking model, each concrete event handler must ensure that it does not perform operations that may starve out other event handlers.

5. *Trades type-safety for flexibility.* All concrete event handlers are abstracted as **`void*`**. When converting a void-pointer back to a pointer of a concrete event handler type, the compiler doesn't have any way to detect an erroneous conversion This potential problem was faced in the implementation of the OBSERVER pattern [4] and the solution is the same for the REACTOR: define unique notification functions for each different type of event handler and bind the functions and event handler together using an **`EventHandler`** structure as described in Listing 1.

Summary

The REACTOR pattern simplifies event-driven applications by decoupling the different responsibilities, encapsulated in separate modules.

There is much more to the REACTOR pattern than described in this article. Particularly several variations that all come with different benefits and trade-offs. For an excellent in-depth treatment of the REACTOR and other patterns in the domain, I recommend the book *Pattern-Oriented Software Architecture, volume 2* [2].

References

1. Robert C. Martin: "*Agile Software Development*", Prentice Hall
2. Schmidt, Stal, Rohnert, Buschmann: "*Pattern-Oriented Software Architecture, volume 2*", Wiley
3. Adam Petersen, "*Patterns in C, part 1*", C Vu 17.1
4. Adam Petersen, "*Patterns in C, part 4: OBSERVER*", C Vu 17.4
5. Gamma, E., Helm, R., Johnson, R., and Vlissides, J, "*Design Patterns*", Addison-Wesley
6. The complete REACTOR sample code used in this article, `www.adampetersen.se`

Acknowledgements

Many thanks to Drago Krznaric and André Saitzkoff for their feedback.