1 Reactor

Handles service requests delivered concurrently by one or more inputs to handlers. The pattern consists of *event*, *event handler*, *demultiplexer* and *dispatcher*. Event holds its type and handle (socket, file descriptor, or similar). Event handler processes the given event. Demultiplexer monitors for events on the set of handles (for instance by calling function `select` on Unix systems). Dispatcher monitors a set of handles by demultiplexing them and calls the appropriate event handlers.

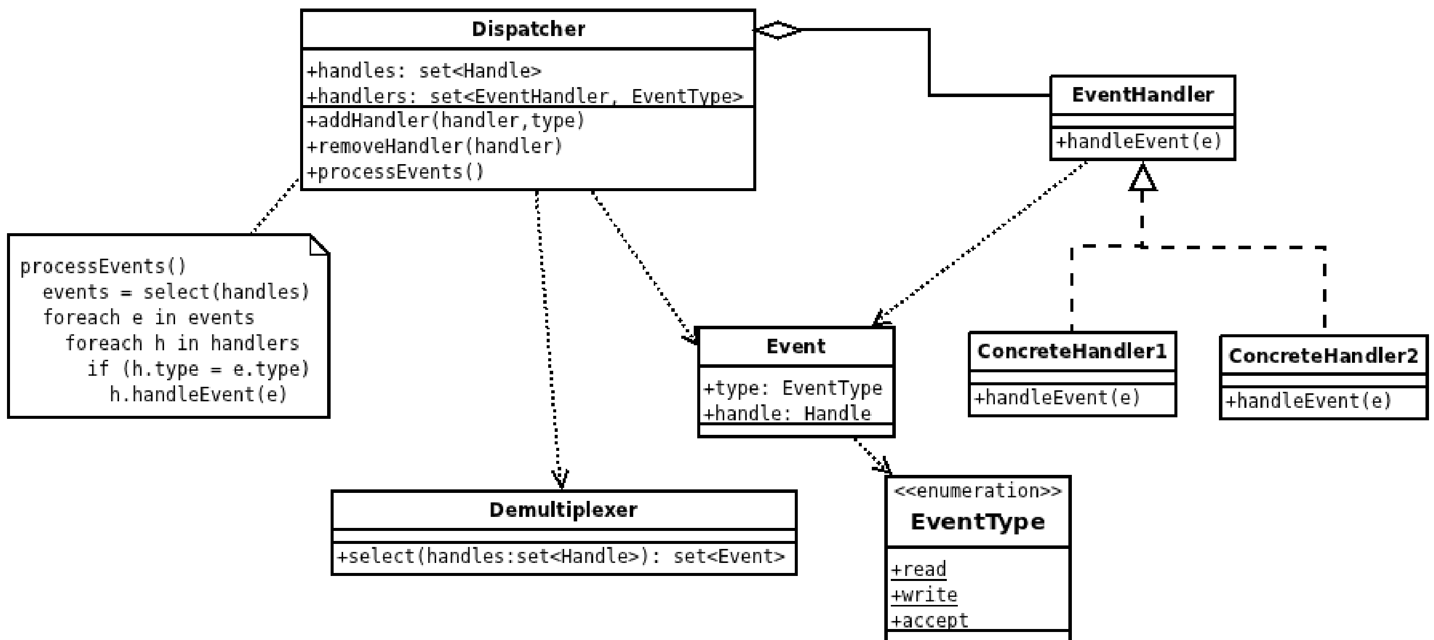

Figure 1: Reactor structure

Application gives to the dispatcher a list of event handlers for the specific event types (read, write, accept, etc.). Dispatcher uses the demultiplexer to get events for the existing handles. For each event it calls hook method of the registered handler for that event type.

1.1 C++ Poco

`SocketReactor` is the dispatcher. It requires two handlers - one for the server socket and other for the client socket. These handlers are managed by the reactor which notifies about read/write events. Read event in server handler means that a client socket can be accepted. Read/write events in client handler means that client socket can be read/written. Client handler is the concrete event handler in the pattern. Notifications about those events are sent via the observer pattern.

Client handler is notified on read/write events, so it can perform the appropriate action. Once the client socket is closed, it destroys event handlers, so reactor will not use them anymore.

Listing 1: Client handler

```
class ClientHandler
{
public:

    void onReadable(ReadableNotification* notification)
    {
        char buf[100];
```

```
        int no = _socket.receiveBytes(buf, 99);
        if (no > 0)
        {
            buf[no] = '\0';
            cout << "ClientHandler::onReadable(): buf=" << buf << endl;
            _request = buf;
        }
        else
        {
            // destroy handlers
            cout << "ClientHandler::onReadable(): destroying handlers" << endl;
            Observer<ClientHandler, ReadableNotification> readObserver(*this, &
               ClientHandler::onReadable);
            _reactor.removeEventHandler(_socket, readObserver);
            Observer<ClientHandler, WritableNotification> writeObserver(*this, &
               ClientHandler::onWritable);
            _reactor.removeEventHandler(_socket, writeObserver);
            _socket.close();
            delete this;
        }
    }

    void onWritable(WritableNotification* notification)
    {
        if (!_request.empty())
        {
            cout << "ClientHandler::onWritable(): _request=" << _request << endl;
            _socket.sendBytes(_request.c_str(), _request.length());
            _request.clear();
        }
    }
};
```

Server handler is notified on read events only, which means that a new socket has to be accepted. Once the client socket is accepted, two handlers are registered on the reactor.

Listing 2: Server handler

```
class ServerHandler
{
public:

    ServerHandler(const ServerSocket& socket, SocketReactor& reactor) : _reactor(
       reactor)
    {
        Observer<ServerHandler, ReadableNotification> readObserver(*this, &
           ServerHandler::onReadable);
        _reactor.addEventHandler(socket, readObserver);
    }

    void onReadable(ReadableNotification* notification)
    {
        ServerSocket server(notification->socket());
        StreamSocket client = server.acceptConnection();
        cout << "ServerHandler::onReadable(): accepted client from " << client.
           address().toString() << endl;

        ClientHandler* handler = new ClientHandler(_reactor, client);
        Observer<ClientHandler, ReadableNotification> readObserver(*handler, &
           ClientHandler::onReadable);
```

```
        _reactor.addEventHandler(client, readObserver);
        Observer<ClientHandler, WritableNotification> writeObserver(*handler, &
           ClientHandler::onWritable);
        _reactor.addEventHandler(client, writeObserver);
    }
};
```

Reactor is created and passed around to client/server handlers to be used for adding/removing event handlers.

Listing 3: Create reactor and connect to handlers

```
int main()
{
    ServerSocket socket(9000);
    SocketReactor reactor(Timespan(0, 0, 0, 1, 0));
    ServerHandler handler(socket, reactor);

    Thread thread;
    thread.start(reactor);
    thread.join();

    return EXIT_SUCCESS;
}
```

The source code is within `Reactor.cpp` file.

1.2 Java SDK

`Selector` class is Java's way for demultiplexing. It reads `SelectionKey`-s and determines which are ready to accept, read or write. For such purposes, two classes are created: `ServerHandler` to accept sockets on server side, and `ClientHandler` to deal with read/write operations, once the connection is established. Client handler is the concrete event handler in the pattern. `Reactor` performs dispatching of read/write events.

Client handler is notified on read/write events, so it can perform the appropriate action. Once the client socket is closed, the corresponding key is cancelled, so reactor will not use it anymore.

Listing 4: Client handler

```
class ClientHandler
{
    void onReadable(SelectionKey key) throws IOException
    {
        SocketChannel client = (SocketChannel)key.channel();
        final int BUFFER_SIZE = 32;
        this.request = ByteBuffer.allocate(BUFFER_SIZE);
        int result = client.read(request);
        if (result <= 0)
        {
            if (result == -1)
            {
                client.close();
                key.cancel();
                System.out.println("ClientHandler.onReadable(): key cancelled");
            }
        }
        else
```

```
        {
            request.flip();
            Charset charset = Charset.forName("ISO-8859-1");
            CharsetDecoder decoder = charset.newDecoder();
            CharBuffer charBuffer = decoder.decode(request);
            System.out.println("ClientHandler.onReadable(): request=" +
                charBuffer);
        }
    }

    void onWritable(SelectionKey key) throws IOException
    {
        SocketChannel client = (SocketChannel)key.channel();
        ByteBuffer response = this.request;
        if (response != null)
        {
            response.rewind();
            Charset charset = Charset.forName("ISO-8859-1");
            CharsetDecoder decoder = charset.newDecoder();
            CharBuffer charBuffer = decoder.decode(response);
            System.out.println("ClientHandler.onWritable(): response=" +
                charBuffer);
            response.rewind();
            client.write(response);
            this.request = null;
        }
    }
}
```

Server handler is notified on accept events only. Once the client socket is accepted, a handler is registered on the reactor.

Listing 5: Server handler

```
class ServerHandler
{
    void onAcceptable(SelectionKey key) throws IOException
    {
        SocketChannel client = this.server.accept();
        if (client != null)
        {
            System.out.println("ServerHandler.onAcceptable(): client accepted");
            client.configureBlocking(false);
            client.register(this.selector, SelectionKey.OP_READ | SelectionKey.
                OP_WRITE);
            client.keyFor(this.selector).attach(new ClientHandler());
        }
    }
}
```

Reactor monitors for read/write/accept events and dispatches them to handlers.

Listing 6: Create reactor and connect to handlers

```
class Reactor implements Runnable
{
    public static void main(String[] args) throws IOException,
        InterruptedException
    {
        Thread t = new Thread(new Reactor());
        t.start();
```

```
        t.join();
    }

    public Reactor() throws IOException, ClosedChannelException
    {
        this.server = ServerSocketChannel.open();
        this.server.socket().bind(new InetSocketAddress(9000));
        if (!server.socket().isBound())
            throw new IOException();
        this.server.configureBlocking(false);
        this.selector = Selector.open();
        this.server.register(selector, SelectionKey.OP_ACCEPT);
        this.serverHandler = new ServerHandler(this, this.server, this.selector);
    }

    public void run()
    {
        try
        {
            while (true)
            {
                this.selector.select();
                Iterator<SelectionKey> keyIter = selector.selectedKeys().iterator
                    ();
                while (keyIter.hasNext())
                {
                    SelectionKey key = keyIter.next();
                    if (key.isAcceptable())
                    {
                        this.serverHandler.onAcceptable(key);
                    }
                    if (key.isReadable() && key != this.server.keyFor(this.
                        selector))
                    {
                        ClientHandler h = (ClientHandler)key.attachment();
                        h.onReadable(key);
                    }
                    if (key.isValid() && key.isWritable())
                    {
                        ClientHandler h = (ClientHandler)key.attachment();
                        h.onWritable(key);
                    }
                }
            }
        }
        catch (IOException e)
        {
            System.out.println("main(): e=" + e);
        }
    }
}
```

The source code is within `Reactor.java` file.