AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Architecture and Design of Distributed Dependable Systems

Journal on Exercises Ex1-Ex3

Group 12

Authors:
Søren Outze Sørensen - 20091917
Daniela Popovici - 201503243
Troels Knudsen - 201270263

Supervisor:
Finn Overgaard Hansen

Revision History

Revision	Date/Authors	Description
1	20.09.2015/Troels	Document setup
2	20.09.2015/Daniela	1, 2.1, 2.3.1, 2.3.4, 2.4.1, 2.4.4, 4
3	20.09.2015/Søren	Diagrams
4	20.09.2015/Troels	Conclusion

1 Introduction

1.1 Intro to requirements for the exercises

The paper includes 3 exercises that apply the Wrapper Facade which is a service access and configuration pattern and the Reactor pattern which is an event handling pattern.
Exercise 1’s goal is to obtain experience with design and implementation of Wrapper Façade classes and to obtain knowledge of the different implementation options. Exercise 2’s goal is to obtain experience with design and implementation of the Reactor pattern and to gain knowledge of the different implementation options. Exercise 3’s goal is to obtain experience with implementation of the Reactor pattern in a real context with a distributed client and server system.
The 1st exercise requires to implement 4 classes that encapsulate some of the WinSock2.h library methods. These classes are: INET_Addr, SOCK_Stream, SOCK_Acceptor and SOCK_Connector. INET_Addr creates an address containing the port, protocol type, etc. SOCK_Stream includes methods that perform I/O operations on a connected socket. SOCK_Acceptor includes the method accept() that accepts a connection on a socket. SOCK_Connector includes the method connect() that connects a client to a specific socket. The four classes are to be used in a server and client application where the acceptor is in the server app and the connector is in the client app.
The 2nd and 3rd exercise requires to create a server and a client application based on the reactor pattern and wrapper facades created in the 1st exercise that would handle 3 types of events.

1.1 Patterns used in the solution

The following patterns are used in the solution:

- Wrapper facade - encapsulates the functions and data provided by existing non-object-oriented APIs within more concise, robust, portable, maintainable, and cohesive object-oriented class interfaces.
- Reactor - allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.
- Singleton - restricts the instantiation of a class to one object; used in the Reactor class to restrict it to a single instance.
- Bridge - decouples an abstraction from its implementation so that the two can vary independently; used in the Reactor class in order to point to the Reactor interface that can have multiple different implementations.
- Hook Method and Template Method patterns - defines the program skeleton of an algorithm in a method, called template method, which defers some steps to subclasses;

used in the single method dispatch interface strategy in order to make it possible to modify the handle_event method in the classes that it would be implemented.

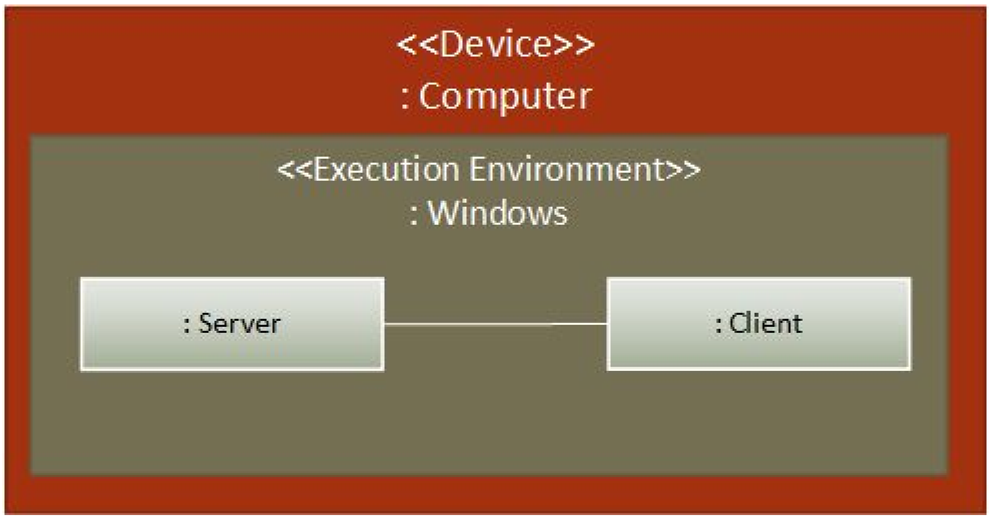
2 Solution

2.1 Discussion of architecture decisions

All the team members develop on Windows and therefore we had to choose the libraries supported by our operating system for both patterns.
In the Wrapper Facade exercise the architectural decisions cannot vary much because the implementation is straightforward and is clearly described in the book and in the exercise requirements.
In the Reactor pattern there were several questions that raised discussions. We had to choose the synchronous event demultiplexer and we chose the select() method as the synchronous event demultiplexer. select() is a common synchronous event demultiplexer function for I/O events supported by many operating systems, including UNIX and Win32 platforms.
Then we had to determine the event handling dispatch interface strategy and we chose to apply the single-method dispatch interface strategy in order to have a single event handling method in the Event_Handler that would handle all type of events and also, this strategy makes it possible to support new types of indication events without changing the class interface.
Another decision was to have just one instance of the reactor in order to centralize event demultiplexing and dispatching in one reactor instance within an application.
We also had to choose between the two general strategies for configuring handles with event handlers which are hard-coded and generic. We didn't hard-code the wrapper facade classes inside a concrete event handler. The wrapper facade classes are instantiated in the applications and then the stream is passed to a concrete event handler as a parameter. So, it is something more generic in a way. This strategy creates more flexible and reusable event handlers.
In designing the solution we also discussed and selected between having three connections between the client and server (one for each event type) or have a solution with one connection for each client (multiplexing the events). We concluded that multiplexing the events and having one connection per client would generate a lot more coding implying parsing and serializing as well as many if statements and switches.

2.2 Deployment diagram

The deployment diagram for these two applicaton is fairly uninteresting as both the application is running on the same computer in the same environment.



2.3 Design and implementation of client

2.3.1 Intro to design of client

The client creates a socket, connects to the server using the server's address and invokes the connect() method. The client in the Wrapper Facade exercise sends a message to the server through the socket stream. The client in the Reactor exercise sends a concrete event through the socket stream.

2.3.2 Class diagrams

2.3.3 Sequence diagrams

The sequence diagram for the client will look similar to the sequence diagram of the server because both applications use the reactor pattern. The only difference is that they may use different Event_Types, but this is also not relevant to the diagram.

2.3.4 Implementation details

The implementation activities start with the demultiplexing/dispatching infrastructure components and then cover the application components. The Event_Handler interface contains a hook method handle_event(Event_Type et) which is a single-method dispatch interface strategy. We use event handler objects as the target dispatching type so that the Reactor pattern implementation dispatches concrete event handler objects.

```
class Event_Handler
	{
	public:
		virtual void handle_event(Event_Type et) = 0;
```

```
            virtual SOCKET get_handle() const = 0;
            virtual ~Event_Handler();
      };
```

The definition of the reactor includes the methods that register or remove event handlers and their associated handles, as well as handling events.We are going to use the singleton pattern in order to have a single instance of the reactor per session. To shield applications from complex and non-portable demultiplexing and dispatching operating system platform mechanisms, the Reactor uses the Bridge pattern. The reactor interface corresponds to the abstraction participant in the Bridge pattern, whereas a platform-specific reactor instance is accessed internally
via a pointer, in accordance with the implementation hierarchy in the Bridge pattern.

```
class Reactor
      {
      public:
            void register_handler(Event_Handler *eh, Event_Type et);
            void remove_handler(Event_Handler *eh, Event_Type et);
            void handle_events(timeval *timeout = 0);
            static Reactor* instance();
            void setReactorInterface(Reactor_Interface* reactorInterface);
      private:
            Reactor_Interface* reactor_interface;
            static Reactor* pointerInstance;
      };
```

The Reactor_Interface is going to be implemented in the Reactor Implementation where we will have to add tuples to a demultiplexing table. When we register a handler we actually add to the table. This table is a manager that contain a set of <handle, event handler, indication event types> tuples. Each handle serves as a 'key' that the reactor implementation uses to associate handles with event handlers in its demultiplexing table. This table also stores the type of indication event, such as WRITE and READ, that each event handler has registered on its handle. In the handle events method we call a function convert_to_fd_sets that converts all the entries in the table into file descriptor sets of event types. These are then selected with the select() method and handled according the concrete event handler and event type.
Then, we have 2 concrete event handlers: alarm, log and patient event handlers. Each of these implement the Event_Handler interface. In the handle event method of a concrete event handler is stated what should be done about what we received on the stream. For example, the handle event method of the AlarmEventHandler outputs a timestamp, the priority text and the alarm text of an Alarm event.

The client for the Wrapper Facade exercise includes INET_Addr, SOCK_Connector and SOCK_Stream. The application begins with initializing the library with WSAStartup. The the client creates an address with INET_Addr and a SOCK_Stream object. These 2 are passed to the SOCK_Connector constructor that would create a socket, call the connect method to connect to that socket and then it sets the connected socket to the stream object.

The client for the Reactor exercise includes Wrapper Facades client libraries, the reactor, reactor implementation and the concrete event handlers. The scenario starts the same as in the client application in the Wrapper Facade exercise and it continues with the instantiation of the reactor, the creation of the write events for a concrete event handler and the handle_events method call.

2.4 Design and implementation of server

2.4.1 Intro to design of server

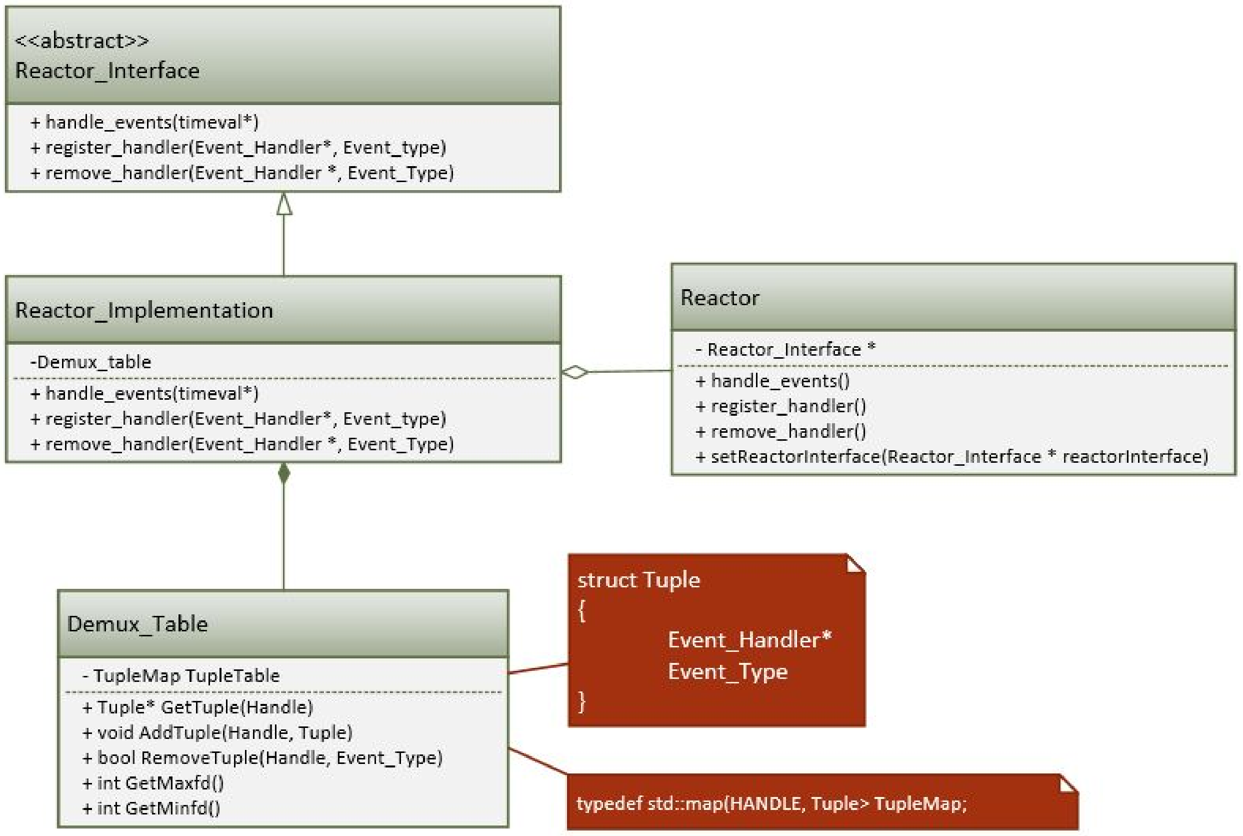
The server application creates a TCP socket with a specific address that listens to connections from the clients. When the client connects to the server, the server uses the stream to receive data from the client. The server application for the Reactor pattern does the same, but, instead of receiving data from the client, it receives event requests and handles them using the reactor instance and its registered concrete event handlers.
The server application shall handle the following three event types:

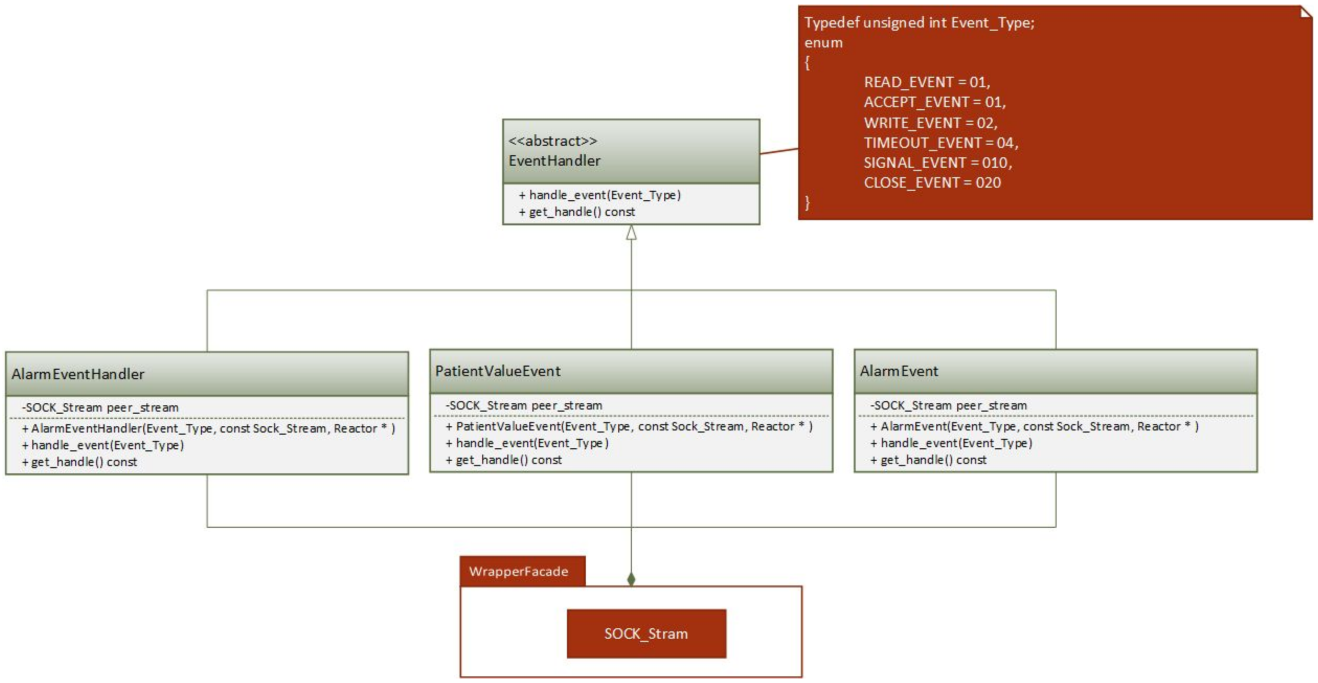
- An alarm event (carrying a priority field and a text string)
- A patient value event (carrying a type field and a value field)
- A log event (carrying a text string)

Each time an alarm event is received the application shall display the alarm info together with a time stamp. This timestamp will be added in the server application at the time the server receives the event.
The handling of the events is minimized so the only thing that happens is the the content in the event is written out to the console.

2.4.2 Class diagrams

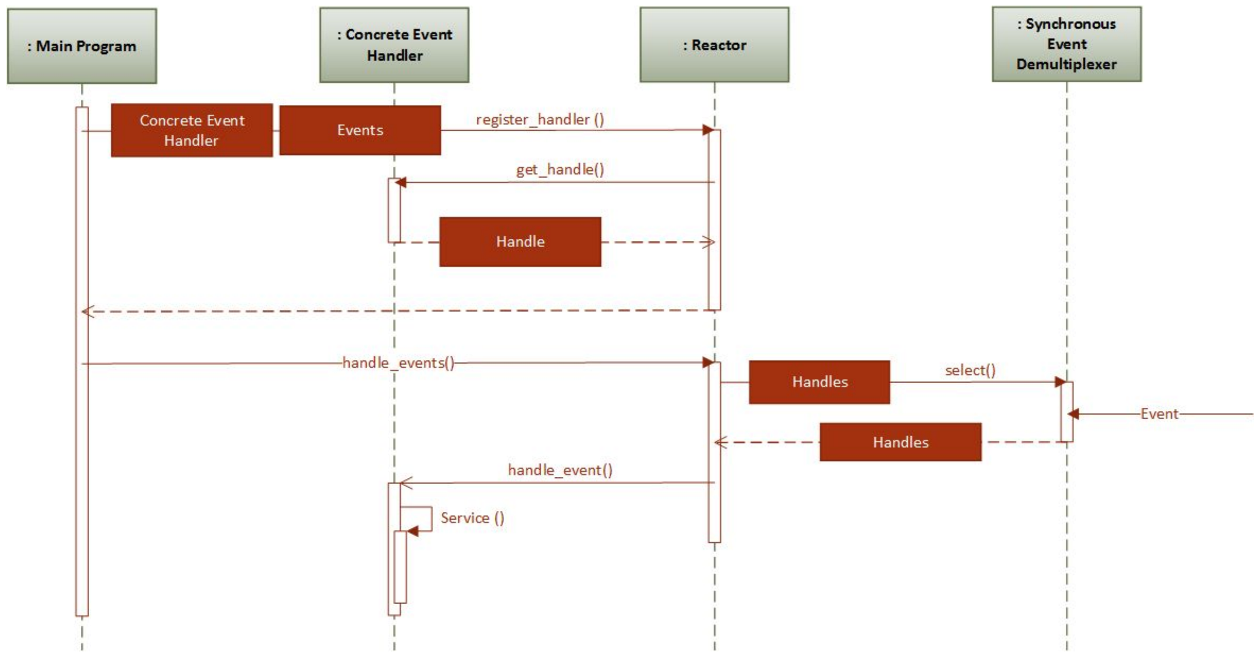
UML class diagram showing some of the classes used in the server application. The bridge pattern can be seen here as the “Reactor” class has a “Reactor_Interface” type pointer to the the “Reactor_Implementation” class. Hereby decoupling the implementation form the abstraction.


<<abstract>>
Reactor_Interface
+ handle_events(timeval*)
+ register_handler(Event_Handler*, Event_type)
+ remove_handler(Event_Handler *, Event_Type)
Reactor_Implementation
-Demux_table
+ handle_events(timeval*)
+ register_handler(Event_Handler*, Event_type)
+ remove_handler(Event_Handler *, Event_Type)
Reactor
- Reactor_Interface *
+ handle_events()
+ register_handler()
+ remove_handler()
+ setReactorInterface(Reactor_Interface * reactorInterface)
Demux_Table
- TupleMap TupleTable
+ Tuple* GetTuple(Handle)
+ void AddTuple(Handle, Tuple)
+ bool RemoveTuple(Handle, Event_Type)
+ int GetMaxfd()
+ int GetMinfd()
struct Tuple
{
Event_Handler*
Event_Type
}
typedef std::map(HANDLE, Tuple> TupleMap;


Typedef unsigned int Event_Type;
enum
{
READ_EVENT = 01,
ACCEPT_EVENT = 01,
WRITE_EVENT = 02,
TIMEOUT_EVENT = 04,
SIGNAL_EVENT = 010,
CLOSE_EVENT = 020
}
<<abstract>>
EventHandler
+ handle_event(Event_Type)
+ get_handle() const
AlarmEventHandler
-SOCK_Stream peer_stream
+ AlarmEventHandler(Event_Type, const Sock_Stream, Reactor *)
+ handle_event(Event_Type)
+ get_handle() const
PatientValueEvent
-SOCK_Stream peer_stream
+ PatientValueEvent(Event_Type, const Sock_Stream, Reactor *)
+ handle_event(Event_Type)
+ get_handle() const
AlarmEvent
-SOCK_Stream peer_stream
+ AlarmEvent(Event_Type, const Sock_Stream, Reactor *)
+ handle_event(Event_Type)
+ get_handle() const
WrapperFacade
SOCK_Stram

2.4.3 Sequence diagrams

In the sequence diagram the main function calls on the server is described.



2.4.4 Implementation details

The implementation details of the Reactor pattern are stated in the 2.3.4 Implementation details. The server for the Wrapper Facade exercise includes INET_Addr, SOCK_Acceptor and SOCK_Stream. The application begins with initializing the library with WSAStartup. Then the server creates an address with INET_Addr. The address is passed to the SOCK_Acceptor constructor that would create a socket, bind the address to the newly created socket and listen for client connections. When a client tries to connect to the server, the server comes out of the listen() method and goes to the accept() method. When the server accepts a connection on a socket, it changes it's active property to True. This socket is then set to the stream so that the server can now receive through the stream.
The server for the Reactor exercise includes Wrapper Facades client libraries, the reactor, reactor implementation and the concrete event handlers. The scenario starts the same as in the server application in the Wrapper Facade exercise and it continues with the instantiation of the reactor, the creation of the read events for a concrete event handler and the handle_events method call. At the creation of the concrete event handler, the stream is passed as well and inside the handle_event method of a concrete event handler the stream is used for reading.

A test stub was developed for testing the server without the client application running. This test stub simply sends events to the server.

3 Discussion of results

The two applications were tested via a local host connection. Here is was possible to input a message via the console on the client, send as an alarm and have the server react to it and write it out in its own console.

4 Conclusion

In conclusion two applications were implemented and tested. We have 3 solutions. The first solution “Ex-1” includes 3 projects: WrapperFacade, ServerApplication and ClientApplication. The second solution actually resolves exercise 2 and 3 but it is named “Ex-2” and it includes 4 projects: WrapperFacade, ReactorPattern, ServerApplication, ClientApp. The 3rd solution is just another version of the 1st exercise, but in C#. A problem can appear when trying to run the apps because of some strange errors from referencing. But the apps run on the developer’s machine. We understood the patterns.

The Wrapper facade pattern is used to encapsulate the complex low level code. There are no obvious downsides to the wrapper facade pattern.

Using the bridge pattern were calling a method in the reactor evokes the same method in the Reactor_Implementation might seem like an unnecessary extra call and if it is done many times it might take up cycle time. However in C++ this does not cause a problem as the function can be inlined thus taking up no extra time [POSA2, s74]. The big advantage of using the bridge pattern is the ability to change the underlying implementation of the reactor. I this exercise this is not done and the bridge pattern is therefore essentially not used. However it is prepared if the implementation should be changed later.

The singleton was implemented but produced a link error. Initially the error could not be solved. As this is the pattern with the smallest impact on the application solving the problem was given a low priority and the reason for the error could not be found.