AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Architecture and Design of Distributed Dependable Systems TI-ARDI

POSA2: *Reactor Architectural Pattern*

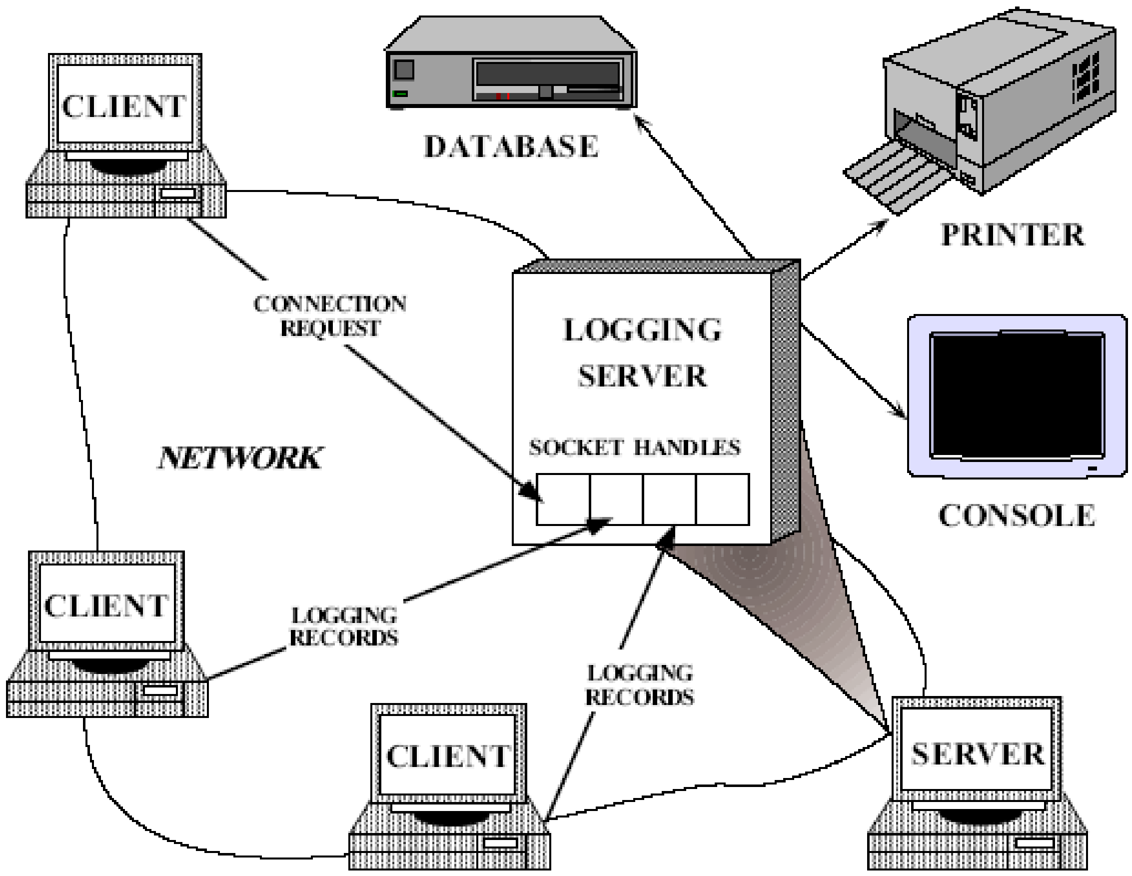
Version: 2.09.2013

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Abstract

The ***Reactor*** architectural pattern allows event-driven applications to demultiplex & dispatch service requests that are delivered to an application from one or more clients

Example – a Distributed Logging Service



TCP communication from clients to a logging server

Context

- An **event-driven application** that receives multiple service requests simultaneously, but processes them **synchronously** and **serially**

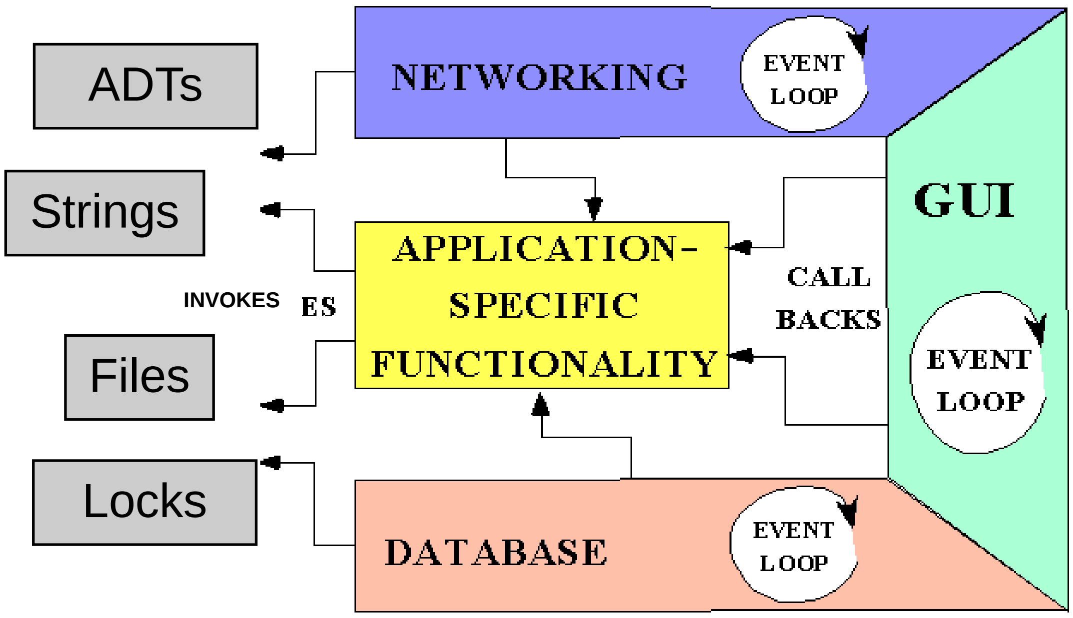
Solution

- **Synchronously** wait for the arrival of indication events on one or more event sources (e.g. connected socket handles)
- Demultiplex and dispatch the events to services that process them
- Perform the application specific functionality in **service handlers**

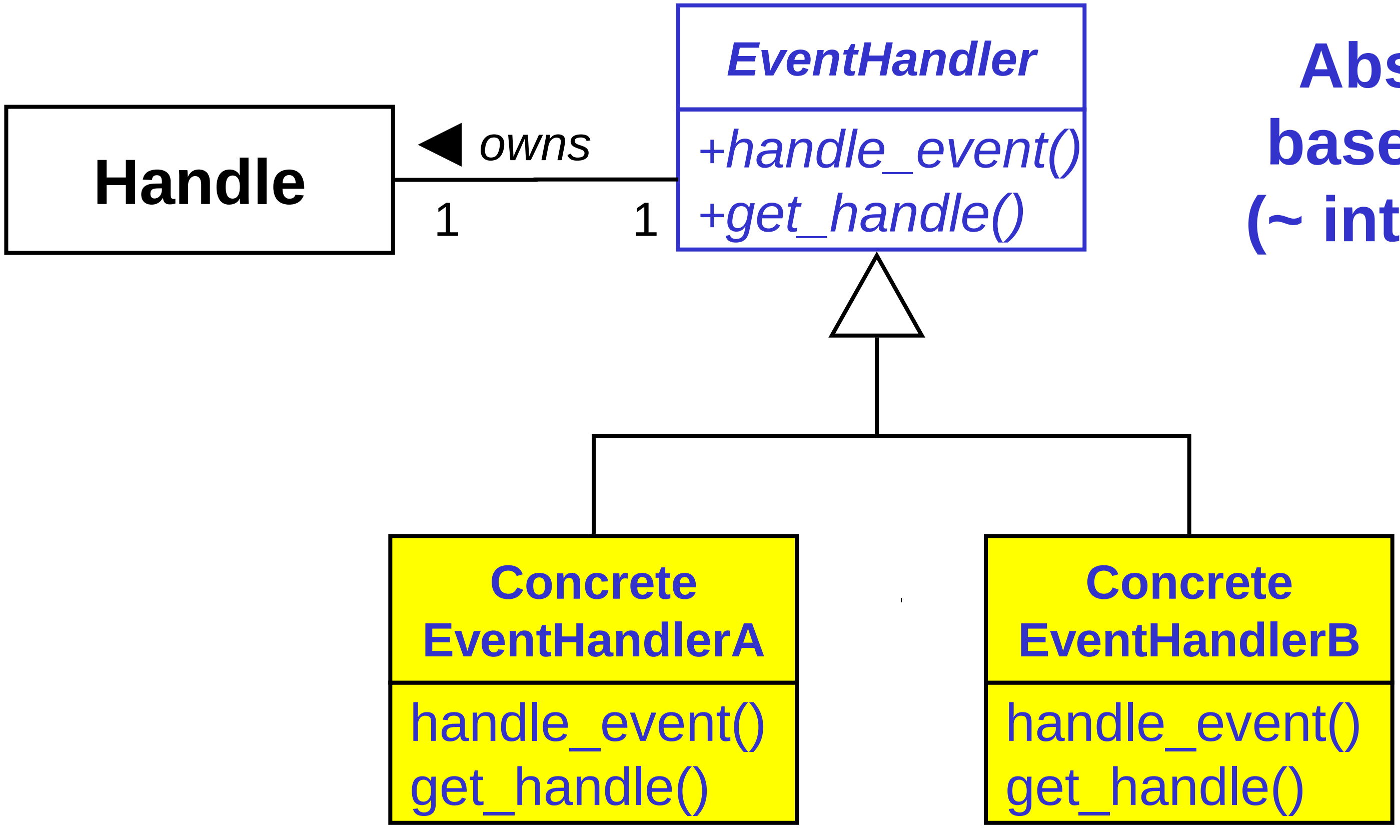
Reactor based Framework

Reactor: Event loop

Framework Architecture

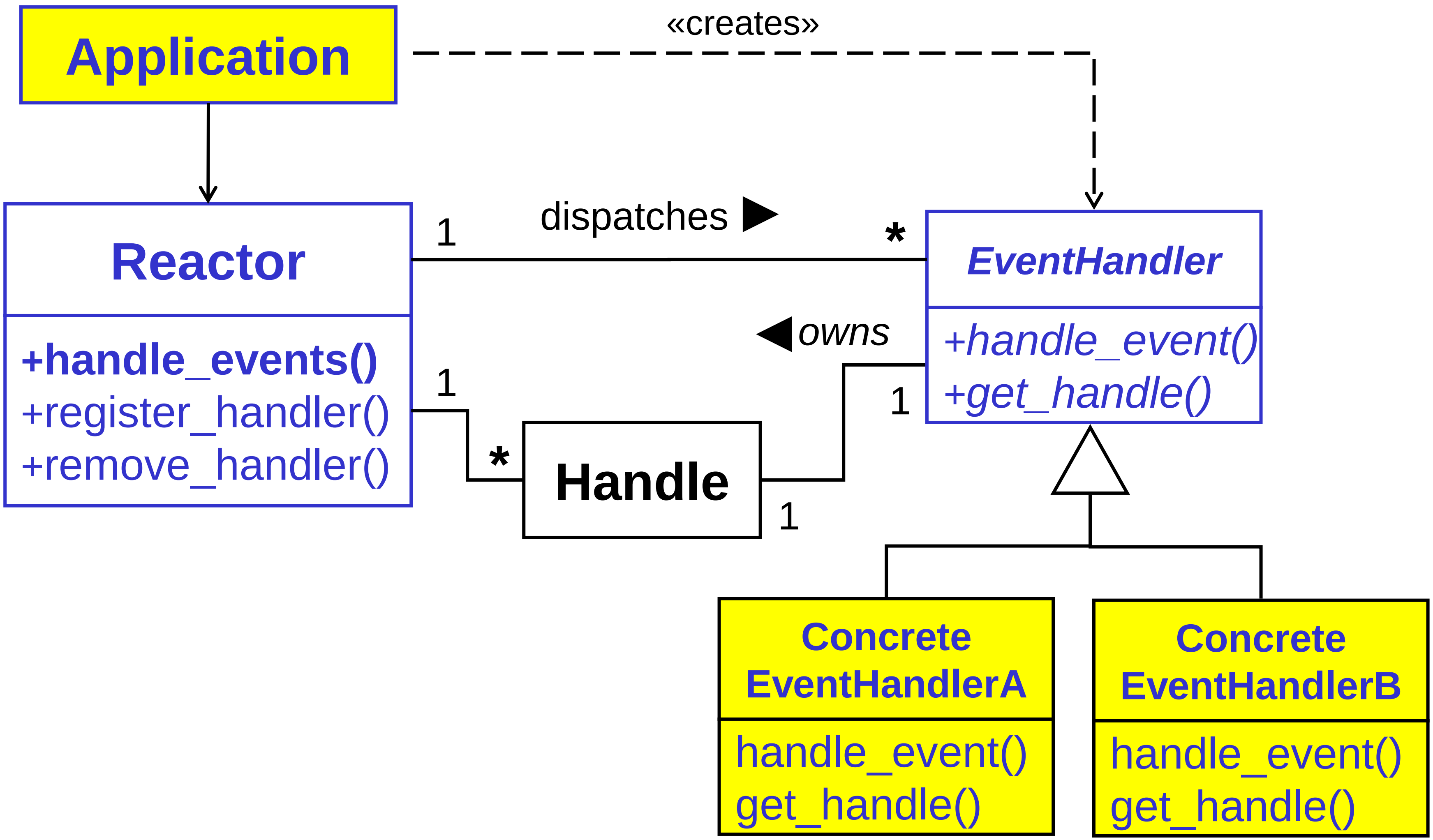


Reactor Pattern – Structure (1)



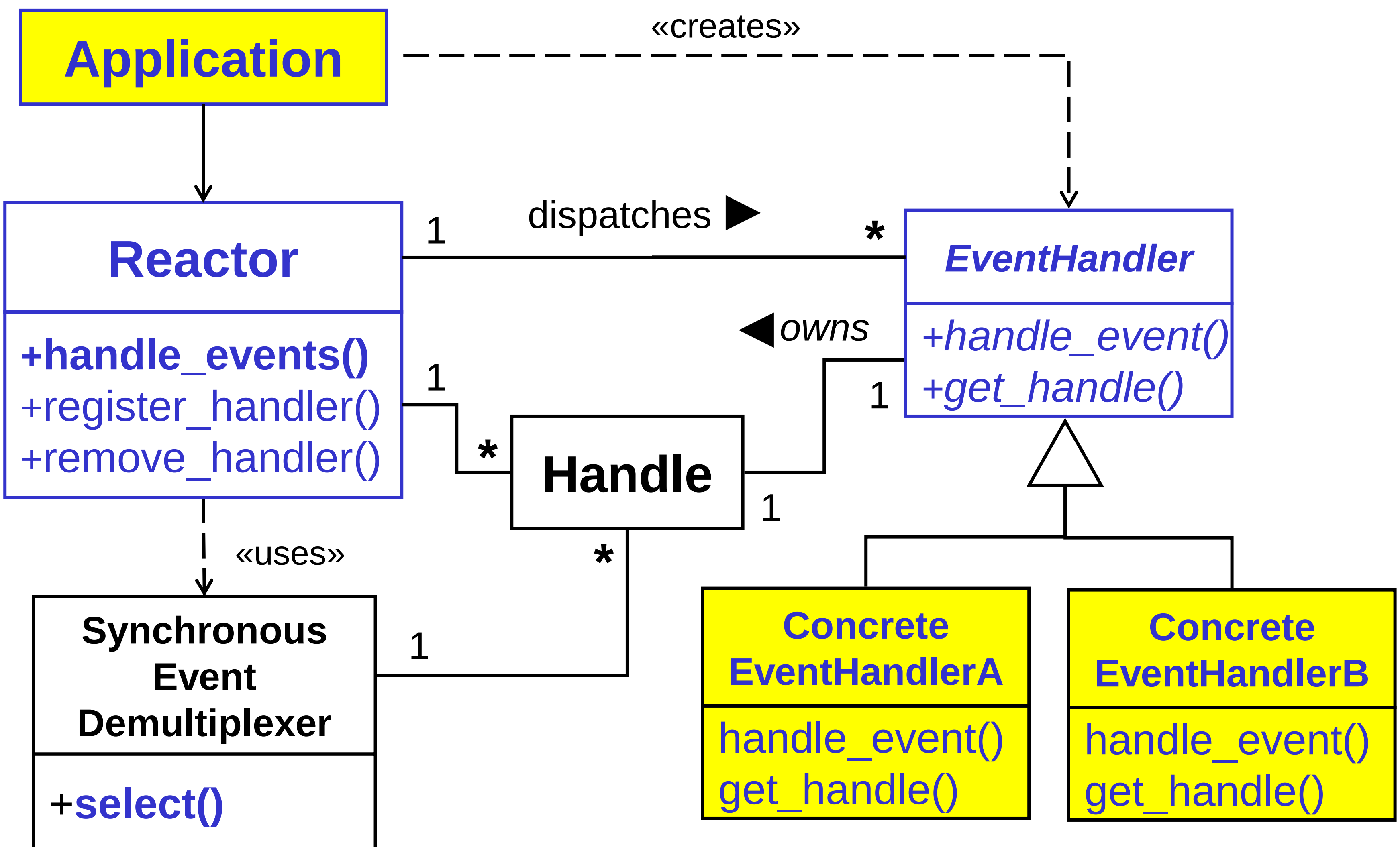
Abstract base class (~ interface)

Reactor Pattern – Structure (2)



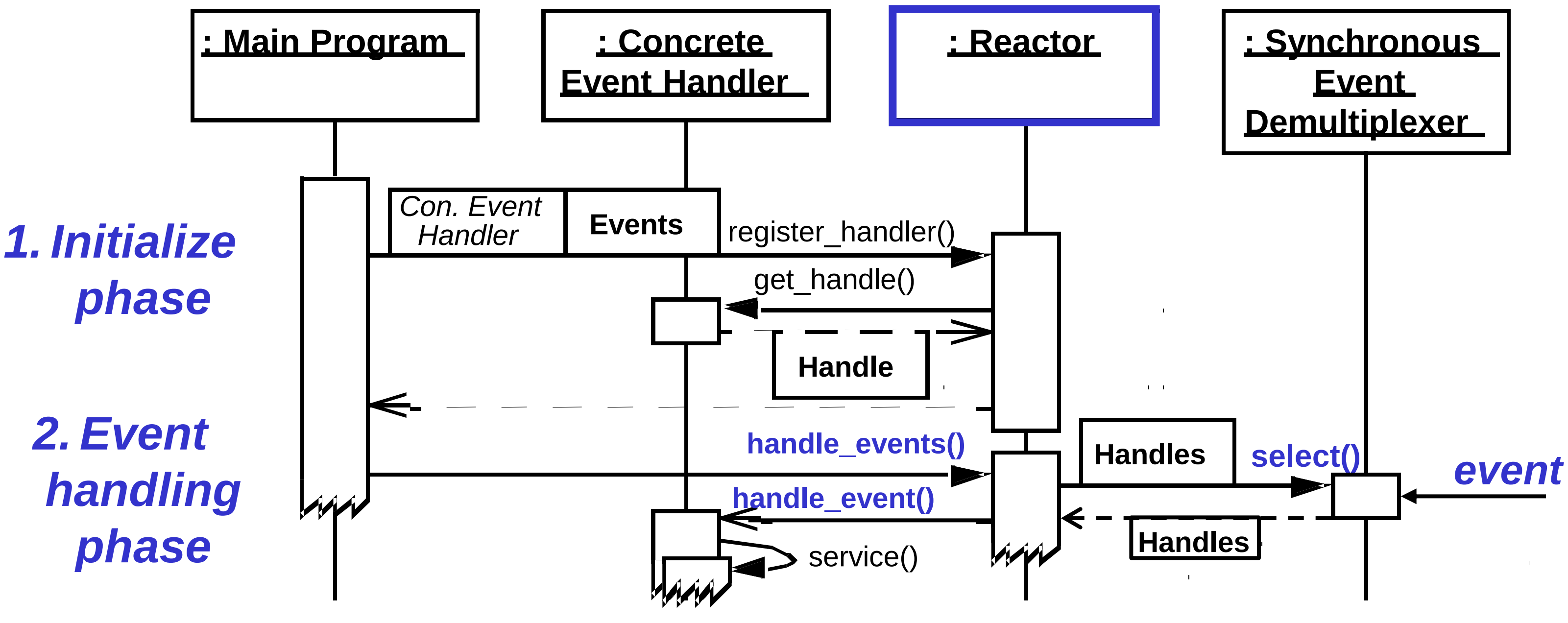
AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Reactor Pattern – Structure (3)



AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Reactor – Sequence Diagram



Observations

- Note inversion of control
- Also note how long-running event handlers can degrade the QoS since callbacks steal the reactor's thread!

Implementation Steps

1. Define the event handler interface
2. Define the reactor interface
3. Implement the reactor interface
4. Determine the number of reactors needed in an application
5. Implement the concrete event handlers

1. Define the Event Handler Interface

Single Method Dispatch (Imp. 1.2)

```
class Event_Handler          // Interface definition in C++
{
public:
    virtual void handle_event(HANDLE handle, Event_Type et) = 0;
    virtual HANDLE get_handle() const = 0;
};


typedef unsigned int Event_Type
enum {
        READ_EVENT = 01,      // ACCEPT_EVENT alias READ_EVENT
        ACCEPT_EVENT= 01,
        WRITE_EVENT= 02,
        TIMEOUT_EVENT= 04
        // etc.
};
```

1. Define the Event Handler Interface

Multi Method Dispatch (Imp. 1.2)

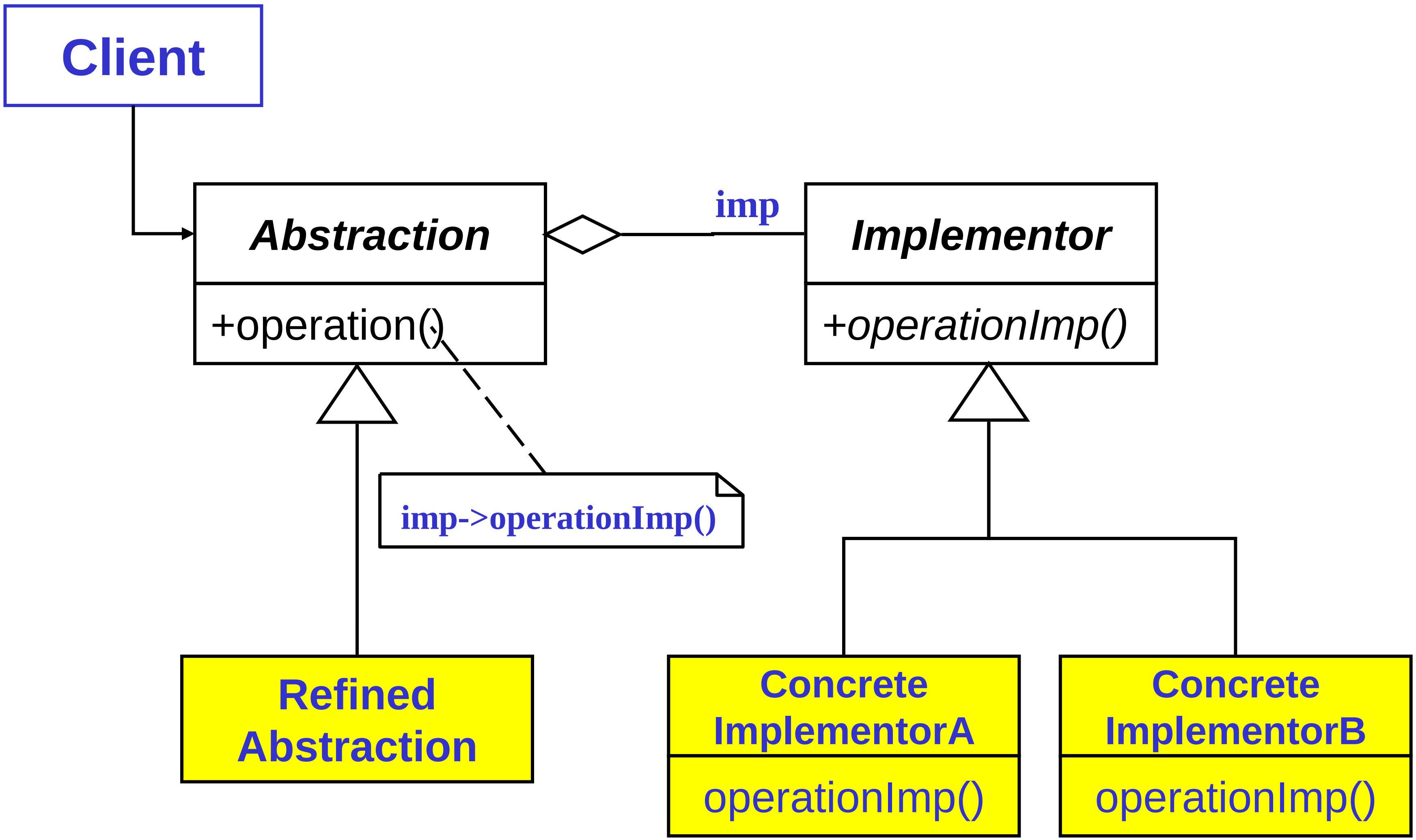
```
class Event_Handler
{
public:
   virtual void handle_input(HANDLE handle) = 0;
   virtual void handle_output(HANDLE handle) = 0;
   virtual void handle_timeout(const Time_Value &) = 0;
   virtual void handle_close(HANDLE handle, Event_Type et) = 0;
   virtual HANDLE get_handle() const = 0;
};
```

2. Define the Reactor Interface

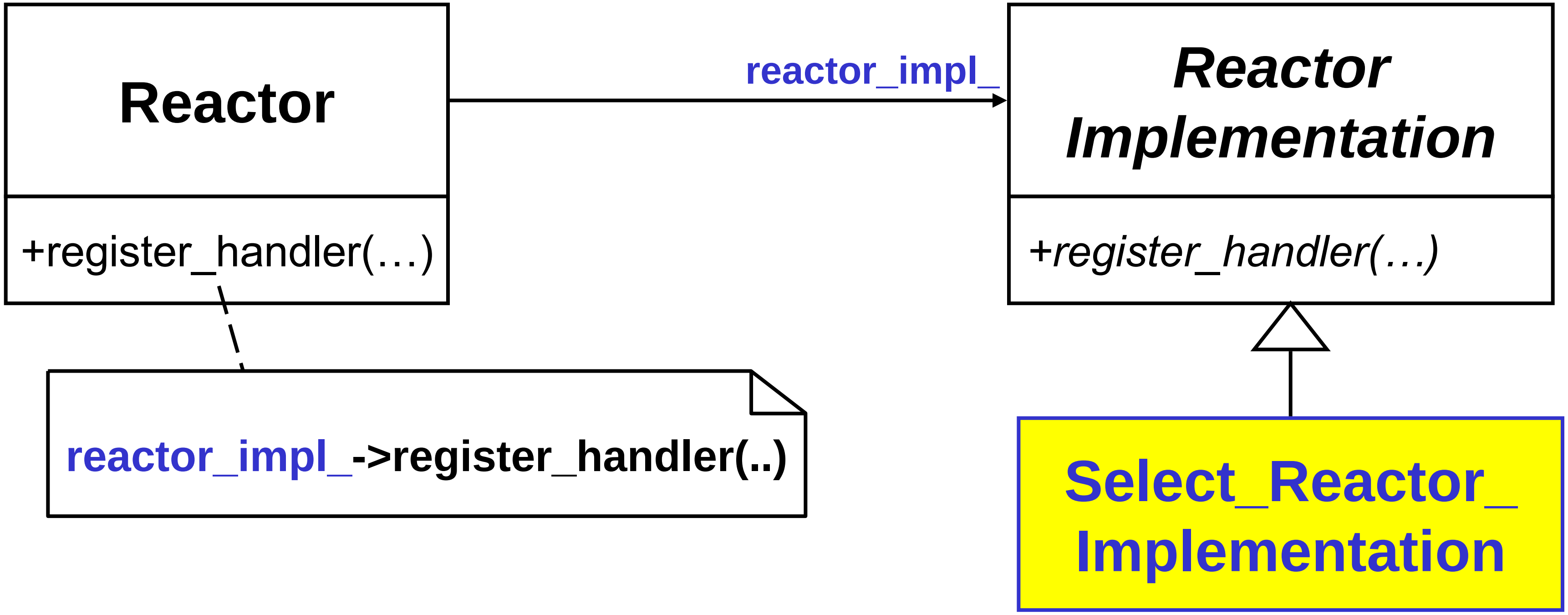
```
class Reactor
{
public:
    virtual void register_handler(Event_Handler *eh, Event_Type et) = 0;
    virtual void register_handler(HANDLE h,
                                  Event_Handler *eh, Event_Type et) = 0;

    virtual void remove_handler(Event_Handler *eh, Event_Type et) = 0;
    virtual void remove_handler(HANDLE h, Event_Type et) const = 0;

    // Entry point into the reactive event loop
    void handle_events(Time_Value *timeout =0);

    // Define a singleton access point (GoF pattern)
    static Reactor *instance();
private:
    Reactor_Implementation *reactor_impl_;     // uses the GoF Bridge pattern
};
```

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Bridge Design Pattern (GoF)



Reactor Implementation (Bridge)




```
void Select_Reactor_Implementation::register_handler(Event_handler *eh,
                                                      Event_type et)
{
   HANDLE handle= eh->get_handle();
//
}
```

3.2 Choose a Synchronous Demultiplexer Mechanism

The synchronous event demultiplexer, as well as the handles and handle sets, are often existing operating system mechanisms (e.g. **select()**)

Operating System Demux Mechanisms:

select():	**Unix, Win32, Linux,VxWork**
poll():	**Unix, System V, release 4.**
WaitForMultipleObjects():	**Win32**

“select” as Demultiplexer Mechanism

```
int select(u_int  max_handle_plus_1,
           fd_set *read_fds,
           fd_set *write_fds,
           fd_set *except_fds,
           timeval *timeout);
```

The select() function examines the three “file descriptor set” (fd_set) to see if any of their handles are ready for reading, writing or have an exceptional condition or check for a timeout

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Example of a ”fd_set” data structure

```
#define FD_SETSIZE      64

typedef struct fd_set
{
      u_int   fd_count;                          /* how many are SET? */
      SOCKET  fd_array[FD_SETSIZE]; /* an array of SOCKETs */
} fd_set;
```

3.3 Implement a Demultiplexing Table

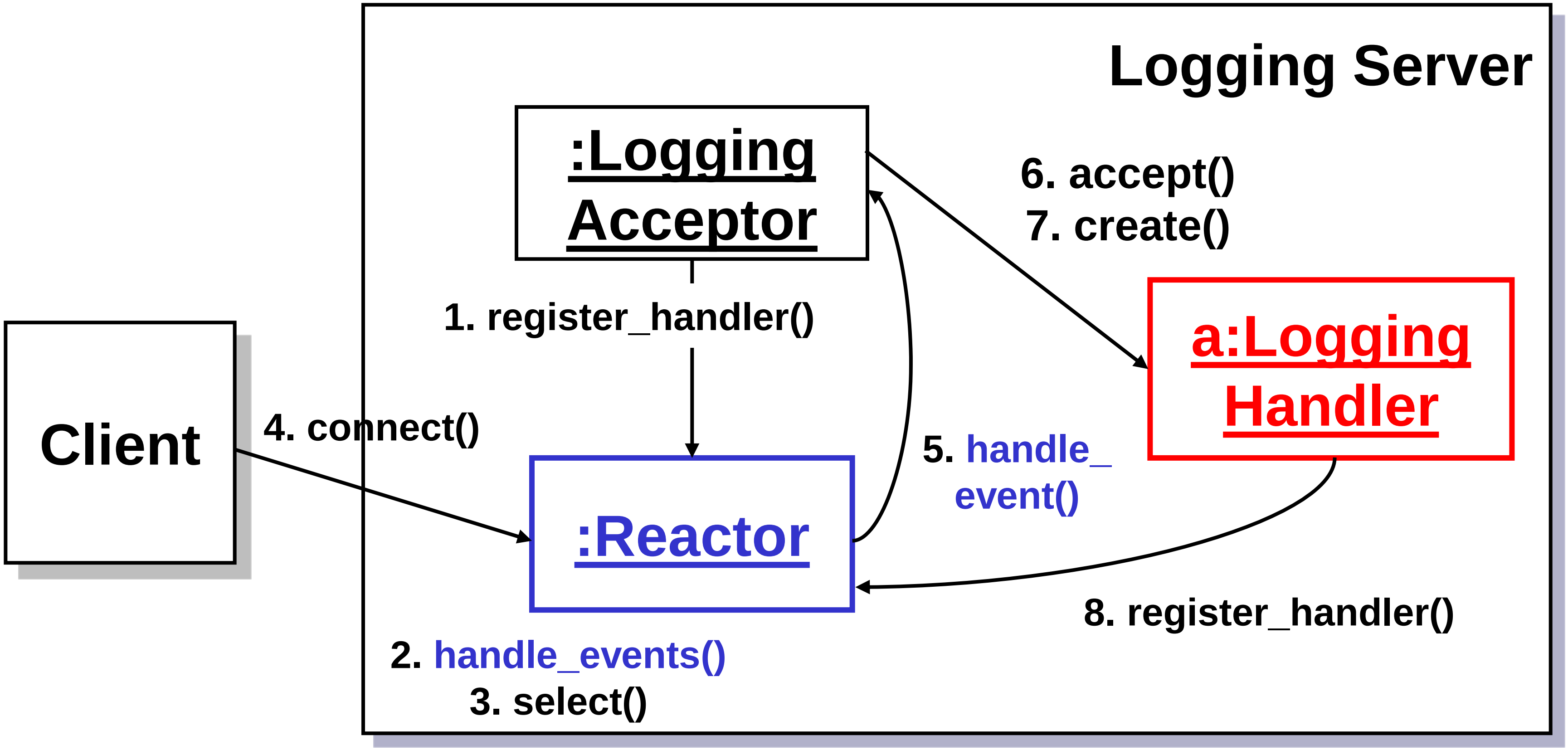
Unix implementation example, where handle values are contiguous ints

```
class Demux_Table
{
public:
    // Convert <Tuple> array to <fd_set>s
    void convert_to_fd_sets(fd_set *read_fds, fd_set *write_fds,
                            fd_set *except_fds);
    struct Tuple
    {
        // Pointer to <Event_Handler> that process
        // the indication event arriving on the handle
        Event_Handler *event_handler_;
        // Bit-mask that tracks which types of indication events
        // <Event_Handler> is registered for
        Event_Type event_type_;
    }
    Tuple table_[FD_SETSIZE];       // FD_SETSIZE macro defined
};                                  // in <sys/socket.h>
```

3.4 Define the Concrete Reactor Implementation (Unix example)

```
class Select_Reactor_Implementation : public Reactor_Implementation {
public:
    void handle_events(Time_Value *timeout = 0) {
            fd_set read_fds, write_fds, except_fds;
            demux_table_.convert_to_fd_sets(&read_fds,&write_fds,&except_fds);
            HANDLE max_handle = MAX_NO_OF HANDLES;
            int result = select( max_handle+1, &read_fds, &write_fds, &except_fds,
                                 timeout);
            if (result <=0) throw;   // handle error or timeout cases here
            for (HANDLE h=0; h <=max_handle; h++) {
                if ( FD_ISSET(&read_fds, h) )                    // std macro
                        demux_table_.table_[h].event_handler_->
                                 handle_event(h, READ_EVENT);
                // perform the same for WRITE_EVENTs and EXCEPT_EVENTs
            }
    };
private:
            Demux_table demux_table_;
}
```

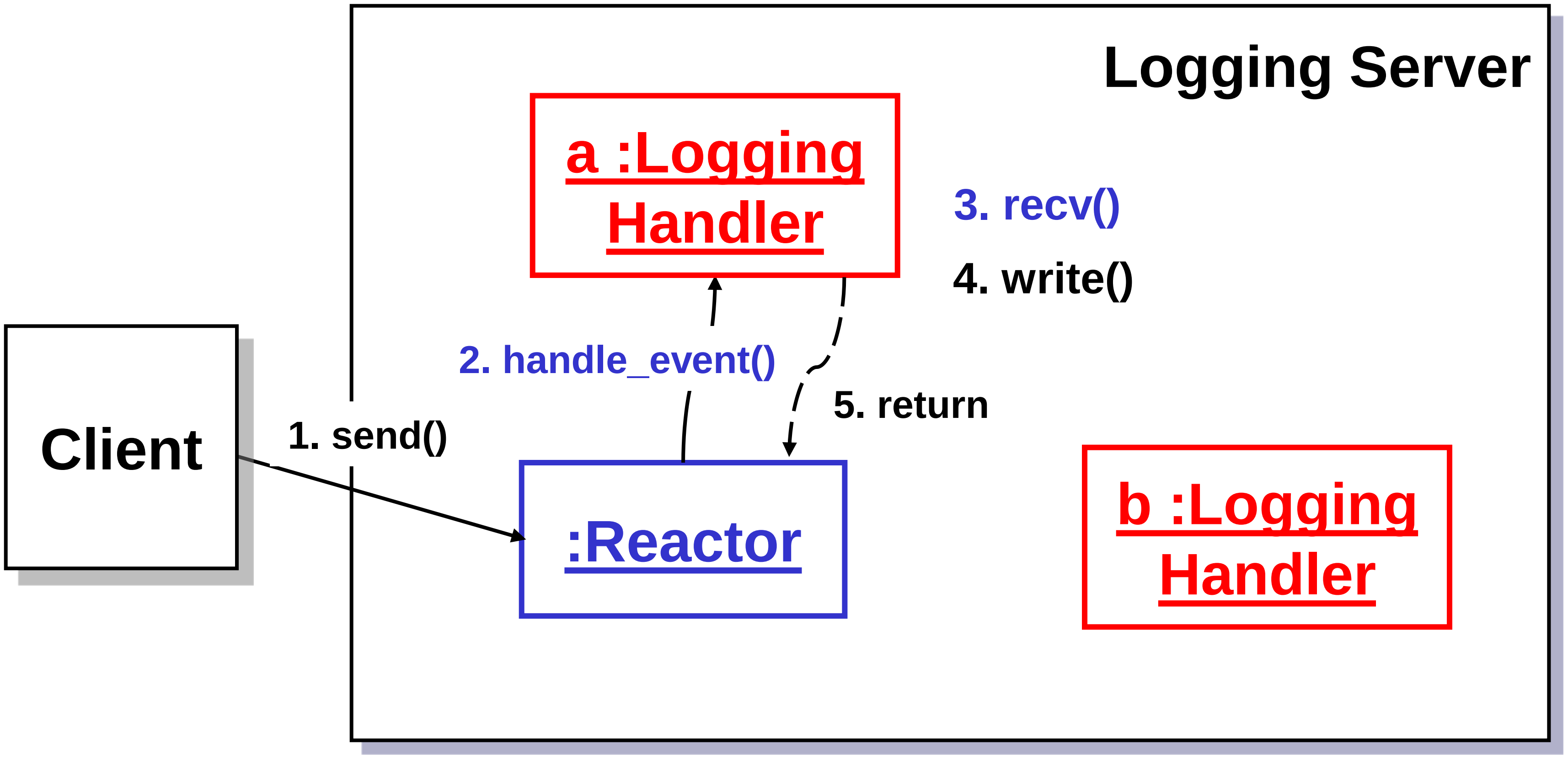
Logging Server Example (1)



Scenario: Client connects to the logging server

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

Logging Server Example (2)



Scenario: client sends a logging record

Logging Server main program

```
const u_short PORT = 10000;   // logging server port number

int main()
{
    INET_Addr addr(PORT);     // Logging server address

    // Initialize logging server endpoint and register
    // with reactor singleton
    Logging_Acceptor la(addr, Reactor::instance() );

   // Event loop that processes client connection requests
   // and log records reactively
   while ( 1 )
         Reactor::instance()->handle_events();
}
```

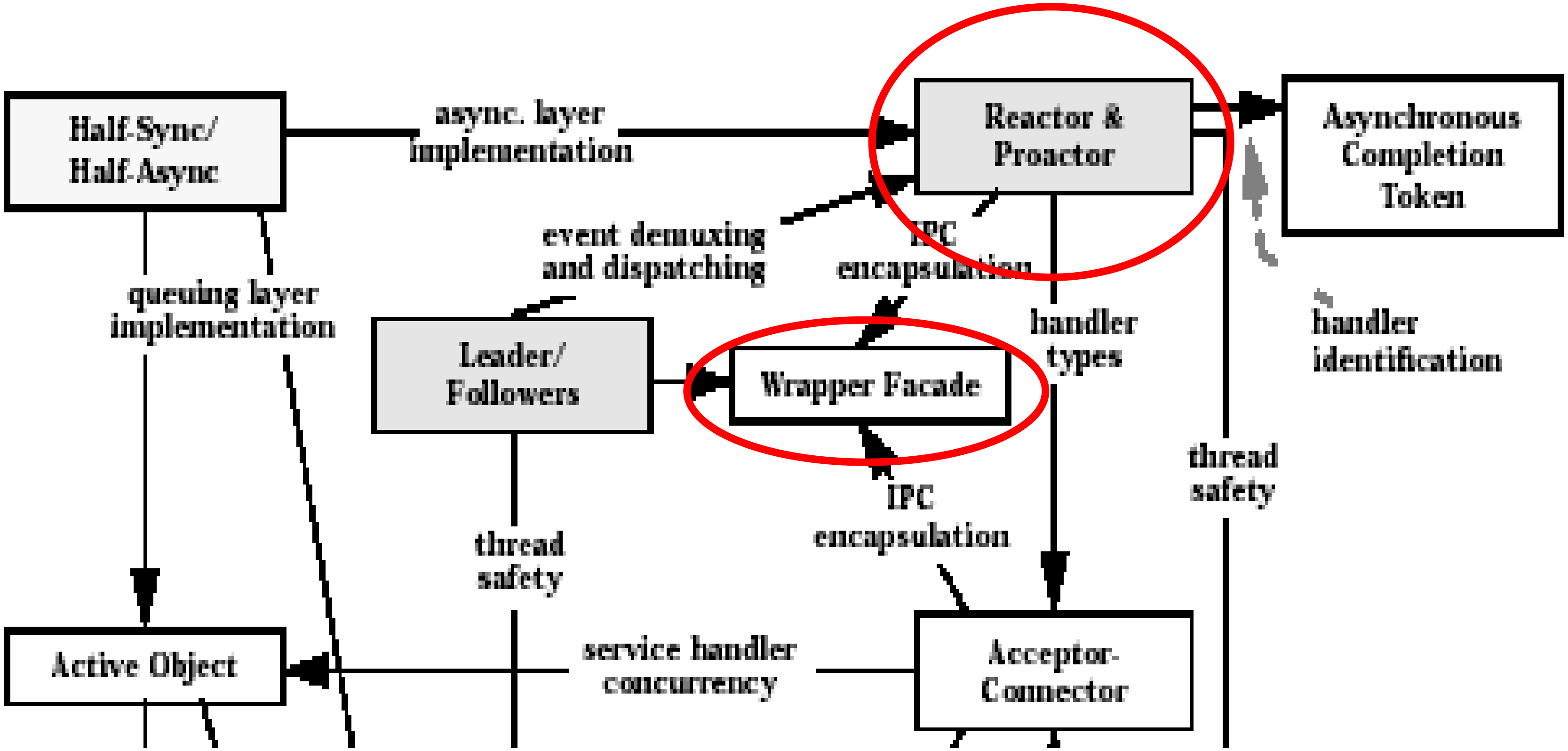
Reactor Pattern - Benefits

- Separation of concerns
  - This pattern decouples application-independent demuxing & dispatching mechanisms from application-specific hook method functionality
- Modularity, reusability, & configurability
  - This pattern separates event-driven application functionality into several components, which enables the configuration of event handler components that are loosely integrated via a reactor
- Portability
  - By decoupling the reactor’s interface from the lower-level OS synchronous event demuxing functions used in its implementation, the Reactor pattern improves portability
- Coarse-grained concurrency control
  - This pattern serializes the invocation of event handlers at the level of event demuxing & dispatching within an application process or thread

Reactor Pattern - Liabilities

- Restricted applicability
  - This pattern can be applied efficiently only if the OS supports synchronous event demuxing on handle sets
- Non-preemptive
  - In a single-threaded application, concrete event handlers that borrow the thread of their reactor can run to completion & prevent the reactor from dispatching other event handlers
- Complexity of debugging & testing
  - It is hard to debug applications structured using this pattern due to its inverted flow of control, which oscillates between the framework infrastructure & the method call-backs on application-specific event handlers

Relation to other POSA2 Patterns



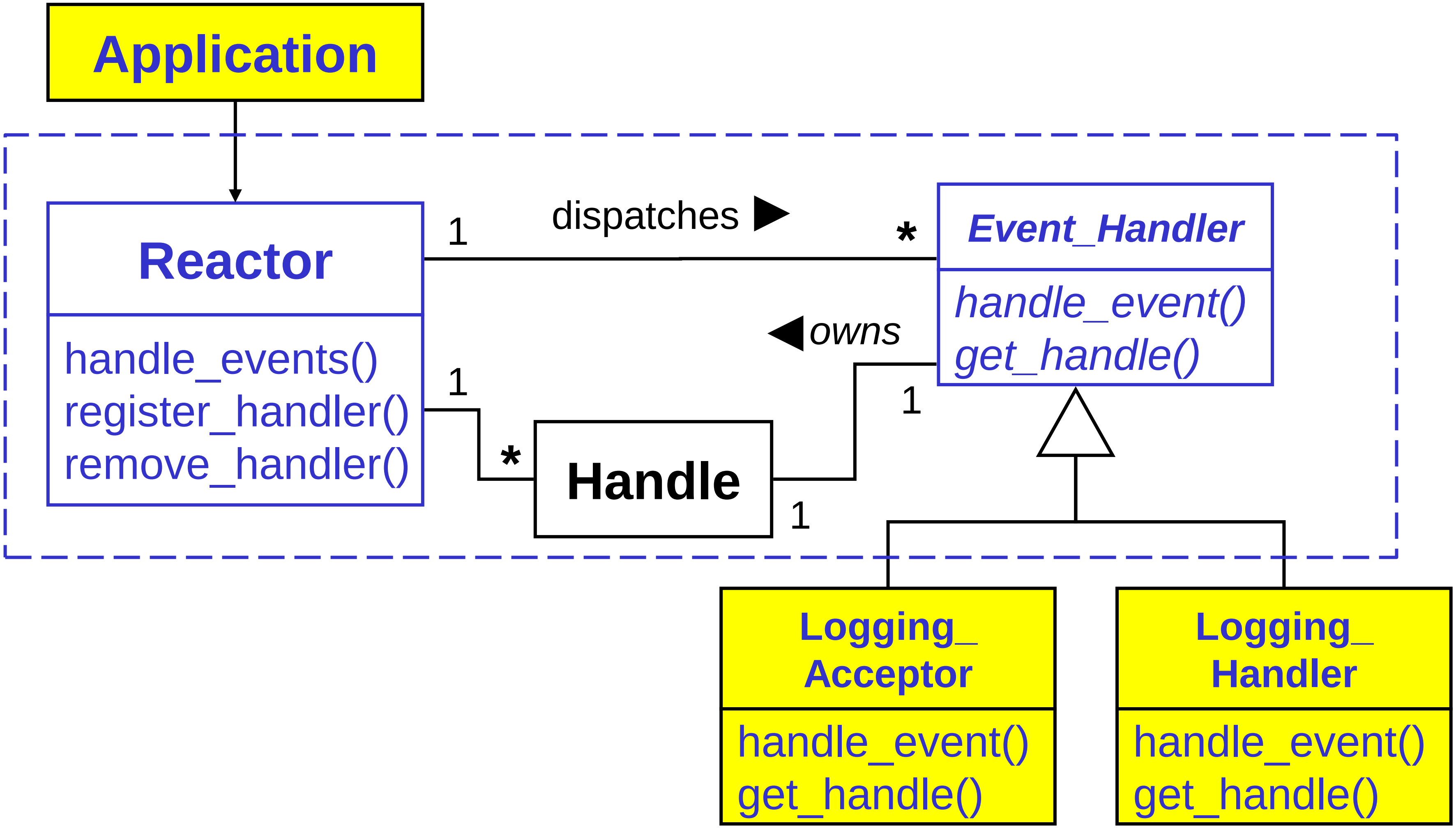
See Also

- Reactor is related to:
  - Observer (GoF), Publisher-Subscriber (POSA1)
    - where all dependent are informed
    - In the Reactor **a single handler is informed**
  - Chain of Responsibility (GoF)
    - Searches the chain to locate the first matching handler
    - The Reactor associates a specific event handler with a particular source of events
  - Proactor
    - The Reactor is a synchronous variant of the asynchronous Proactor pattern

Logging Server - Example

- Repetition
  - see slides 22-24, where two scenarios and the main server program are shown
- The following slides will show the two concrete Event_Handler subclasses:
  - Logging_Acceptor
    - accepts a new client connection
  - Logging_Handler
    - receives client log records

Logging Server Example



Logging_Acceptor class (1)

```
class Logging_Acceptor : public Event_Handler {
public:
   Logging_Acceptor(const INET_Addr &addr, Reactor *reactor) :
                                   acceptor_(addr), reactor_(reactor)
   {
         reactor_->register_handler(this, ACCEPT_EVENT);
   }
   virtual void handle_event(HANDLE, Event_Type event_type)
   {
         if (event_type == ACCEPT_EVENT) {
            SOCK_Stream client_connection;          // NB! a WrapperFacade
            acceptor_.accept(client_connection);    // NB! a WrapperFacade

            // Create a new <Logging_Handler> (NB! copies client_connection)
            Logging_Handler *handler = new Logging_Handler(
                    client_connection, reactor_);
         }
   }
```

Logging_Acceptor class (2)

```
class Logging_Acceptor : public Event_Handler {
public:
        // continued from previous slide
   virtual HANDLE get_handle() const
   {
        return acceptor_.get_handle();
   }
private:
   // Socket factory that accepts client connections
   SOCK_Acceptor acceptor_;        // NB! a WrapperFacade

   // Cached <Reactor>
   Reactor *reactor_;
};
```

Logging_Handler class (1)

```
class Logging_Handler : public Event_Handler {
public:
    Logging_Handler(const SOCK_Stream &stream, Reactor *reactor) :
                    peer_stream_(stream), reactor_(reactor) {
            reactor_->register_handler(this, READ_EVENT);
    }
    virtual void handle_event(HANDLE, Event_Type event_type) {
            if (event_type == READ_EVENT) {
                Log_Record log_record;
                int result= peer_stream_.recv(&log_record, sizeof log_record);
                if (result != STREAM_ERROR)
                        log_record.write(STDOUT);
                else
                {
                        reactor_->remove_handler(this,READ_EVENT);
                        delete this;           // Deallocate ourselves !
                }
    }
```

Logging_Handler class (2)

```
class Logging_Handler : public Event_Handler {
public:
        // continued from previous slide

   virtual HANDLE get_handle() const
   {
        return peer_stream_.get_handle();
   }
private:
   // Receives logging records from a connected client
   SOCK_Stream peer_stream_;     // NB! a WrapperFacade
   Reactor *reactor_;
}
```

Summary

- The reactor pattern is very useful for designing of event-based frameworks in general
- In this context it takes care of handling and dispatching of network events