/* MauroZorzin 866001 Reactor & Proactor */

Reactor & Proactor

concurrency architecture patterns

Università Degli Studi Milano Bicocca

```
/* MauroZorzin 866001                                    Reactor & Proactor */
```

Reactor

/* MauroZorzin 866001 Reactor & Proactor */

Reactor (Dispatcher, Notifier)

Il modello Proactor consente alle applicazioni event-driven di eseguire il demultiplex e di gestire e distribuire i servizi ai richiedenti

In particolare il pattern ci permette di gestire richieste provenienti da diversi client coordinando l’accesso alle risorse.

Spesso usato in applicazioni web event-driven o in Non-Blocking IO

```
/* MauroZorzin 866001                                    Reactor & Proactor */
```

La struttura introdotta dal pattern Reactor 'inverte' il flusso di controllo all'interno di un'applicazione:

Hollywood Principle – Don’t Call Us, We’ll Call You!

Anche il pattern Observer segue lo stesso principio

Componenti dell'architettura ad eventi

- Event sources: I padri degli eventi

- Demultiplexer: Attende che gli eventi si verifichino sul set di Event sources e li invia ai relativi callback degli Event handlers

- Event handlers: Esegue operazioni specifiche dell'applicazione in risposta alle callback

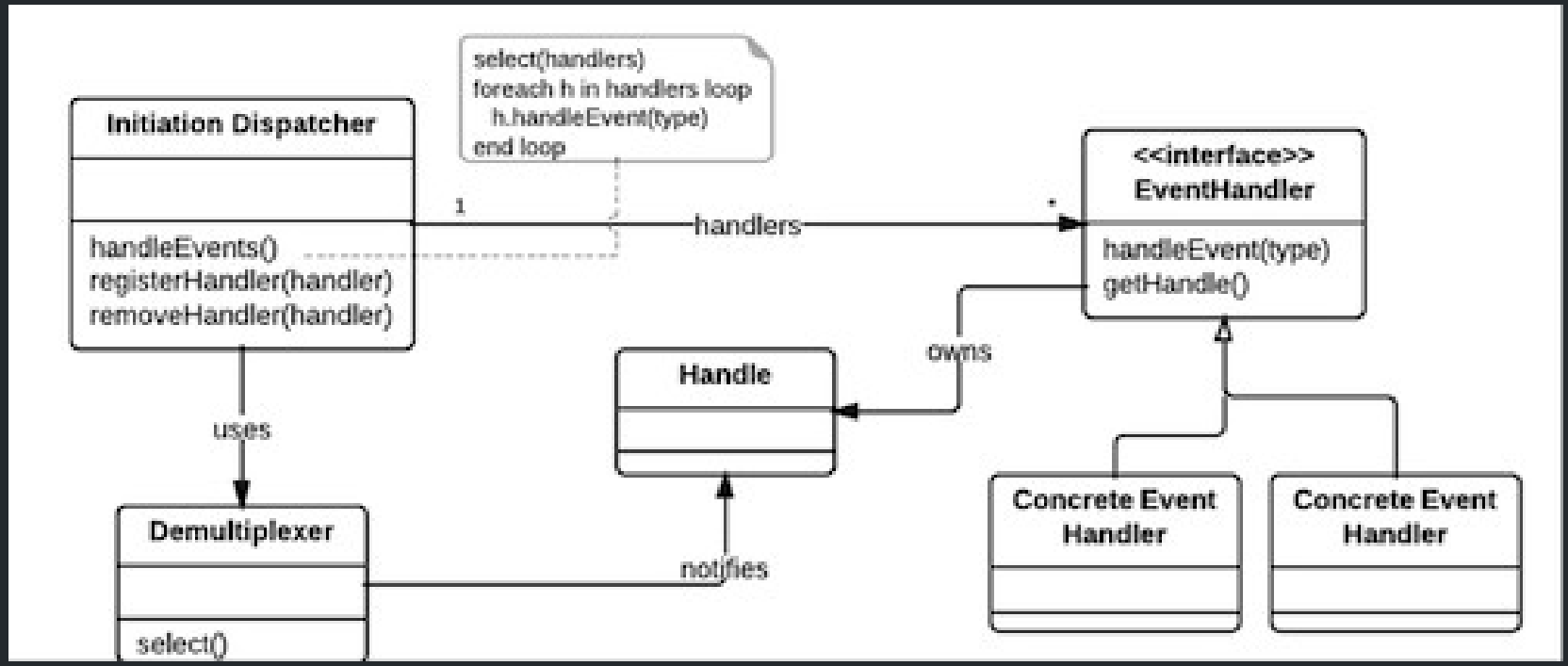
/* MauroZorzin 866001 Reactor & Proactor */

Principali differenze rispetto al tradizionale flusso di controllo:

- Il comportamento è causato da eventi asincroni
- La maggior parte degli eventi deve essere gestita tempestivamente
- Macchine a stati finiti per il controllo dell'elaborazione degli eventi
- Nessun controllo sull'ordine di arrivo degli eventi

/* MauroZorzin 866001 Reactor & Proactor */

Cominciamo dalla struttura:



```
/* MauroZorzin 866001                                    Reactor & Proactor */

Esempio del telefono


- Rete di telecomunicazioni -> Reactor

- Numero di telefono -> Handler

- Qualcuno chiama il numero -> evento in arrivo

- La rete notifica al client che un evento di richiesta è in sospeso,
far squillare il telefono -> demultiplex e dispatch

- Il cliente reagisce alzando il telefono e "elabora" la richiesta
rispondendo alla parte connessa -> specifico handle_event()
```

/* MauroZorzin 866001 Reactor & Proactor */

- Handle: Identifica le fonti di eventi che possono essere prodotte o accodate da richieste esterne o interne

- Event Handler: Definisce un'interfaccia con un insieme di metodi hook

- Concrete Event Handler: Specializza il gestore di eventi per un particolare service e implementa i metodi hook

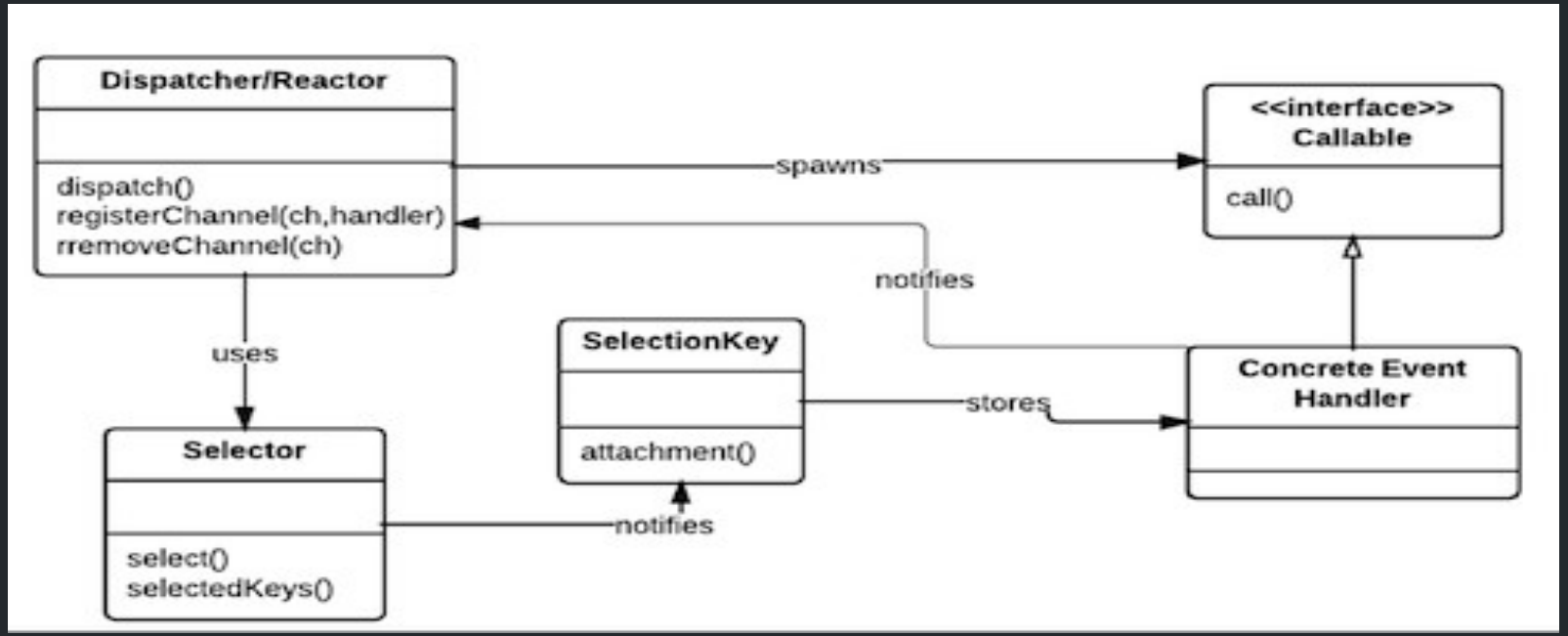
- Reactor: Specifica un'interfaccia per registrare e rimuovere Event Handler e Handlers, esegue l’event loop per reagire a ogni evento demultiplandolo l'handler al gestore di eventi e lanciando l’appropriato hook method

- Synchronous Event Demultiplexer: Attende il verificarsi di eventi, funge da demultiplexer, le sue funzionalità dipendono dal sistema operativo

/* MauroZorzin 866001 Reactor & Proactor */

Per implementare un Reactor in Java necessitiamo di un freamwork

Java NIO framework



```java
/* MauroZorzin 866001                                                    Reactor & Proactor */

  public static void main(String[] args) throws Exception {

          Socket clientSocket = new Socket("localhost", 7070);
          PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
          BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream())
          BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

          String str = br.readLine();

          out.println(str);
          String str2 = in.readLine();
          System.out.println("Server says: " + str2);

          in.close();
          out.close();
          clientSocket.close();
      }


```

```java
/* MauroZorzin 866001                                                    Reactor & Proactor */
public class ReactorManager {
    private static final int SERVER_PORT = 7070;

    public void startReactor(int port) throws Exception {

        ServerSocketChannel server = ServerSocketChannel.open();

        server.socket().bind(new InetSocketAddress(port));

        server.configureBlocking(false);

        Reactor reactor = new Reactor();

    public static void main(String[] args) {
        log.info("Server Started at port : " + SERVER_PORT);
        try {
            new ReactorManager().startReactor(SERVER_PORT);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

```java
// * Associamo il canale con il selettore tramite la register
reactor.registerChannel(SelectionKey.OP_ACCEPT, server);

// * Associamo ad ogni evento il suo handler
reactor.registerEventHandler(
        SelectionKey.OP_ACCEPT, new AcceptEventHandler(
                reactor.getDemultiplexer()));

reactor.registerEventHandler(
        SelectionKey.OP_READ, new ReadEventHandler(
                reactor.getDemultiplexer()));

reactor.registerEventHandler(
        SelectionKey.OP_WRITE, new WriteEventHandler());

reactor.run(); // * Run the dispatcher loop
```

}

```java
/* MauroZorzin 866001                                                    Reactor & Proactor */
public class Reactor {
    private Map<Integer, EventHandler> registeredHandlers = new ConcurrentHashMap<Integer, EventHandler>();
    private Selector demultiplexer;

    public Reactor() throws Exception {
        demultiplexer = Selector.open();
    }

    public Selector getDemultiplexer() {
        return demultiplexer;
    }

    // * Rimuove l'evento dal selettore
    public void removeEventHandler(SelectionKey handle) {
        registeredHandlers.remove(handle.interestOps());
    }

    // * Associa l'evento con il suo handler
    public void registerEventHandler(
            int eventType, EventHandler eventHandler) {
        registeredHandlers.put(eventType, eventHandler);
    }

    // * Associa il canale con il selettore tramite la register
    public void registerChannel(
            int eventType, SelectableChannel channel) throws Exception {
        channel.register(demultiplexer, eventType);
    }
```

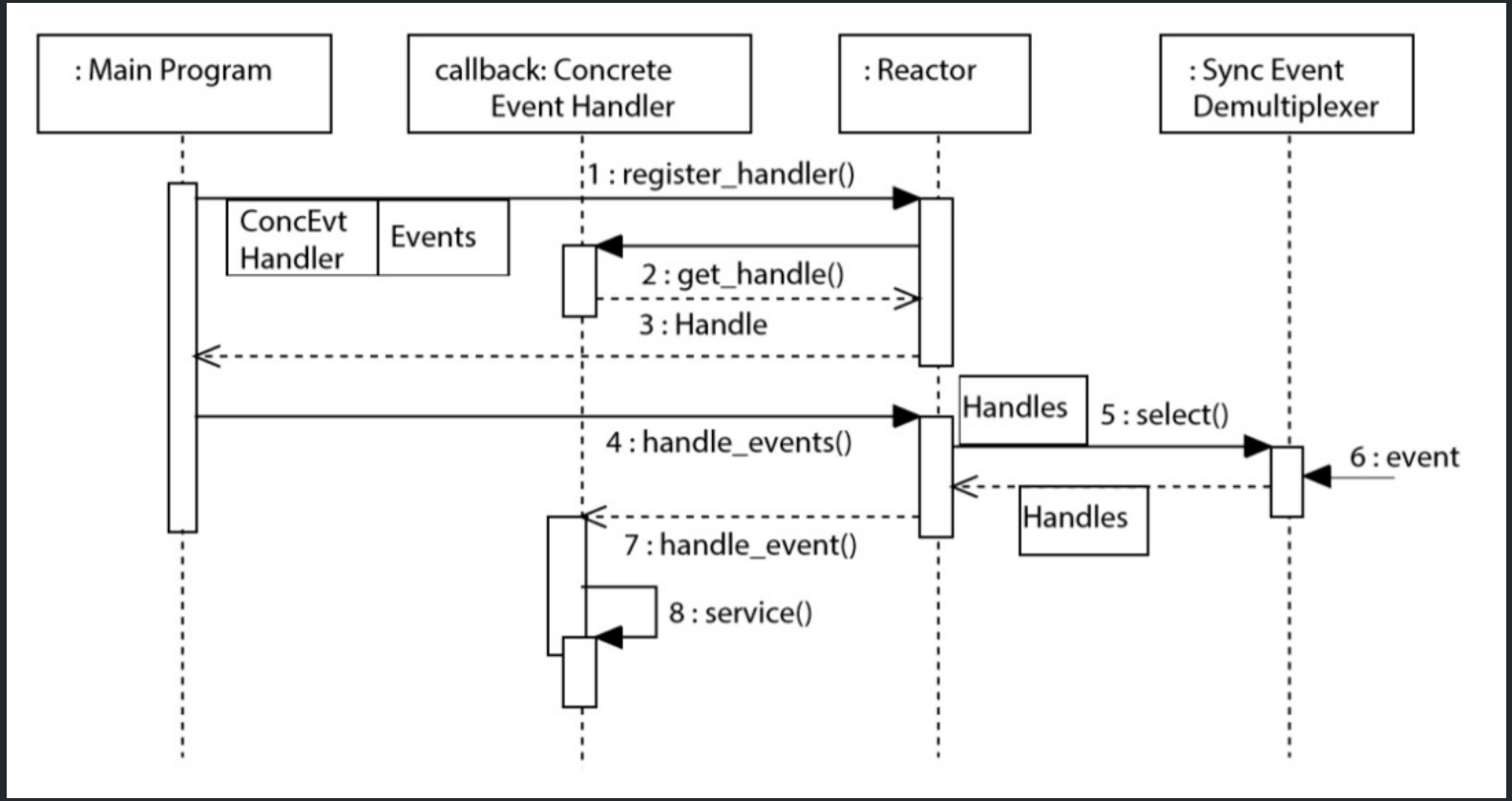
```java
/* MauroZorzin 866001                                                    Reactor & Proactor */
public void run() {
    try {
        while (!Thread.currentThread().isInterrupted()) {

            // * Il selettore attende che un canale sia pronto per l'operazione
            if (demultiplexer.select() != 0)
                continue;
            Set<SelectionKey> readyHandles = demultiplexer.selectedKeys();
            Iterator<SelectionKey> handleIterator = readyHandles.iterator();

            while (handleIterator.hasNext()) {
                SelectionKey handle = handleIterator.next();

                // * Eseguo il dispatching dell'evento
                dispatch(handle);
            }                           private void dispatch(SelectionKey handle) throws Exception {
{...}                                       // * Chiamo l'handler approporiato per l'evento
                                            EventHandler handler = registeredHandlers.get(handle.interestOps());
                                            handler.handleEvent(handle);
                                        }
                                    }
```

```java
/* MauroZorzin 866001                                                    Reactor & Proactor */

public class AcceptEventHandler implements EventHandler {
    private Selector demultiplexer;

    public AcceptEventHandler(Selector demultiplexer) {
        this.demultiplexer = demultiplexer;
    }

    public void handleEvent(SelectionKey handle) throws Exception {
        log.info("Accept");
        ServerSocketChannel serverSocketChannel = (ServerSocketChannel) handle.channel();
        SocketChannel socketChannel = serverSocketChannel.accept();
        if (socketChannel != null) {
            socketChannel.configureBlocking(false);
            socketChannel.register(
                    demultiplexer, SelectionKey.OP_READ);
        }
                                    public interface EventHandler {
    }
}                                       public void handleEvent(SelectionKey handle) throws Exception;
                                    }
```

```java
/* MauroZorzin 866001                                                    Reactor & Proactor */

  public void handleEvent(SelectionKey handle) throws Exception {
        log.info("Read Event Handler");

        SocketChannel socketChannel = (SocketChannel) handle.channel();

        socketChannel.read(inputBuffer);
        String s = MsgCodec.decode(inputBuffer);
        inputBuffer.clear();
        System.out.println("Received message from client : " + s);

        // ? Simulo esecuzione di codice
        Thread.sleep(10000);

        // * Passo il messaggio al demultiplexer
        socketChannel.register(demultiplexer, SelectionKey.OP_WRITE, MsgCodec.encode(s));

    }

```

```java
/* MauroZorzin 866001                                                Reactor & Proactor */


    public void handleEvent(SelectionKey handle) throws Exception {
            log.info("Write Event Handler");

            SocketChannel socketChannel = (SocketChannel) handle.channel();
            ByteBuffer inputBuffer = (ByteBuffer) handle.attachment();

            // ? Simulo esecuzione di codice
            Thread.sleep(10000);

            socketChannel.write(inputBuffer);
            socketChannel.close();
        }


```

```
/* MauroZorzin 866001                                    Reactor & Proactor */
Diagramma di Sequenza
```



```
/* MauroZorzin 866001                                    Reactor & Proactor */

Pro:

-Separation of concerns

-Improve modularity, reusability, and configurability of
event-driven applications

-Improves application portability

-Provides coarse-grained concurrency control(Concorrenza threadSafe)

                                        Con:

                                            -Non-preemptive

                                            -Hard to debug

                                            -Restricted applicability (Richiede
                                                        il supporto dell'OS)
```

/* MauroZorzin 866001 Reactor & Proactor */

Tipi di Demultiplexer

- select(): Portabile ma inefficiente O(n) con la selezione del descrittore limitato a 1024 descrittori, apolidi
- poll(): Consente un controllo più granulare degli eventi, ma comunque O(n) selezione del descrittore, senza stato
- epool(): Conserva le informazioni, set di descrittori dinamici, efficiente O(1), solo su piattaforme Linux
- kqueue(): Meccanismo più generale, selezione del descrittore O(1), solo attivo Sistemi OS X e FreeBSD
- WaitForMultipleObjects(): Funziona su più tipi di sincronizzazione oggetti, solo su Windows

Varianti

- Concurrent Event Handlers: Per migliorare le prestazioni, i gestori di eventi può essere eseguiti su di un proprio thread, invece di prendere in prestito il thread Reactor

- Concurrent synchronous event demultiplexers: Chiamate sugli'handler effettuate da più thread, per migliorare il throughput(portata)

- Re-entrant Reactors: Il ciclo di eventi viene richiamato da Concrete Event Handler reattivi

- Integrated demultiplexing of Timer and I/O events: Permette di registrare gestori di eventi basati sul tempo.

```
/* MauroZorzin 866001                                    Reactor & Proactor */
```

Proactor

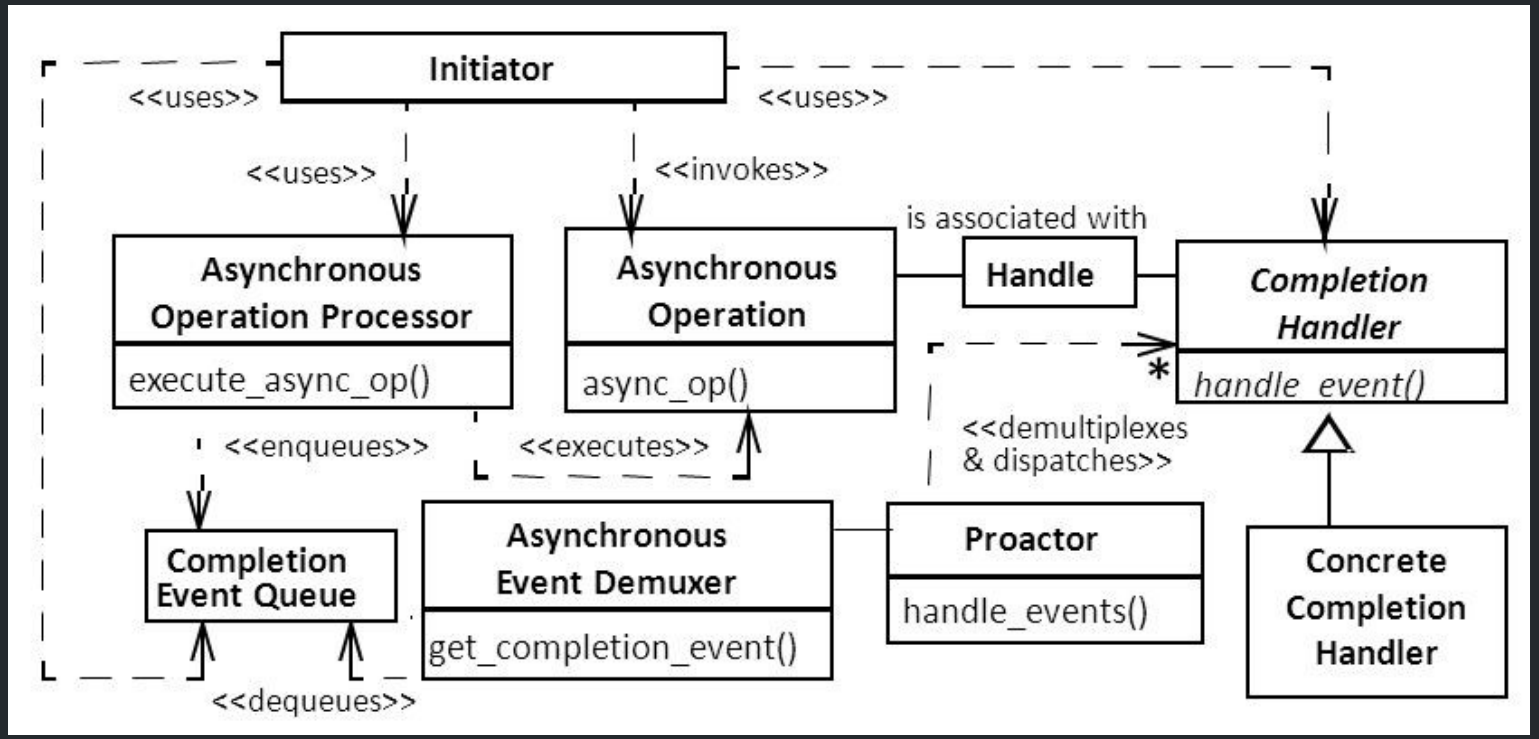
Proactor

Il modello Proactor consente alle applicazioni event-driven di eseguire il demultiplex e di gestire e distribuire i servizi ai richiedenti in modo asincorno

Quello di prima, ma asincorno

/* MauroZorzin 866001 Reactor & Proactor */

Anche in questo caso necessitiamo di un framework: Java NIO 2.0 framework



```
/* MauroZorzin 866001                                    Reactor & Proactor */

Esempio del telefono


- Chiami un amico -> Initiator

- non rispondere. Lasci un messaggio ->  asynchronous operation processor

- In attesa della richiamata, puoi fare altre cose.

- Il tuo amico ascolta la segreteria telefonica -> completion event of the
                                                   asynchronous operation

- ti richiama -> Proactor

- parli insieme -> Gestore di completamento, specifico handle_event()
```

/* MauroZorzin 866001 Reactor & Proactor */

- Handle: Identifica le fonti di eventi che possono essere prodotte o accodate da richieste esterne o interne
- Completion Handler: Definisce un'interfaccia con un insieme di metodi hook
- Concrete Completion Handler: Implementa il metodo hook
- Proactor: Fornisce il ciclo di eventi dell'applicazione, demultiplexa gli eventi completati ai relativi gestori e invia metodi hook per elaborare i risultati
- Asynchronous Event Demultiplexer: Funzione che che attende e coordina gli eventi in entrata, uscita e completati

/* MauroZorzin 866001 Reactor & Proactor */

- Completion Event Queue: Memorizza nel buffer gli eventi di completamento mentre sono in attesa di essere demultiplexati ai rispettivi gestori di completamento

- Asynchronous operations: Rappresentano operazioni di potenzialmente lunga durata utilizzate dai servizi per conto dell'applicazione

- Asynchronous Operation Processor: Esegue operazioni asincrone richiamate sugli handle, genera il rispettivo evento di completamento e lo mette in coda

- Initiator: entità locale all'applicazione, avvia un operazione asincrona, registra i gestori di completamento e un Proactor con un processore di operazioni asincrone, che lo notifica quando le operazioni sono completate

```java
 /* MauroZorzin 866001                                                    Reactor & Proactor */
public class ProactorInitiator {
    public static int ASYNC_SERVER_PORT = 4333;

    public void initiateProactiveServer(int port) throws IOException {

        final AsynchronousServerSocketChannel listener = AsynchronousServerSocketChannel.open()
                .bind(new InetSocketAddress(port));
        AcceptCompletionHandler acceptCompletionHandler = new AcceptCompletionHandler(listener);

        SessionState state = new SessionState();
        listener.accept(state, acceptCompletionHandler);
    }

    public static void main(String[] args) {
        try {
            System.out.println("Async server listening on port : " + ASYNC_SERVER_PORT);
            new ProactorInitiator().initiateProactiveServer(ASYNC_SERVER_PORT);
        } catch (IOException e) {
            e.printStackTrace();
        }


```

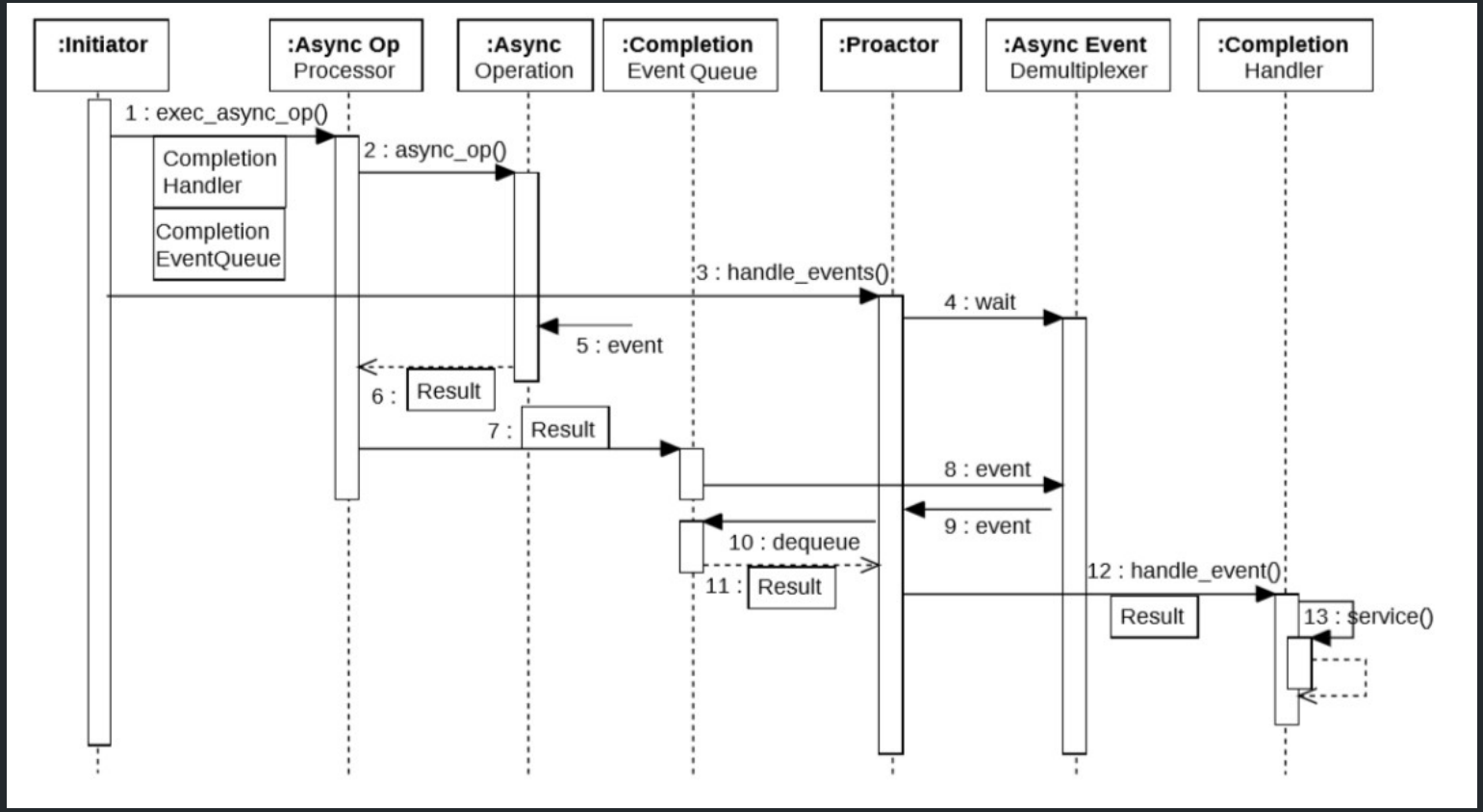
```java
/* MauroZorzin 866001                                                    Reactor & Proactor */

    public class AcceptCompletionHandler implements CompletionHandler<AsynchronousSocketChannel, SessionState> {
        private final AsynchronousServerSocketChannel listener;

        public AcceptCompletionHandler(AsynchronousServerSocketChannel listener) {
            this.listener = listener;
        }


                    @Override
                    public void completed(AsynchronousSocketChannel socketChannel, SessionState sessionState) {
                        // accept the next connection
                        SessionState newSessionState = new SessionState();
                        listener.accept(newSessionState, this);

                        // handle this connection
                        ByteBuffer inputBuffer = ByteBuffer.allocate(2048);
                        ReadCompletionHandler readCompletionHandler = new ReadCompletionHandler(socketChannel, inputBuffer);
                        socketChannel.read(inputBuffer, sessionState, readCompletionHandler);
                    }

```

```java
/* MauroZorzin 866001                                                    Reactor & Proactor */
public class ReadCompletionHandler implements CompletionHandler<Integer, SessionState> {
    private final AsynchronousSocketChannel socketChannel;
    private final ByteBuffer inputBuffer;

    public ReadCompletionHandler(AsynchronousSocketChannel socketChannel, ByteBuffer inputBuffer) {
        this.socketChannel = socketChannel;
        this.inputBuffer = inputBuffer;
    }

    @Override
    public void completed(Integer bytesRead, SessionState sessionState) {
        byte[] buffer = new byte[bytesRead];
        inputBuffer.rewind();
        // Rewind the input buffer to read from the beginning
        inputBuffer.get(buffer);
        String message = new String(buffer);
        System.out.println("Received message from client : " + message);
        WriteCompletionHandler writeCompletionHandler = new WriteCompletionHandler(socketChannel);
        ByteBuffer outputBuffer = ByteBuffer.wrap(buffer);
        socketChannel.write(outputBuffer, sessionState, writeCompletionHandler);
    }
```

```java
/* MauroZorzin 866001                                                    Reactor & Proactor */


    public class WriteCompletionHandler implements CompletionHandler<Integer, SessionState> {
        private final AsynchronousSocketChannel socketChannel;

        public WriteCompletionHandler(AsynchronousSocketChannel socketChannel) {
            this.socketChannel = socketChannel;
        }

        @Override
        public void completed(Integer bytesWritten, SessionState attachment) {
            try {
                System.out.println("Closing connection with client");
                socketChannel.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }


```

```java
/* MauroZorzin 866001                                                    Reactor & Proactor */


  public class SessionState {
      private final Map<String, String> sessionProps = new ConcurrentHashMap<String, String>();

      public String getProperty(String key) {
          return sessionProps.get(key);
      }

      public void setProperty(String key, String value) {
          sessionProps.put(key, value);
      }

  }
```

```
/* MauroZorzin 866001                                    Reactor & Proactor */
Diagramma di Sequenza
```



/* MauroZorzin 866001 Reactor & Proactor */

Pro:

- Increase separation of concerns
- Improve application portability
- Encapsulate concurrency mechanisms
- Concurrency policy independent from threading policy
- Increase performance
- Simplify application synchronization

Con:
- No control over scheduling of operations
- Efficiency depends on the platform
- Complexity of debugging and testing

/* MauroZorzin 866001 Reactor & Proactor */

Varianti

- Asynchronous Completion Handlers: Per migliorare le prestazioni, i gestori di completamento potrebbero fungere da iniziatori e invocare procedure di lunga durata sincrone

- Concurrent asynchronous event Demultiplexer: Un pool di thread che condividere un evento Demultiplexer asincrono, particolarmente scalabile

- Shared Completion handlers: Più operazioni asincrone avviate contemporaneamente possono condividere lo stesso gestore di completamento concreto

- Asynchronous operation Processor emulation: Nel sistema operativo piattaforme che non esportano operazioni asincrone nelle applicazioni.

```
/* MauroZorzin 866001                                    Reactor & Proactor */
```

Tutto il codice è disponibile su GitHub:

https://github.com/MauroZorzin/Reator-And-Proactor-JavaPattern.git

Risorse utili:

https://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf

https://wiki.sch.bme.hu/images/5/50/Sznikak_Pattern-Oriented-SA_vol2.pdf

https://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf

https://dzone.com/articles/the-hollywood-principle

https://github.com/kasun04/nio-reactor

https://en.wikipedia.org/wiki/Non-blocking_I/O_(Java)

https://www.javacodegeeks.com/2012/08/io-demystified.html

https://www.boost.org/doc/libs/1_47_0/doc/html/boost_asio/overview/core/async.html

http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformatica/tdp/tpd_reactor_proactor.pdf