



1 /* MauroZorzin 866001

Reactor & Proactor */

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

Reactor & Proactor

concurrency architecture patterns

Università Degli Studi Milano Bicocca



```
1  /* MauroZorzin 866001
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

```
Reactor & Proactor */
```

Reactor



```
1  /* MauroZorzin 866001
```

```
Reactor & Proactor */
```

```
2
```

```
3
```

```
4
```

Reactor_(Dispatcher, Notifier)

```
5
```

```
6
```

```
7
```

```
8
```

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

```
10
```

```
11
```

```
12
```

```
13
```

```
14
```

```
15
```

```
16
```

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

```
17
```

```
18
```

```
19
```

```
20
```

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

```
21
```

```
22
```



```
1  /* MauroZorzin 866001
```

```
Reactor & Proactor */
```

```
2
```

```
3
```

```
4
```

```
5
```

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

```
8
```

```
9
```

```
10      Hollywood Principle - Don't Call Us, We'll Call You!
```

```
11
```

```
12
```

```
13
```

```
14
```

```
15  Anche il pattern Observer segue lo stesso principio
```

```
16
```

```
17
```

```
18
```

```
19
```

```
20
```

```
21
```

```
22
```



1 /* MauroZorzin 866001

Reactor & Proactor */

2

3 Componenti dell'architettura ad eventi

4

5

6

- Event sources: Rileva e recupera gli eventi

7

8

9

10

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

13

14

15

16

17

18

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

19

20

21

22



```
1  /* MauroZorzin 866001
```

```
Reactor & Proactor */
```

```
2
```

```
3  Principali differenze rispetto al tradizionale flusso di  
4  controllo:  
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
- Il comportamento è causato da eventi asincroni
```

```
10
```

```
11
```

```
- La maggior parte degli eventi deve essere gestita tempestivamente
```

```
12
```

```
13
```

```
- Macchine a stati finiti per il controllo dell'elaborazione degli eventi
```

```
14
```

```
15
```

```
- Nessun controllo sull'ordine di arrivo degli eventi
```

```
16
```

```
17
```

```
18
```

```
19
```

```
20
```

```
21
```

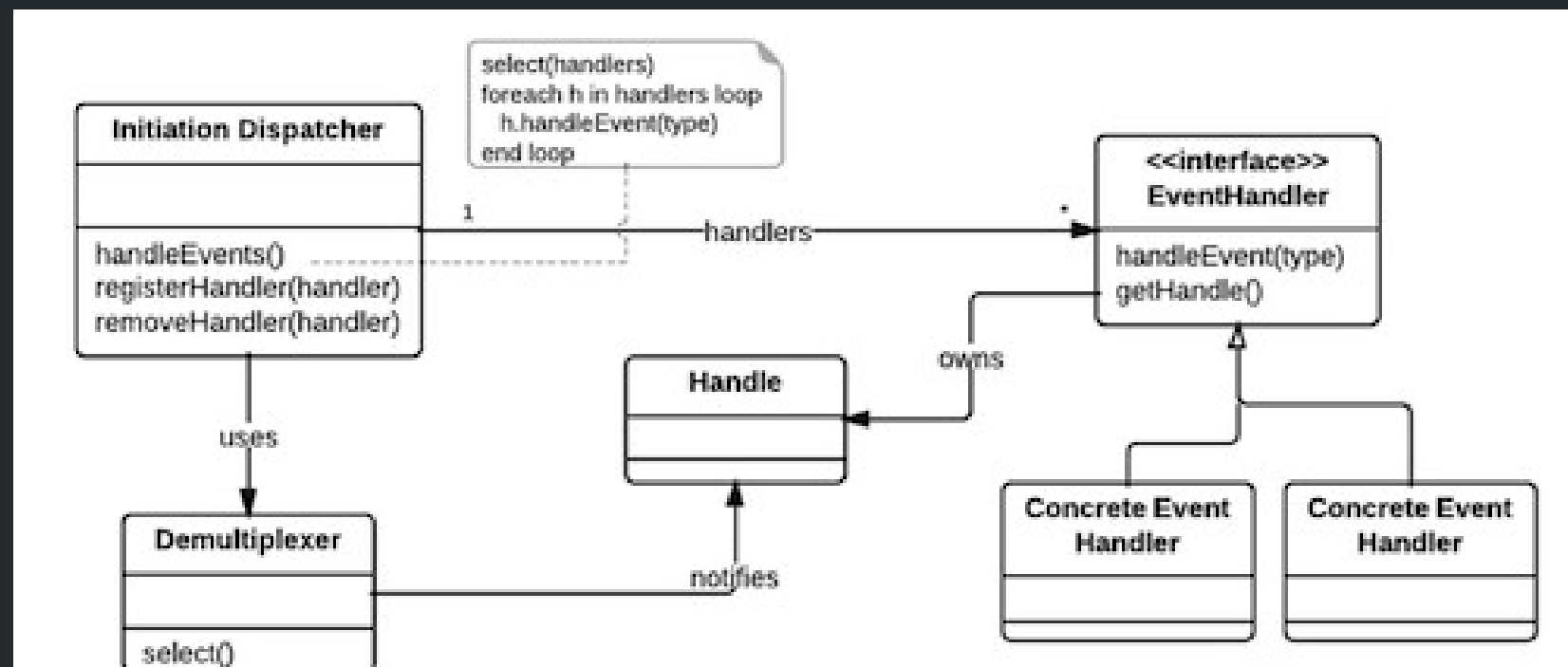
```
22
```



1 /* MauroZorzin 866001

Reactor & Proactor */

3 Cominciamo dalla struttura:





```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2
3  • Handle: Identifica le fonti di eventi che possono essere prodotte o
4             accodate da richieste esterne o interne
5
6  • Event Handler: Definisce un'interfaccia con un insieme di metodi hook
7
8  • Concrete Event Handler: Specializza il gestore di eventi per un
9                           particolare service e implementa i metodi hook
10
11 • Reactor: Specifica un'interfaccia per registrare e rimuovere Event Handler
12            e Handlers, esegue l'event loop per reagire a ogni
13            evento demultiplicandolo l'handler al gestore di eventi e
14            lanciando l'appropriato hook method
15
16 • Synchronous Event Demultiplexer: Attende il verificarsi di eventi, funge
17            da demultiplexer, le sue funzionalità
18            dipendono dal sistema operativo
19
20
21
22
```




```
1  /* MauroZorzin 866001
```

```
Reactor & Proactor */
```

```
2  Eempio del telefono
```

```
3
```

```
4
```

```
5  - Rete di telecomunicazioni -> Reattore
```

```
6
```

```
7  - Cliente -> Event Handler
```

```
8
```

```
9  - Numero di telefono -> Handler
```

```
10
```

```
11  - Qualcuno chiama il numero -> evento in arrivo
```

```
12
```

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

```
15
```

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

```
18
```

```
19
```

```
20
```

```
21
```

```
22
```



```
1  /* MauroZorzin 866001
```

```
Reactor & Proactor */
```

```
2
```

```
3  Per implementare un Reactor in Java necessitiamo di un freamework
```

```
4
```

```
5  Java NIO framework
```

```
6
```

```
7
```

```
8
```

```
9
```

```
10
```

```
11
```

```
12
```

```
13
```

```
14
```

```
15
```

```
16
```

```
17
```

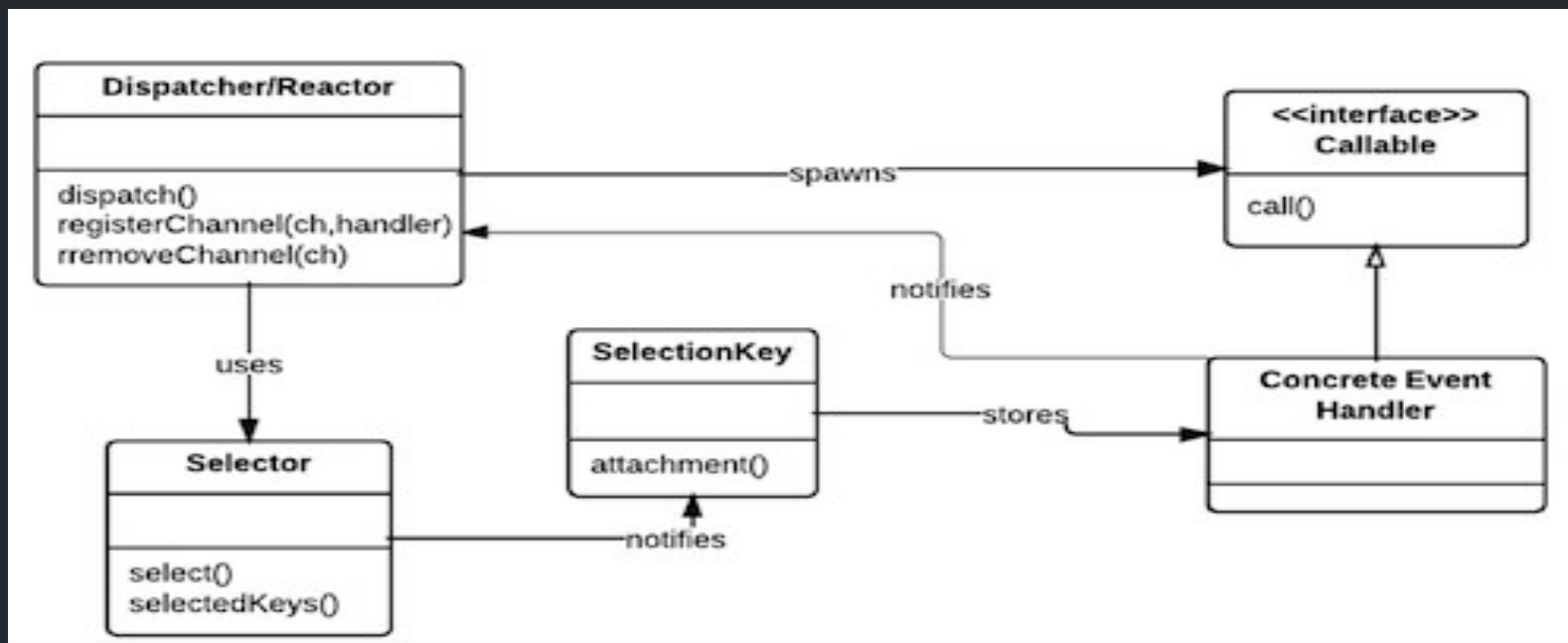
```
18
```

```
19
```

```
20
```

```
21
```

```
22
```





```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2
3  public static void main(String[] args) throws Exception {
4
5      Socket clientSocket = new Socket("localhost", 7070);
6      PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
7      BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()))
8      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
9
10     String str = br.readLine();
11
12
13     out.println(str);
14     String str2 = in.readLine();
15     System.out.println("Server says: " + str2);
16
17     in.close();
18     out.close();
19     clientSocket.close();
20 }
21
22
```



```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2
3  public class AcceptEventHandler implements EventHandler {
4      private Selector demultiplexer;
5
6      public AcceptEventHandler(Selector demultiplexer) {
7          this.demultiplexer = demultiplexer;
8      }
9
10     public void handleEvent(SelectionKey handle) throws Exception {
11         log.info("Accept");
12         ServerSocketChannel serverSocketChannel = (ServerSocketChannel) handle.channel();
13         SocketChannel socketChannel = serverSocketChannel.accept();
14         if (socketChannel != null) {
15             socketChannel.configureBlocking(false);
16             socketChannel.register(
17                 demultiplexer, SelectionKey.OP_READ);
18         }
19     }
20 }
21
22     public interface EventHandler {
23         public void handleEvent(SelectionKey handle) throws Exception;
24     }
```



```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2
3  public void handleEvent(SelectionKey handle) throws Exception {
4      log.info("Read Event Handler");
5
6      SocketChannel socketChannel = (SocketChannel) handle.channel();
7
8      socketChannel.read(inputBuffer);
9      String s = MsgCodec.decode(inputBuffer);
10     inputBuffer.clear();
11     System.out.println("Received message from client : " + s);
12
13     // ? Simulo esecuzione di codice
14     Thread.sleep(10000);
15
16     // * Passo il messaggio al demultiplexer
17     socketChannel.register(demultiplexer, SelectionKey.OP_WRITE, MsgCodec.encode(s));
18
19 }
20
21
22
```



```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2
3
4  public void handleEvent(SelectionKey handle) throws Exception {
5      log.info("Write Event Handler");
6
7
8      SocketChannel socketChannel = (SocketChannel) handle.channel();
9      ByteBuffer inputBuffer = (ByteBuffer) handle.attachment();
10
11
12      // ? Simulo esecuzione di codice
13      Thread.sleep(10000);
14
15
16      socketChannel.write(inputBuffer);
17      socketChannel.close();
18  }
19
20
21
22
```



```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2  public class Reactor {
3      private Map<Integer, EventHandler> registeredHandlers = new ConcurrentHashMap<Integer, EventHandler>();
4      private Selector demultiplexer;
5
6      public Reactor() throws Exception {
7          demultiplexer = Selector.open();
8      }
9
10     public Selector getDemultiplexer() {
11         return demultiplexer;
12     }
13
14     // * Rimuove l'evento dal selettore
15     public void removeEventHandler(SelectionKey handle) {
16         registeredHandlers.remove(handle.interestOps());
17     }
18
19     // * Associa l'evento con il suo handler
20     public void registerEventHandler(
21         int eventType, EventHandler eventHandler) {
22         registeredHandlers.put(eventType, eventHandler);
23     }
24
25     // * Associa il canale con il selettore tramite la register
26     public void registerChannel(
27         int eventType, SelectableChannel channel) throws Exception {
28         channel.register(demultiplexer, eventType);
29     }
30 }
```




```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2  public void run() {
3      try {
4          while (!Thread.currentThread().isInterrupted()) {
5
6              // * Il selettore attende che un canale sia pronto per l'operazione
7              if (demultiplexer.select() != 0)
8                  continue;
9              Set<SelectionKey> readyHandles = demultiplexer.selectedKeys();
10             Iterator<SelectionKey> handleIterator = readyHandles.iterator();
11
12             while (handleIterator.hasNext()) {
13                 SelectionKey handle = handleIterator.next();
14
15                 // * Eseguo il dispatching dell'evento
16                 dispatch(handle);
17             }
18             {...}
19
20             private void dispatch(SelectionKey handle) throws Exception {
21                 // * Chiamo l'handler appropriato per l'evento
22                 EventHandler handler = registeredHandlers.get(handle.interestOps());
23                 handler.handleEvent(handle);
24             }
25         }
26     }
27 }
```

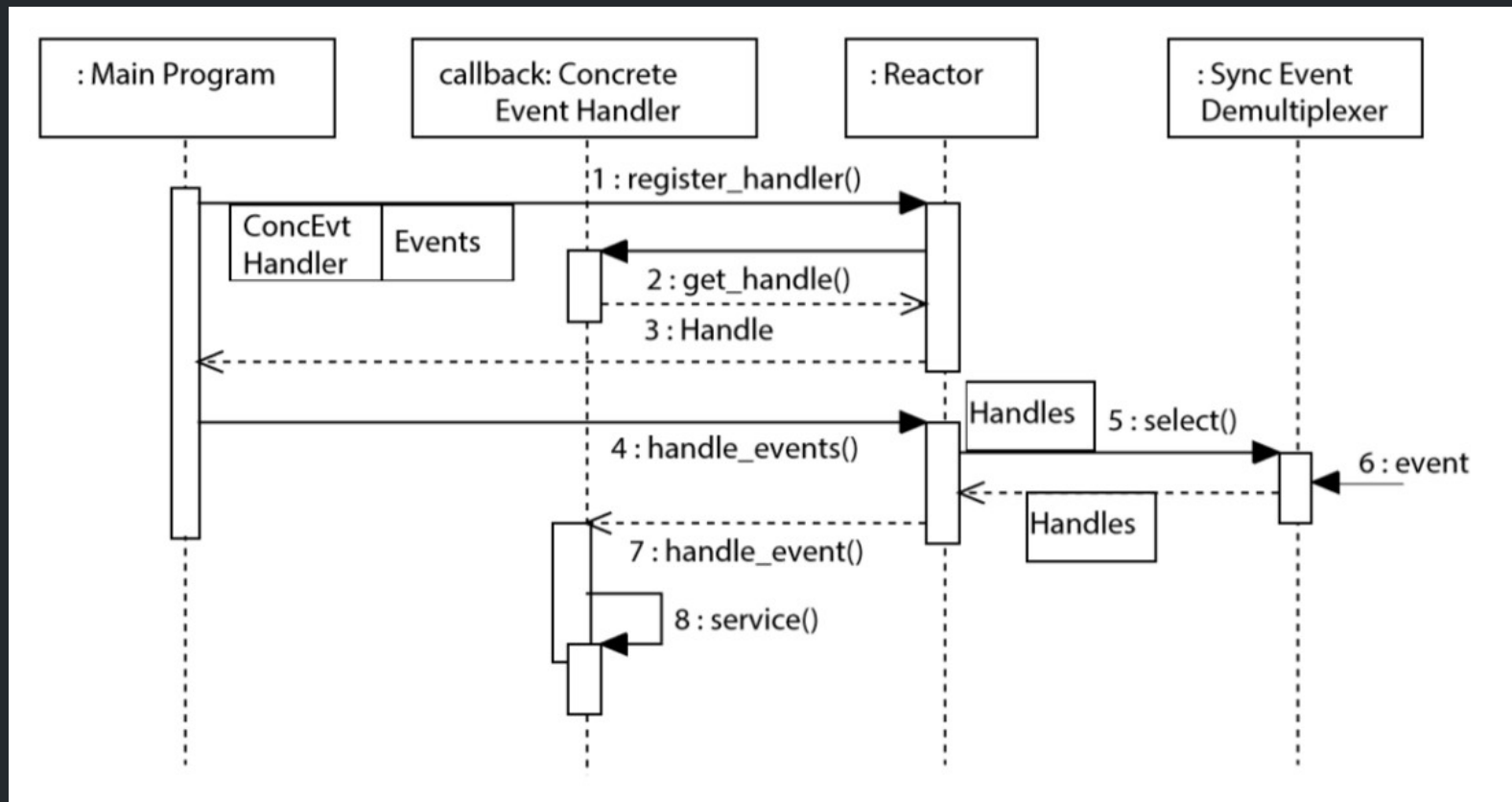



```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2  public class ReactorManager {
3      private static final int SERVER_PORT = 7070;
4
5      public void startReactor(int port) throws Exception {
6
7          ServerSocketChannel server = ServerSocketChannel.open();
8
9          server.socket().bind(new InetSocketAddress(port));
10
11         server.configureBlocking(false);
12
13         Reactor reactor = new Reactor();
14
15         public static void main(String[] args) {
16             log.info("Server Started at port : " + SERVER_PORT);
17             try {
18                 new ReactorManager().startReactor(SERVER_PORT);
19             } catch (Exception e) {
20                 e.printStackTrace();
21             }
22         }
23     }
24
25     // * Associamo il canale con il selettore tramite la register
26     reactor.registerChannel(SelectionKey.OP_ACCEPT, server);
27
28     // * Associamo ad ogni evento il suo handler
29     reactor.registerEventHandler(
30         SelectionKey.OP_ACCEPT, new AcceptEventHandler(
31             reactor.getDemultiplexer()));
32
33     reactor.registerEventHandler(
34         SelectionKey.OP_READ, new ReadEventHandler(
35             reactor.getDemultiplexer()));
36
37     reactor.registerEventHandler(
38         SelectionKey.OP_WRITE, new WriteEventHandler());
39
40     reactor.run(); // * Run the dispatcher loop
41 }
```



```
1  /* MauroZorzin 866001
2  Diagramma di Sequenza
```

```
Reactor & Proactor */
```





1 /* MauroZorzin 866001

Reactor & Proactor */

2

3 Pro:

4

5 -Separation of concerns

6

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

9

10 -Improves application portability

11

12 -Provides coarse-grained concurrency control(Concorrenza threadSafe)

13

14

15 Con:

16

17 -Non-preemptive

18

19 -Hard to debug

20

21 -Restricted applicability (Richiede
22 il supporto dell'OS)



```
1  /* MauroZorzin 866001
```

```
Reactor & Proactor */
```

```
2
```

3 Tipi di Demultiplexer

```
4
```

```
5
```

```
6  • select(): Portabile ma inefficiente  $O(n)$  con la selezione del descrittore  
7             limitato a 1024 descrittori, apolidi
```

```
8
```

```
9  • poll(): Consente un controllo più granulare degli eventi, ma comunque  $O(n)$   
10             selezione del descrittore, senza stato
```

```
11
```

```
12  • epool(): Conserva le informazioni, set di descrittori dinamici, efficiente  
13              $O(1)$ , solo su piattaforme Linux
```

```
14
```

```
15  • kqueue(): Meccanismo più generale, selezione del descrittore  $O(1)$ , solo  
16             attivo Sistemi OS X e FreeBSD
```

```
17
```

```
18  • WaitForMultipleObjects(): Funziona su più tipi di sincronizzazione  
19             oggetti, solo su Windows
```

```
20
```

```
21
```

```
22
```



1 /* MauroZorzin 866001

Reactor & Proactor */

2

3 Varianti

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

- 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.



```
1  /* MauroZorzin 866001
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

```
Reactor & Proactor */
```

Proactor



```
1  /* MauroZorzin 866001
```

```
Reactor & Proactor */
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

```
7
```

```
8
```

```
9
```

```
10
```

```
11
```

```
12
```

```
13
```

```
14
```

```
15
```

```
16
```

```
17
```

```
18
```

```
19
```

```
20
```

```
21
```

```
22
```

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

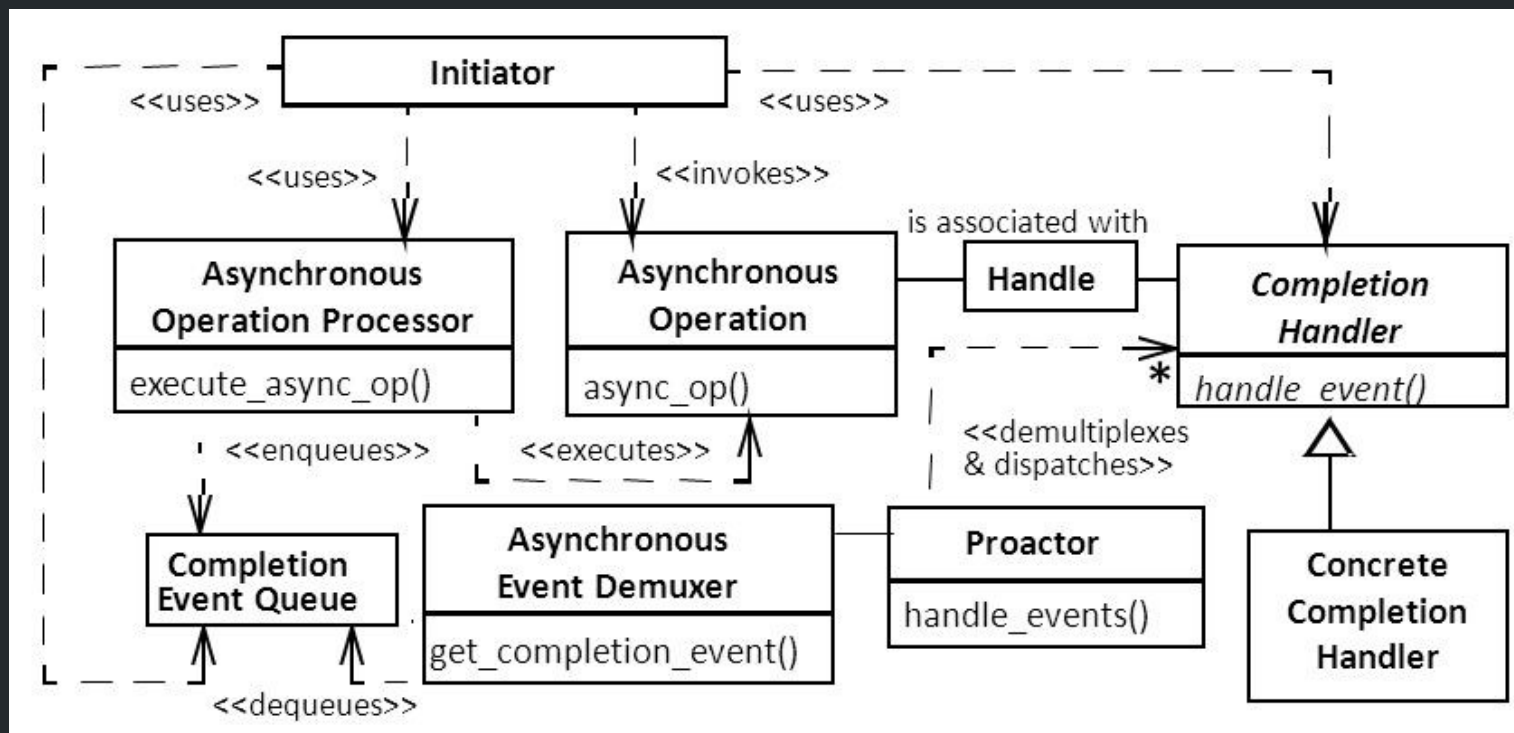
Quello di prima, ma asincorno



```
1  /* MauroZorzin 866001
```

```
Reactor & Proactor */
```

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





```
1  /* MauroZorzin 866001                                Reactor & Proactor */
2
3  • Handle: Identifica le fonti di eventi che possono essere prodotte o
4             accodate da richieste esterne o interne
5
6  • Completion Handler: Definisce un'interfaccia con un insieme di metodi hook
7
8
9  • Concrete Completion Handler: Implementa il metodo hook
10
11 • Proactor: Fornisce il ciclo di eventi dell'applicazione, demultiplexa gli
12             eventi completati ai relativi gestori e invia metodi hook per
13             elaborare i risultati
14
15 • Asynchronous Event Demultiplexer: Funzione che che attende e coordina gli
16             eventi in entrata, uscita e completati
17
18
19
20
21
22
```



```
1  /* MauroZorzin 866001
```

Reactor & Proactor */

- Completion Event Queue: Memorizza nel buffer gli eventi di completamento mentre sono in attesa di essere demultipletzati 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



```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2  Eempio del telefono
3
4
5
6  - Chiami un amico -> Initiator
7
8  - non rispondere. Lasci un messaggio -> asynchronous operation processor
9
10 - In attesa della richiamata, puoi fare altre cose.
11
12 - Il tuo amico ascolta la segreteria telefonica -> completion event of the
13                                                         asynchronous operation
14
15 - ti richiama -> Proactor
16
17 - parli insieme -> Gestore di completamento, specifico handle_event()
18
19
20
21
22
```



```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2
3
4  public class SessionState {
5      private final Map<String, String> sessionProps = new ConcurrentHashMap<String, String>();
6
7      public String getProperty(String key) {
8          return sessionProps.get(key);
9      }
10
11     public void setProperty(String key, String value) {
12         sessionProps.put(key, value);
13     }
14
15 }
16
17
18
19
20
21
22
```



```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2
3  public class AcceptCompletionHandler implements CompletionHandler<AsynchronousSocketChannel, SessionState> {
4      private final AsynchronousServerSocketChannel listener;
5
6      public AcceptCompletionHandler(AsynchronousServerSocketChannel listener) {
7          this.listener = listener;
8      }
9
10
11      @Override
12      public void completed(AsynchronousSocketChannel socketChannel, SessionState sessionState) {
13          // accept the next connection
14          SessionState newSessionState = new SessionState();
15          listener.accept(newSessionState, this);
16
17          // handle this connection
18          ByteBuffer inputBuffer = ByteBuffer.allocate(2048);
19          ReadCompletionHandler readCompletionHandler = new ReadCompletionHandler(socketChannel, inputBuffer);
20          socketChannel.read(inputBuffer, sessionState, readCompletionHandler);
21      }
22
```



```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2  public class ReadCompletionHandler implements CompletionHandler<Integer, SessionState> {
3      private final AsynchronousSocketChannel socketChannel;
4      private final ByteBuffer inputBuffer;
5
6      public ReadCompletionHandler(AsynchronousSocketChannel socketChannel, ByteBuffer inputBuffer) {
7          this.socketChannel = socketChannel;
8          this.inputBuffer = inputBuffer;
9      }
10
11     @Override
12     public void completed(Integer bytesRead, SessionState sessionState) {
13         byte[] buffer = new byte[bytesRead];
14         inputBuffer.rewind();
15         // Rewind the input buffer to read from the beginning
16         inputBuffer.get(buffer);
17         String message = new String(buffer);
18         System.out.println("Received message from client : " + message);
19         WriteCompletionHandler writeCompletionHandler = new WriteCompletionHandler(socketChannel);
20         ByteBuffer outputBuffer = ByteBuffer.wrap(buffer);
21         socketChannel.write(outputBuffer, sessionState, writeCompletionHandler);
22     }
```

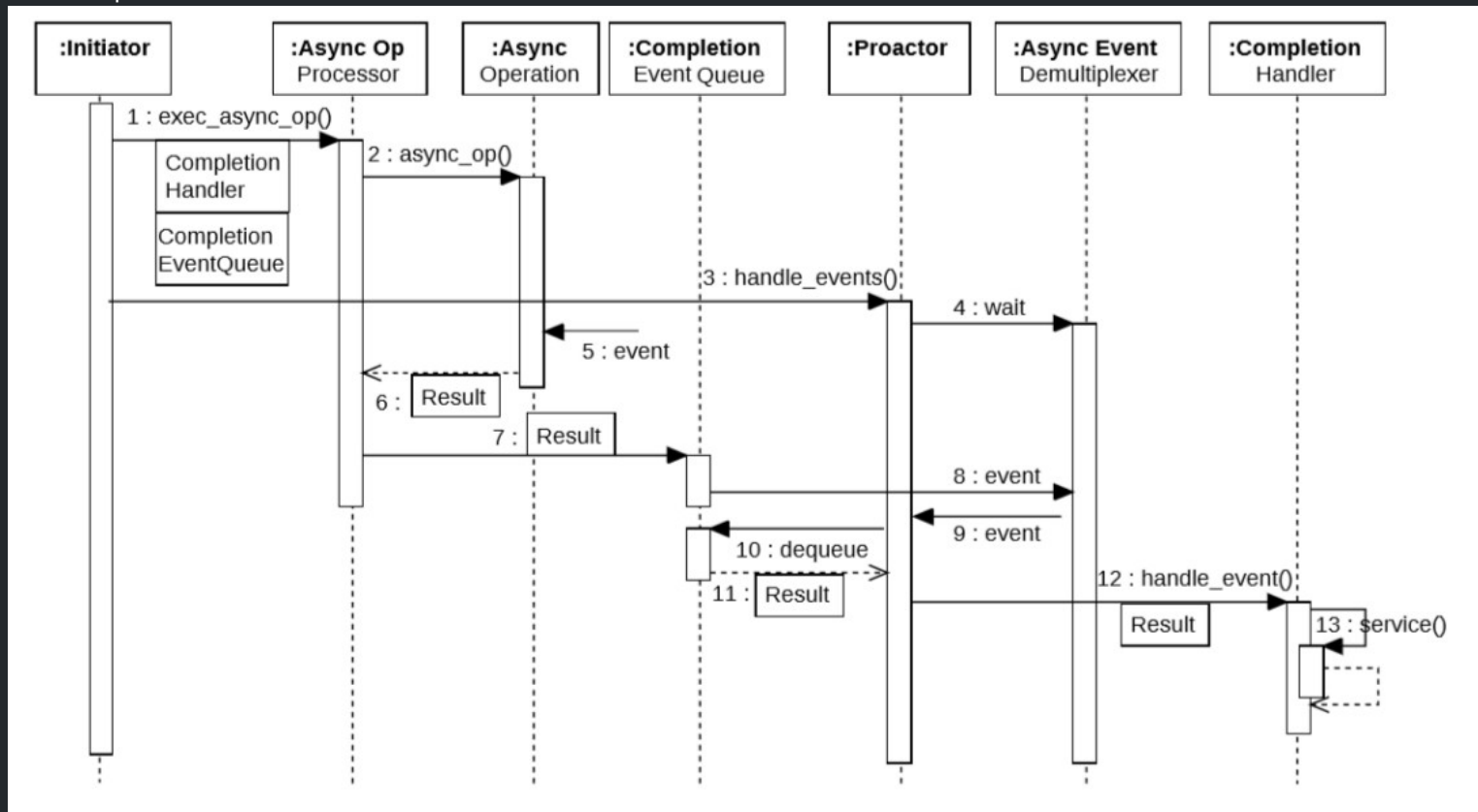


```
1  /* MauroZorzin 866001                                     Reactor & Proactor */
2
3
4  public class WriteCompletionHandler implements CompletionHandler<Integer, SessionState> {
5      private final AsynchronousSocketChannel socketChannel;
6
7      public WriteCompletionHandler(AsynchronousSocketChannel socketChannel) {
8          this.socketChannel = socketChannel;
9      }
10
11     @Override
12     public void completed(Integer bytesWritten, SessionState attachment) {
13         try {
14             System.out.println("Closing connection with client");
15             socketChannel.close();
16         } catch (IOException e) {
17             e.printStackTrace();
18         }
19     }
20
21
22
```



1 /* MauroZorzin 866001
2 Diagramma di Sequenza

Reactor & Proactor */





1 /* MauroZorzin 866001

Reactor & Proactor */

2 Pro:

- 3
- 4 - Increase separation of concerns
 - 5 - Improve application portability
 - 6 - Encapsulate concurrency mechanisms
 - 7 - Concurrency policy independent from threading policy
 - 8 - Increase performance
 - 9 - Simplify application synchronization
- 10

11

12

13 Con:

- 14 - No control over scheduling of operations
 - 15 - Efficiency depends on the platform
 - 16 - Complexity of debugging and testing
- 17
- 18
- 19
- 20
- 21
- 22



1 /* MauroZorzin 866001

Reactor & Proactor */

2 Varianti

- 3
- 4
- 5 • Asynchronous Completion Handlers: Per migliorare le prestazioni,
6 i gestori di completamento potrebbero
7 fungere da iniziatori e invocare
8 procedure di lunga durata sincrone
9
- 10 • Concurrent asynchronous event Demultiplexer: Un pool di thread che
11 condividere un evento Demultiplexer
12 asincrono, particolarmente scalabile
13
- 14 • Shared Completion handlers: Più operazioni asincrone avviate
15 contemporaneamente possono condividere lo
16 stesso gestore di completamento concreto
17
- 18 • Asynchronous operation Processor emulation: Nel sistema operativo
19 piattaforme che non esportano
20 operazioni asincrone nelle applicazioni.
21
22



```
1  /* MauroZorzin 866001
```

```
Reactor & Proactor */
```

```
2
```

```
3 Tutto il codice è disponibile su GitHub:
```

```
4
```

```
5
```

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

```
7
```

```
8 Risorse utili:
```

```
9
```

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

```
11
```

```
12 https://wiki.sch.bme.hu/images/5/50/Sznikak\_Pattern-Oriented-SA\_vol2.pdf
```

```
13
```

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

```
15
```

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

```
17
```

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

```
19
```

```
20 https://en.wikipedia.org/wiki/Non-blocking\_I/O\_\(Java\)
```

```
21
```

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

```
23
```

```
24 https://www.boost.org/doc/libs/1\_47\_0/doc/html/boost\_asio/overview/core/async.html
```

```
25
```

```
26 http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistraleinformatica/tdp/tpd\_reactor\_proactor.pdf
```