Realizzazione di oggetti remoti e Proxy-Skeleton in Java

Advanced Computer Programming

Prof. Luigi De Simone

Sommario

- Client-server e socket;
- Il lato client (Proxy);
- Il lato server (Skeleton):
  - realizzazione per delega;
  - realizzazione per ereditarietà.
- Esempio completo con socket UDP.

Richiamo: Applicazione client-server con socket

In un'applicazione con comunicazione basata su socket, client e server prevedono meccanismi di:

- connessione;
- *esternalizzazione* dei dati;
- invio-ricezione delle richieste;
- ricostruzione dei valori ricevuti.

Potenziali rischi:

– l'implementazione dei meccanismi di comunicazione "distrae" dalla realizzazione delle funzionalità effettive dell'applicazione;

– sovrapposizione della logica i) applicativa con quella di ii) comunicazione client-server.

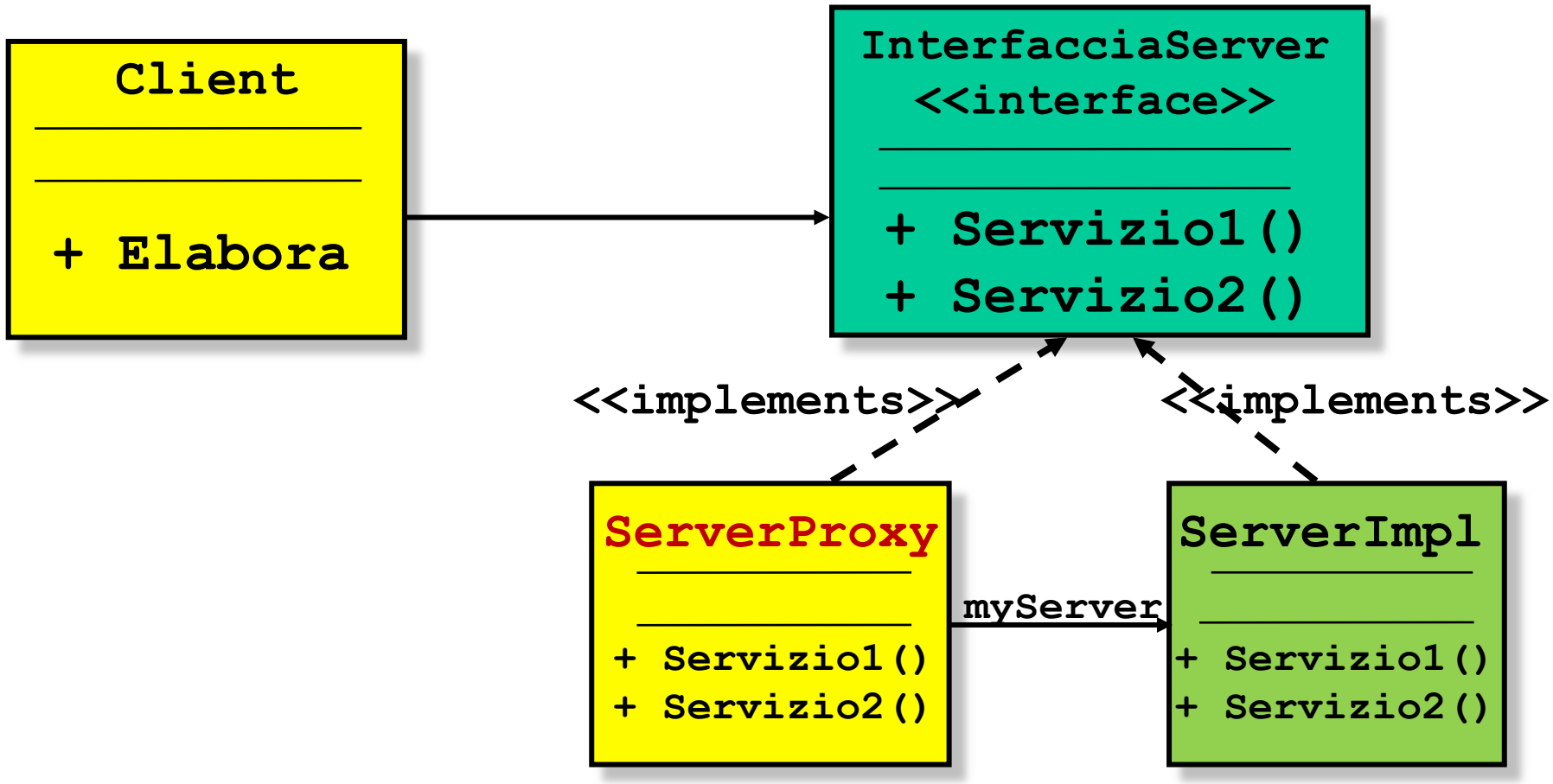
Richiamo: Applicazione client-server con socket

Il programmatore lato client dovrebbe concentrarsi sulla logica applicativa (invocando i servizi del server specificati da un'interfaccia), e separare la logica applicativa dai meccanismi di comunicazione/interazione con il server

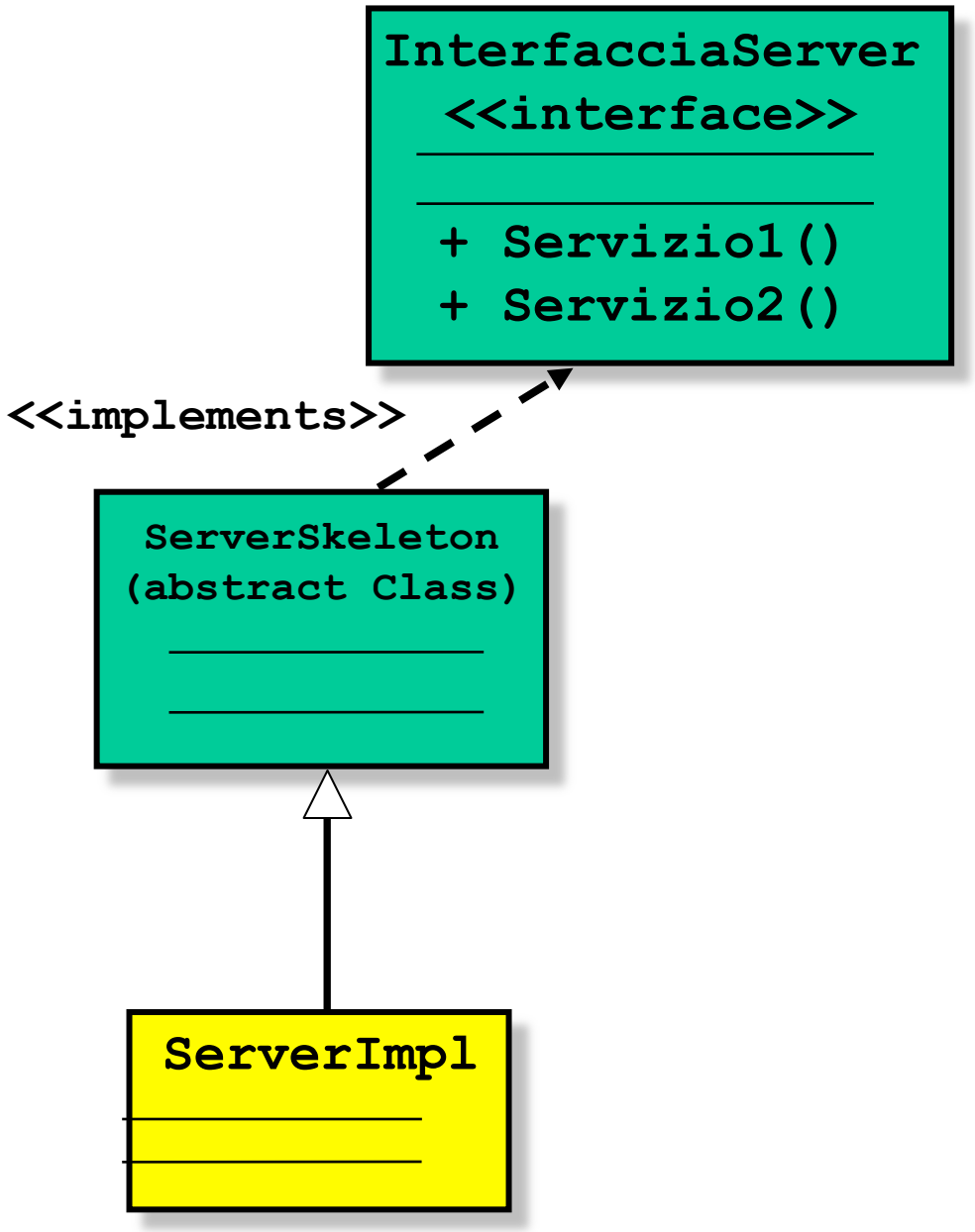
Analogamente, il programmatore lato server dovrebbe concentrarsi sulla codifica dei servizi offerti, separandola dai meccanismi di comunicazione/interazione con il client.

Richiamo: Il pattern Proxy

L'implementazione del pattern Proxy ha la seguente vista statica:



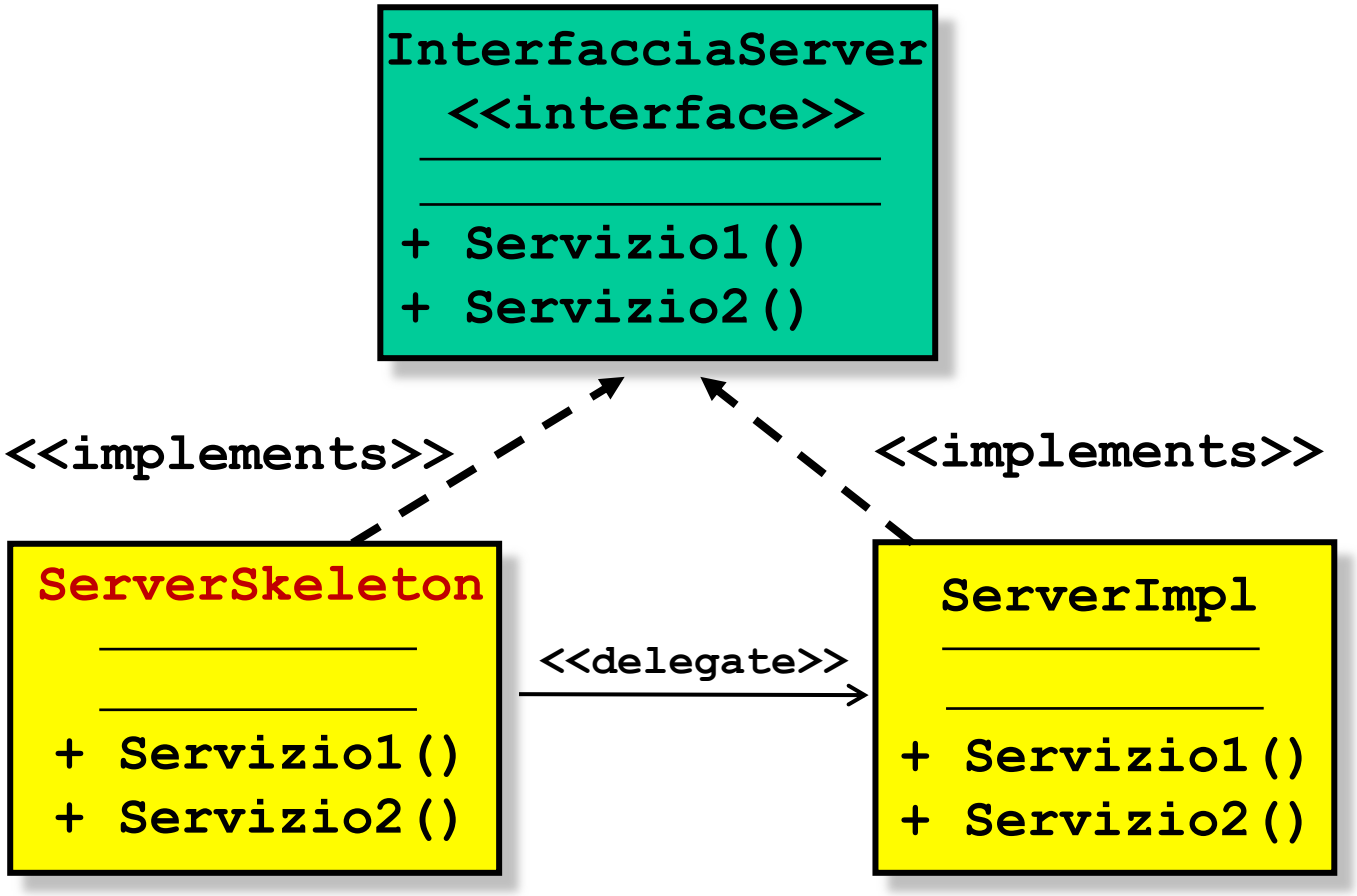
Richiamo: Skeleton per Ereditarietà



- **Lo skeleton può essere implementato per ereditarietà**
  - la classe astratta **Skeleton implementa solo gli opportuni schemi di comunicazione**, ma lascia senza implementazione i metodi dell'interfaccia

- ServerImpl è una **sottoclasse dello skeleton** e fornisce implementazione ai metodi astratti.

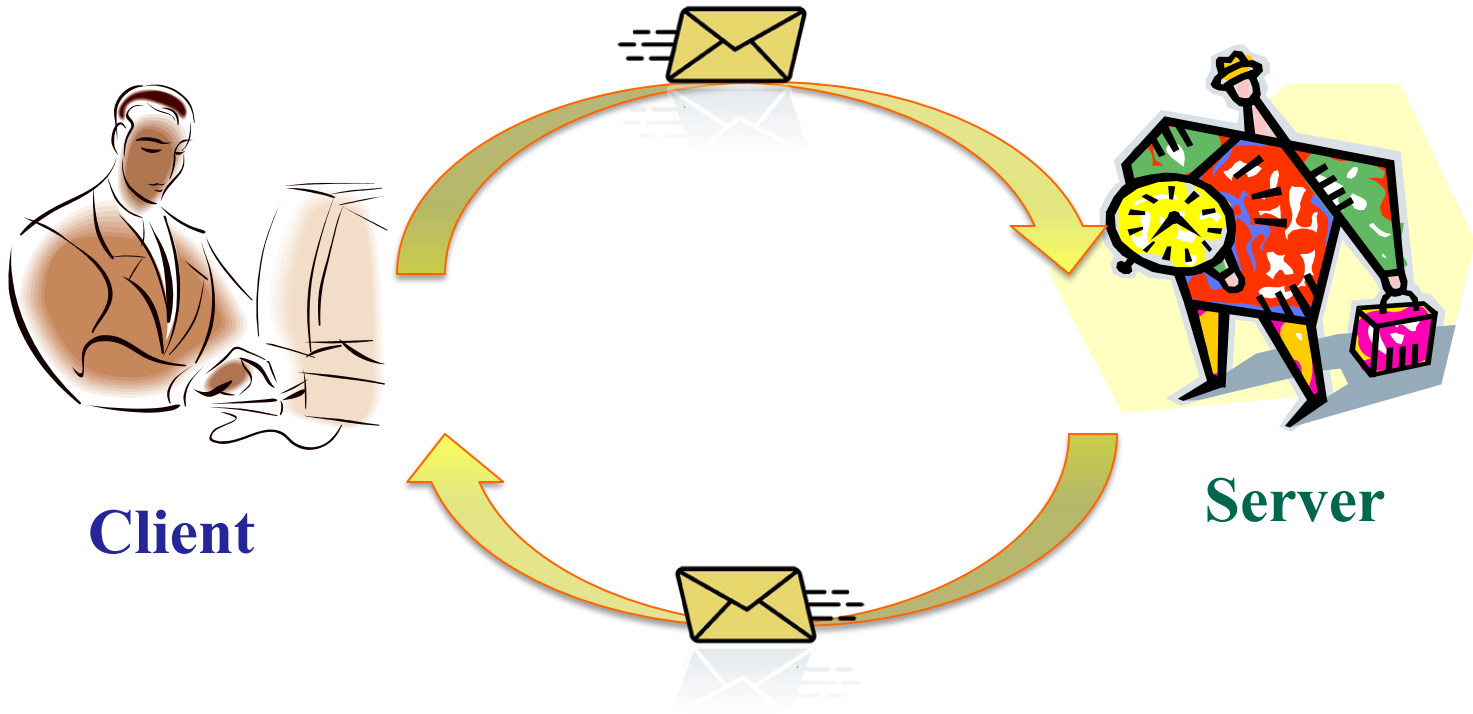
Richiamo: Skeleton per Delega



- Lo **skeleton** può anche essere **implementato per delega**: la classe Skeleton presenta al suo interno un riferimento a **`InterfacciaServer`** (delegate), e i metodi sono così realizzati:

```
void Servizio1() {
    delegate.Servizio1();
}
```

Esempio: Contatore Remoto



- **Il Server gestisce un oggetto Contatore (ossia il servizio reale) che permette l'inizializzazione, l'incremento e l'aggiornamento di una variabile contatore;**
- **Il Client invoca i servizi offerti dal Server, specificati da un'interfaccia.**

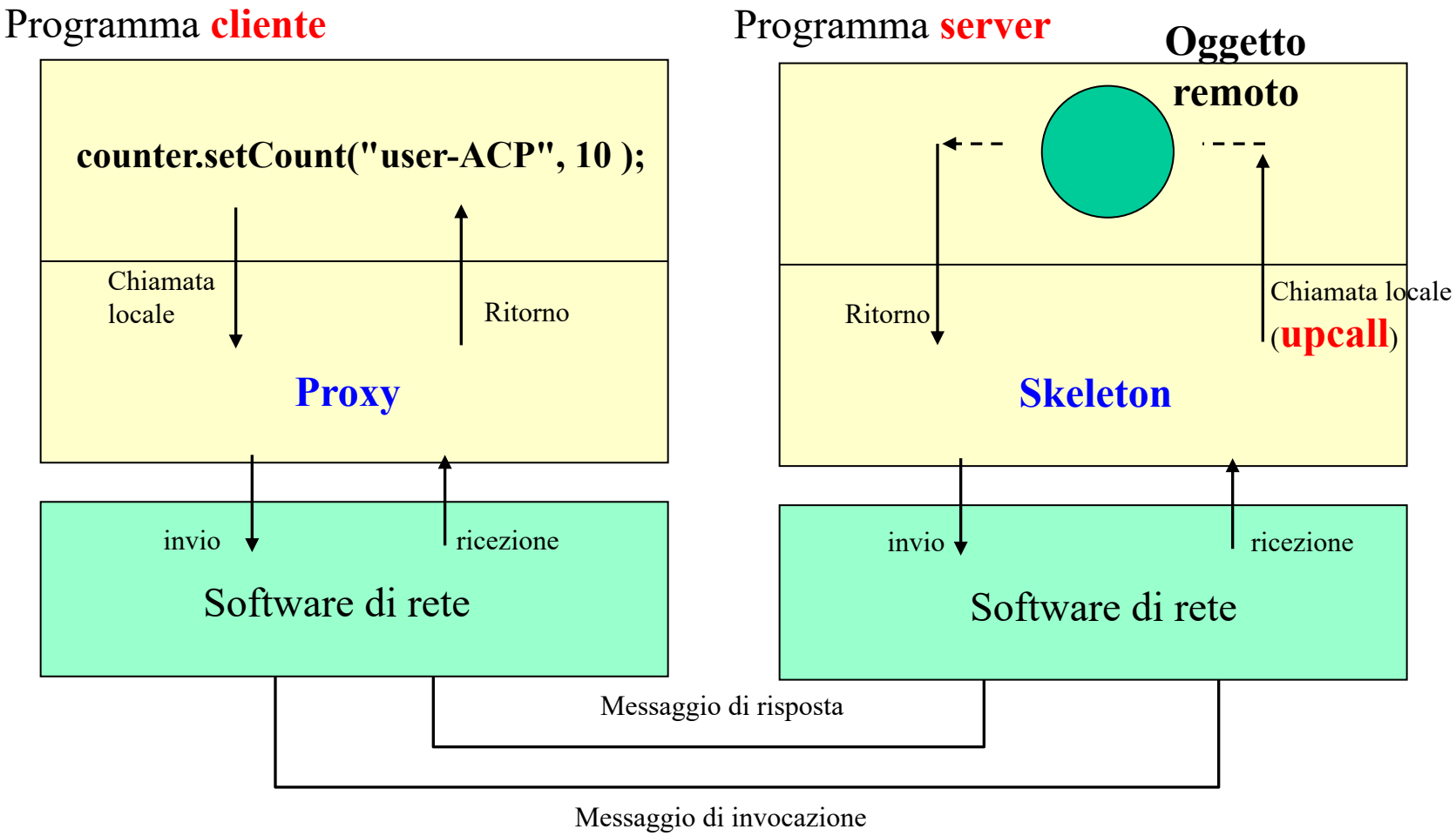
Esempio: Contatore Remoto

Il servizio offerto dal Server è descritto da un'interfaccia Java:

1. setCount: il client identificato dalla String "id" imposta il contatore al valore iniziale s;
2. sum: il valore corrente del contatore è incrementato di s; sum restituisce il valore aggiornato;
3. increment: il valore corrente del contatore viene incrementato di 1; increment restituisce il valore aggiornato;

```
public interface ICounter {
    public void setCount(String id, int s);
    public int sum(int s);
    public int increment();
}
```

Schema concettuale

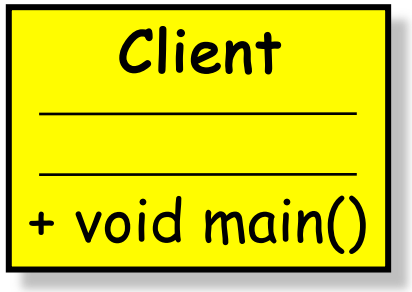


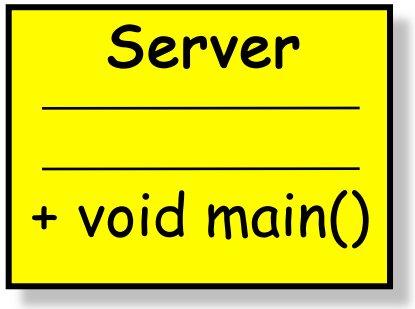
Esempio

Programma client-server che realizza un **contatore remoto** tramite Proxy-Skeleton e socket UDP.

Esempio

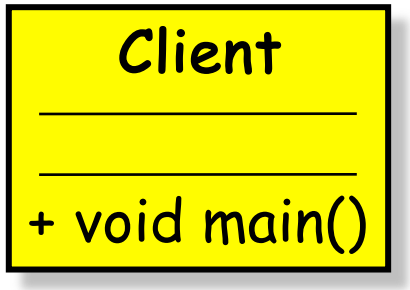
Programma client-server che realizza un contatore remoto tramite Proxy-Skeleton e socket UDP.

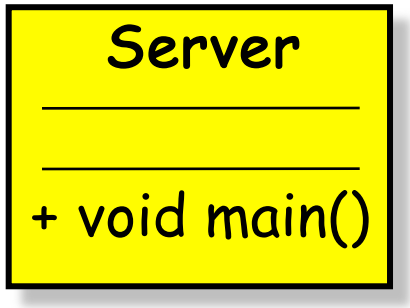


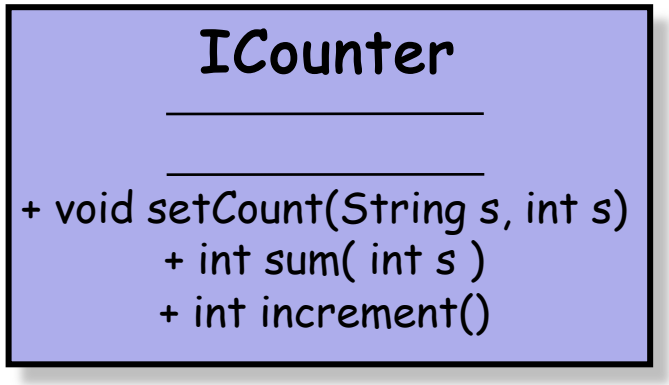


Esempio

Programma client-server che realizza un contatore remoto tramite Proxy-Skeleton e socket UDP.

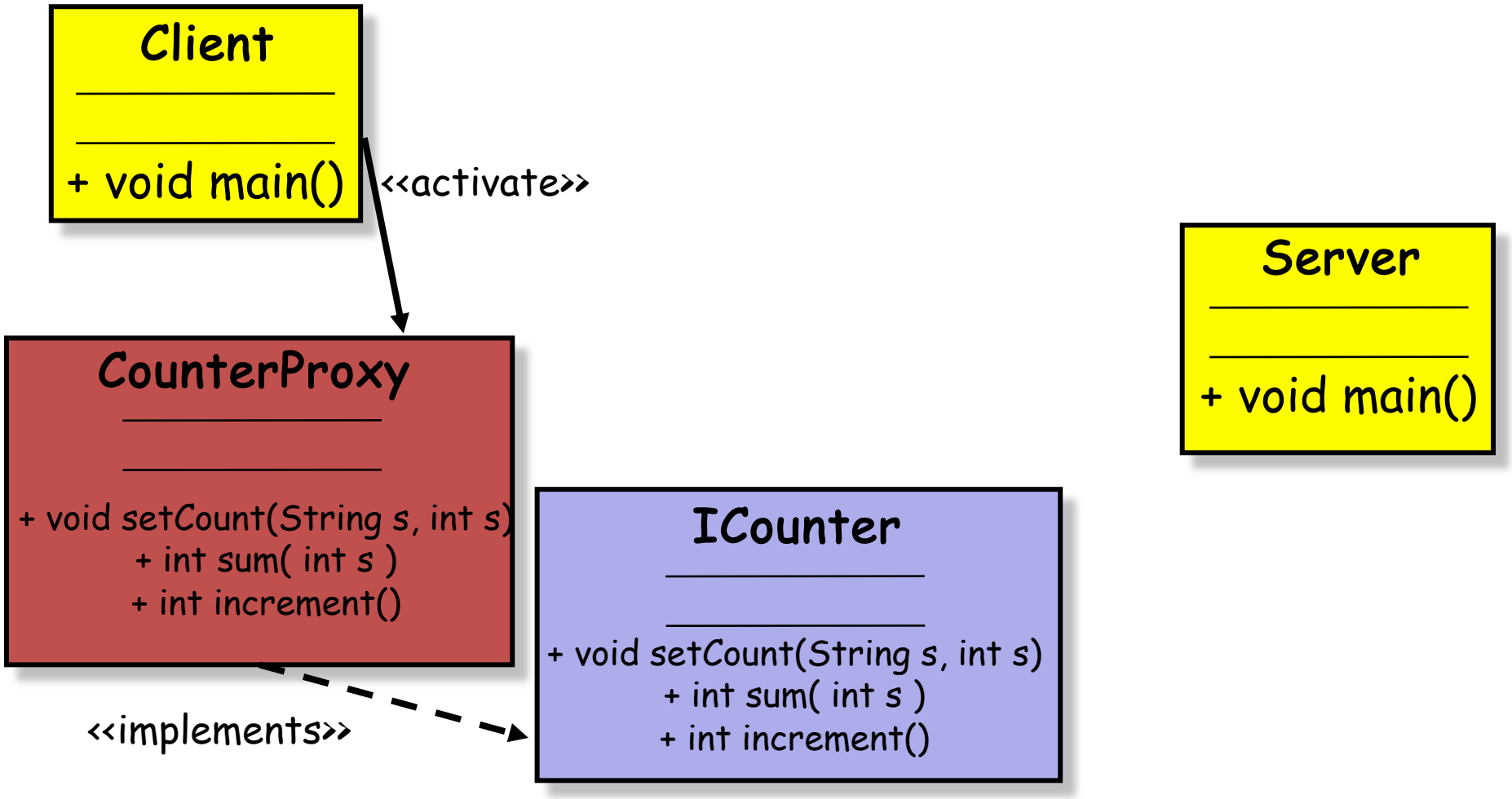






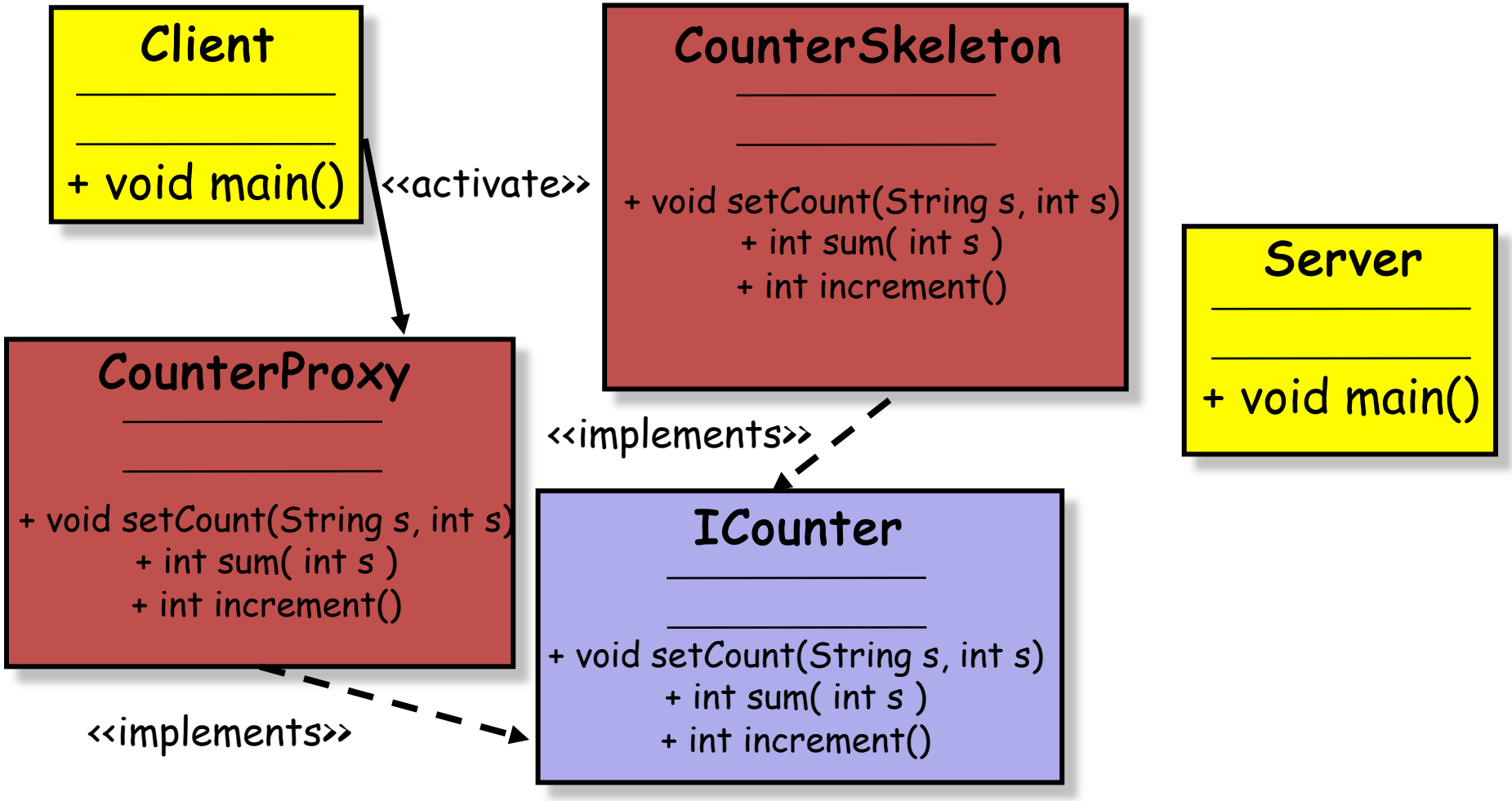
Esempio

Programma client-server che realizza un contatore remoto tramite Proxy-Skeleton e socket UDP.



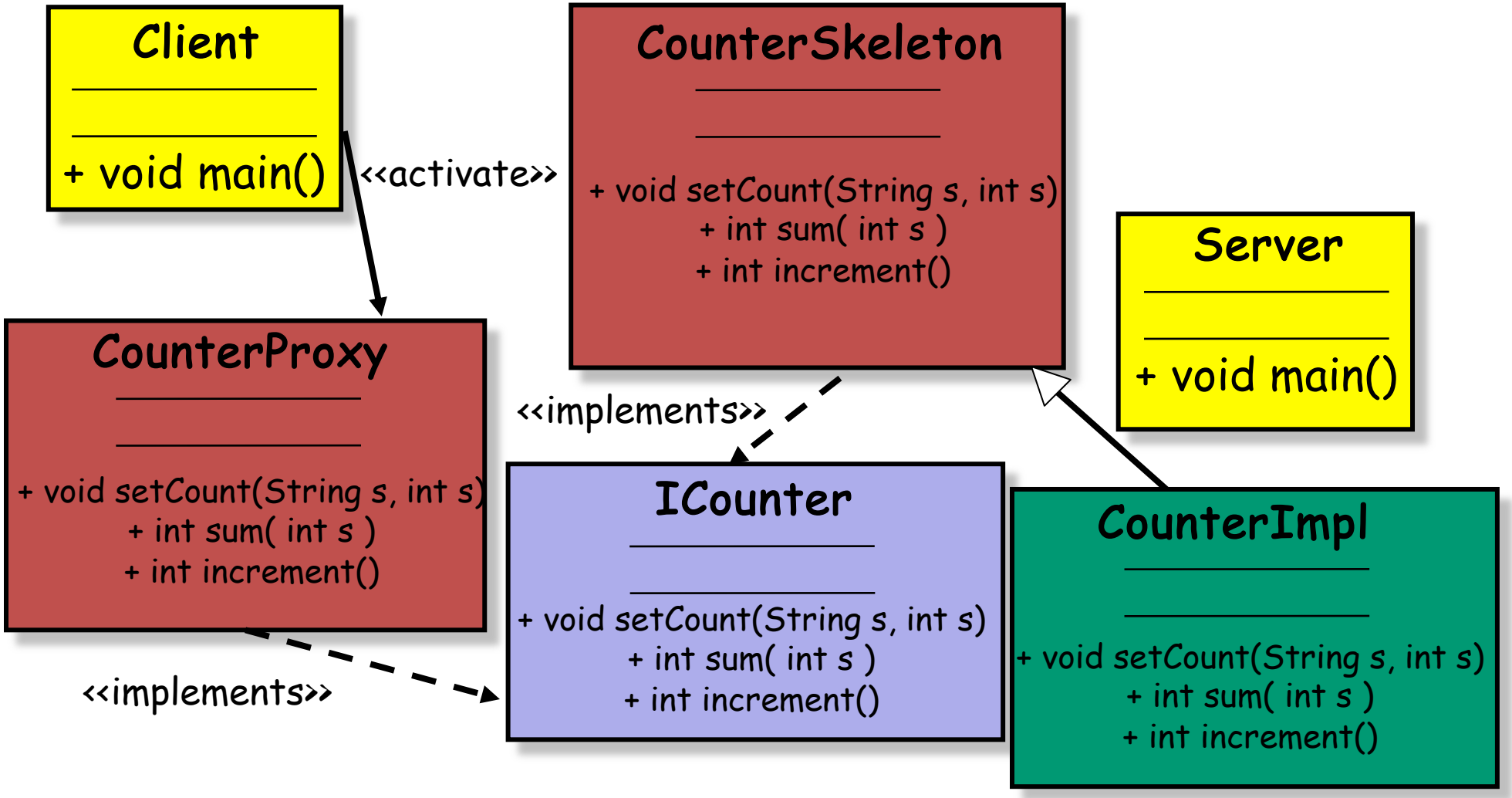
Esempio

Programma client-server che realizza un contatore remoto tramite Proxy-Skeleton e socket UDP.



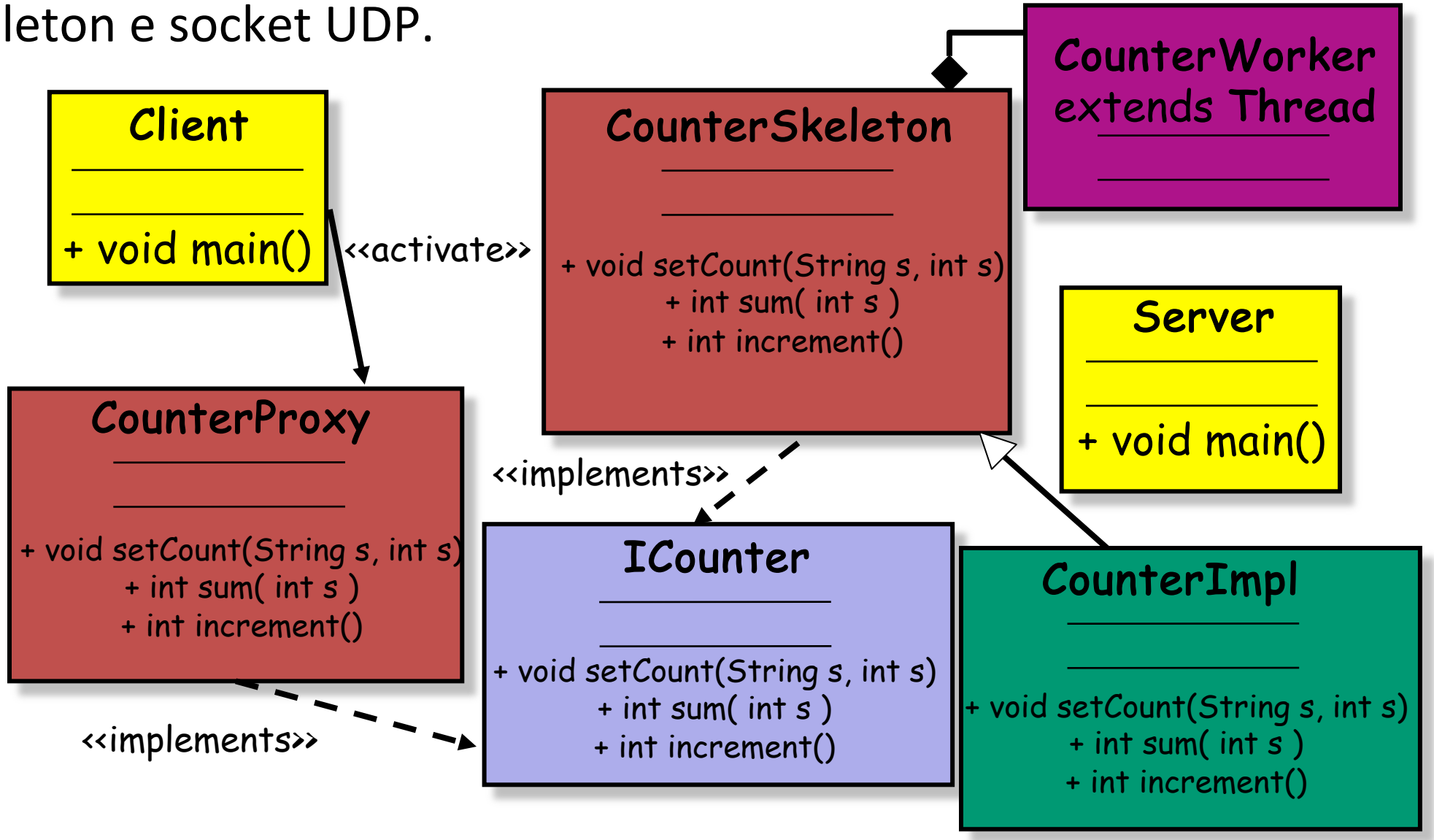
Esempio

Programma client-server che realizza un contatore remoto tramite Proxy-Skeleton e socket UDP.



Esempio

Programma client-server che realizza un contatore remoto tramite Proxy-Skeleton e socket UDP.



Main program

Client

```
public static void main(String[] args) {
      ICounter counter = new CounterProxy ( );

      int x=0;

      System.out.println ( "set count to 10 " );
      counter.setCount("user-ACP", 10 );

      System.out.print ( "sum 15 to count; "  );
      x= counter.sum( 15 );
      System.out.println ( "current count: " +  x );

      System.out.print ( "increment count; " );
      x= counter.increment();
      System.out.println (  "current count: " + x );
}
```

Riferimento al proxy

Server

```
public static void main(String[] args) {
      CounterImpl counter = new CounterImpl ();
      counter.runSkeleton();
}
```

Attivazione oggetto remoto (schema per ereditarietà)

Proxy UDP

```
public class CounterProxy implements ICounter {

private DatagramSocket socket ;

public CounterProxy ( ){
        try{
                socket = new DatagramSocket ();
        }catch ( SocketException e ){
                e.printStackTrace();
        }
}

public void setCount(String id, int s){

        String message = new String ( "setCount#" + id + "#" + s + "#" );

        try{
                DatagramPacket request = new DatagramPacket ( message.getBytes(),
                        message.getBytes().length, InetAddress.getLocalHost(),
                        9000 );

                socket.send( request );

                byte[] buffer = new byte[ 100 ];
                DatagramPacket reply = new  DatagramPacket (buffer,
                         buffer.length );

                socket.receive( reply );

        }catch ( IOException e ){
                e.printStackTrace();
        }
}
```

Proxy UDP

```
public int sum(int s){
        int x=0;

        String message = new String ( "sum#" + s + "#" );

        try{
                DatagramPacket request = new DatagramPacket ( message.getBytes(),
                        message.getBytes().length,  InetAddress.getLocalHost(), 9000 );

                socket.send( request );

                byte[] buffer = new byte[ 100 ];
                DatagramPacket reply = new  DatagramPacket ( buffer, buffer.length );

                socket.receive( reply );

                String replyMessage = new String ( reply.getData(), 0,
                        reply.getLength() );

                x = Integer.valueOf( replyMessage ).intValue();

        }catch ( IOException e ){
                e.printStackTrace();
        }

        return x;
}
```

Proxy UDP

```
public int increment(){
        int x=0;

        String message = new String ( "increment#"  );

        try{
                DatagramPacket request = new DatagramPacket ( message.getBytes(),
                        message.getBytes().length, InetAddress.getLocalHost(), 9000 );

                socket.send( request );

                byte[] buffer = new byte[ 100 ];
                DatagramPacket reply = new  DatagramPacket ( buffer, buffer.length );

                socket.receive( reply );

                String replyMessage = new String ( reply.getData(), 0,
                        reply.getLength() );

                x = Integer.valueOf( replyMessage ).intValue();

        }catch ( IOException e ){
                e.printStackTrace();
        }

        return x;
}
```

Gestione dei pacchetti UDP (1/3)

- NOTA: il payload del messaggio UDP è costituito da un **array di byte**:
  - Necessità di adattare il nome del metodo (ed eventuali parametri dell'invocazione) al pacchetto di richiesta UDP.
- Una possibile soluzione consiste nel creare una String concatenando nome del metodo e singoli parametri, e delimitarli con un carattere separatore (per es. #)
  - `nomeMetodo#argomento1#argomento2#argomentoN#`
  - Esempio:

```
String message = new String ( "setCount#" + id + "#" + s + "#" );
```

Gestione dei pacchetti UDP (2/3)

- La *String* è quindi inserita nel pacchetto di richiesta UDP tramite getBytes()

```
DatagramPacket request = new DatagramPacket ( message.getBytes(),
        message.getBytes().length, InetAddress.getLocalHost(), 9000 );
```

- **Il ricevente (si veda classe CounterWorker):**
  - estrae la *String* dal pacchetto UDP ricevuto;
  - individua nome metodo e singoli parametri tramite la classe StringTokenizer ed il metodo nextToken();
  - quando necessario, converte i parametri ricevuti da String al tipo richiesto tramite il metodo valueOf della classe Wrapper associata al tipo richiesto. Per es. (caso int):

```
int x = Integer.valueOf( messageTokens.nextToken() ).intValue();
```

Gestione dei pacchetti UDP (3/3)

- **StringTokenizer: scompone una String in pezzi (token) delimitati dal carattere separatore scelto:**
  - invocazioni successive del metodo nextToken() della classe StringTokenizer resituiscono i singoli token.

    setCount#user-ACP#10# → setCount user-ACP 10

- **Metodi di utilità delle classi Wrapper (Boolean, Character, Integer, Long..). Esempio Integer:**

  ```
  public static String toString(int i)
  ```

  rappresentazione in formato Stringa dell'argomento int

  Per es.: 24 → "24";

  ```
  static Integer valueOf(String s)
  ```

  restituisce un Integer contenente l'intero rappresentato dalla Stringa;

NOTA: consultare la documentazione Java per maggiori dettagli.

Skeleton UDP (per ereditarietà)

```
public abstract class CounterSkeleton implements ICounter {

    public void runSkeleton  ( ){

        try{
            DatagramSocket socket = new DatagramSocket ( 9000 );

            System.out.println ( "[CounterSkeleton] Entering main loop..." );

            while ( true ){
                byte[] buffer = new byte[100];
                DatagramPacket request = new DatagramPacket ( buffer,
                        buffer.length );

                socket.receive( request );

                //Avvio del thread worker
                CounterWorker worker = new CounterWorker ( socket, request,
                        this );
                worker.start();
            }

        }catch ( IOException e ){
                e.printStackTrace();
        }
    }

}
```

Il buffer va dimensionato in base ai dati scambiati (massima dimensione: 65508)

CounterWorker

```
public class CounterWorker extends Thread {

        private DatagramSocket socket ;
        private DatagramPacket request ;
        private ICounter skeleton ;

        public CounterWorker ( DatagramSocket s, DatagramPacket r, ICounter sk){
                socket = s;
                request = r;
                skeleton = sk;
        }


        public void run ( ){

                String message = new String ( request.getData(), 0,
                      request.getLength() );

                System.out.println ( "\n[CounterWorker] Processing packet:\n" +
                                     "-> request size = " + request.getLength() + "\n"
                                     + "-> message = " + message );


                StringTokenizer messageTokens = new StringTokenizer ( message, "#" );
                String method = messageTokens.nextToken();
```

Estrae la String dal pacchetto UDP

StringTokenizer: il costruttore specifica il carattere separatore (#); nextToken() restituisce il token successivo (NOTA: il primo token è il nome del metodo)

CounterWorker

A seconda del metodo, gli N token successivi sono i parametri dell'invocazione

upcall

Creazione ed invio del messaggio/pacchetto di risposta

```
if ( method.compareTo("setCount") == 0 ){

        String id = messageTokens.nextToken();
        int x = Integer.valueOf( messageTokens.nextToken() ).intValue();

        skeleton.setCount( id,  x );


        String replyMessage = "ack";
        DatagramPacket reply = new DatagramPacket ( replyMessage.getBytes(),
                        replyMessage.getBytes().length, request.getAddress(),
                        request.getPort());

        try{
                socket.send(reply);
        }catch ( IOException e ){
                e.printStackTrace();
        }

}else if ( method.compareTo("sum") == 0 ) {

        int x = Integer.valueOf( messageTokens.nextToken() ).intValue();

        int res = skeleton.sum( x );

        String replyMessage = Integer.toString( res );

        DatagramPacket reply = new DatagramPacket ( replyMessage.getBytes(),
                        replyMessage.getBytes().length, request.getAddress(),
                        request.getPort());

        try{
                socket.send(reply);
        }catch ( IOException e ){
                e.printStackTrace();
        }
}
```

CounterWorker

```
else if ( method.compareTo("increment") == 0 ) {

        int res = skeleton.increment();

        String replyMessage = Integer.toString( res );
        DatagramPacket reply = new DatagramPacket ( replyMessage.getBytes(),
                        replyMessage.getBytes().length, request.getAddress(),
                        request.getPort());

        try{
                socket.send(reply);
        }catch ( IOException e ){
                e.printStackTrace();
        }

}else
        System.out.println ( "* BAD METHOD! *" );


}       // end run
}       // end CounterWorker
```

CounterImpl

```
public class CounterImpl extends CounterSkeleton {
        private int count;

        public void setCount(String id, int s){
                System.out.println ( "    [CounterImpl]: count set to " + s +
                                     " by client " + id + "." );
                count = s;
        }

        public int sum(int s){
                System.out.println ( "    [CounterImpl]: adding " + s + " to
                                    count. " );
                count = count + s;
                return count;
        }

        public int increment(){
                System.out.println ( "    [CounterImpl]: adding 1 to count. " );
                count = count + 1;
                return count;
        }
}
```

- **NOTA: Dove previste, eventuali politiche di sincronizzazione atte a disciplinare l'accesso ai servizi remoti, vanno implementate nella classe *Impl (ossia nell'implementazione reale dei servizi dichiarati nell'interfaccia).**

Esercizi

- Implementare il servizio Contatore remoto tramite socket **UDP**
  - Versione Skeleton per **delega**
- Implementare il servizio Contatore remoto tramite socket **TCP**
  - Versione Skeleton per **ereditarietà**
  - Versione Skeleton per **delega**