Capitolo 14
Canoo ULC e il pattern Half-Object

Il successo delle applicazioni Web è legato alla facilità di deployment e di manutenzione. L'intera applicazione viene infatti installata e gestita su un server centrale, al quale accedono client da ogni parte (*thin clients*), senza che questi ultimi abbiano la necessità di installare software specifico dell'applicazione sulla loro macchina. Il client accede infatti all'applicazione attraverso un browser, un software generico, in grado di interpretare HTML e di interagire con l'applicazione server mediante il protocollo HTTP. Lo stesso browser viene utilizzato dal client per accedere a un numero infinito di applicazioni.

Limiti delle applicazioni Web

Con l'aumentare della popolarità e della diffusione di questo paradigma molte applicazioni sono state adattate con lo scopo di fornire un'interfaccia Web alle funzionalità esistenti. In molti casi, però, ci si è accorti che questo spostamento crea problemi di sviluppo e di manutenzione. Come mai? Il motivo principale è determinato dal fatto che HTML non è pensato per realizzare e gestire interfacce utente (GUI) altamente interattive. Per risolvere i limiti di HTML, gli sviluppatori di applicazioni Web sono così costretti a integrare in HTML numerose tecnologie, come applet Java, JSP, JavaScript, ActionScript, istanze XML proprietarie, e via dicendo,

che creano miscugli di codice e non fanno altro che complicare il lavoro di sviluppo e, soprattutto, di manutenzione.

Una possibile soluzione a questo problema è naturalmente quella di tornare all'idea di *rich client*, utilizzando tecnologie come Swing o SWT, che offrono maggiore interattività, con sistemi di eventi avanzati e widget preconfezionati a disposizione. Eclipse, con la sua piattaforma rich client, sta dando senza dubbio una grossa spinta in questa direzione.

Il prezzo da pagare in questo caso è però la presenza di codice applicativo sul client (*fat client*), con le conseguenze nello sviluppo (separazione client/server) e nel deployment (l'applicazione non risiede più interamente sul server, perciò ogni modifica all'applicazione comporta un aggiornamento del client).

Canoo ULC

La libreria ULC (Ultra Light Client) di Canoo permette di offrire funzionalità di tipo rich client, senza dover ricorrere a fat client. In altre parole abbiamo a disposizione una tecnologia avanzata di widget e gestione di eventi, pur mantenendo l'applicazione interamente sul server. Si parla in questo caso di *rich thin client* (RTC), oppure di *rich internet application* (RIA). Come viene realizzata questa idea?

Il concetto in ULC è relativamente semplice e viene reso possibile dall'utilizzo di un pattern, il cosiddetto pattern Half-Object [Meszaros-1995]. Questo pattern permette a ULC di offrire

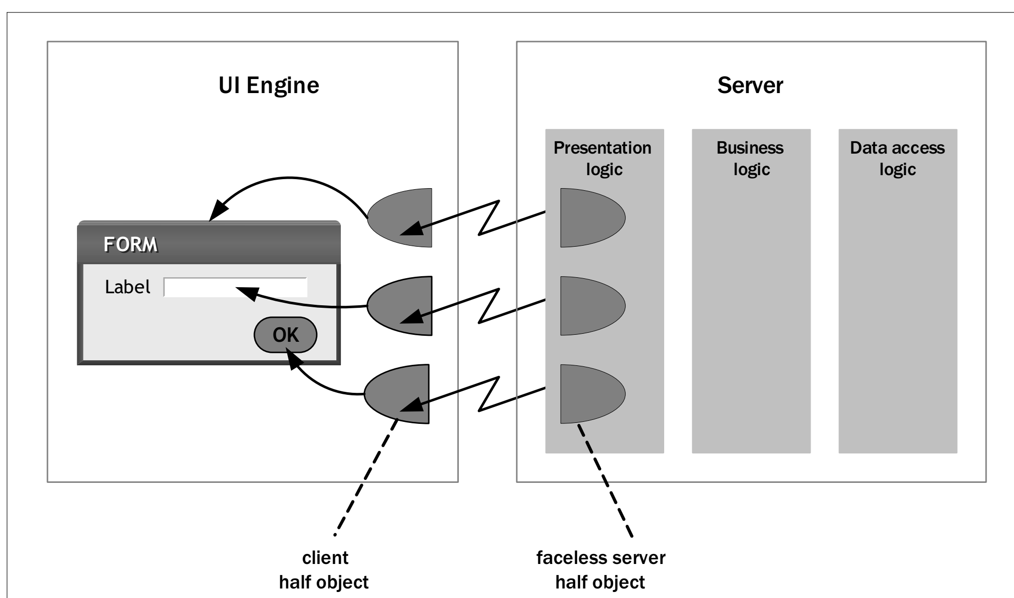

Figura 14.1 – Half-Object per ULC.

una API per Swing a livello server. Chi implementa l'applicazione utilizza questa API direttamente nel server, senza nemmeno specificare la separazione client/server. È come se i widget della GUI venissero utilizzati nel server. L'effettiva separazione tra client e server avviene solo durante l'esecuzione. A runtime, ogni oggetto si suddivide in due: una parte client (*client half object*) e una parte server (*faceless server half object*), che interagiscono tra di loro.
La gestione sul client delle relative "metà di oggetto" avviene tramite il cosiddetto *UI Engine*, un motore generico di interfaccia, utilizzato da tutte le applicazioni, non più grande di 500 kb. La sua funzione è paragonabile a quella del browser per le applicazioni Web.

Il pattern permette di nascondere l'elemento di comunicazione tra client e server, in modo tale che lo sviluppatore possa utilizzare i widget di Swing come se questi si trovassero sul server. L'implementazione del pattern prevede un proxy lato server per ogni componente Swing. Il proxy rappresenta quindi le componenti Swing e viene suddiviso in due metà di oggetti (*half-object*): la metà sul client interagisce effettivamente con Swing, mentre la metà sul server mette a disposizione la API per lo sviluppatore. Considerando l'onnipresenza di Java, con un JRE standard su client e su server, questo approccio permette di lavorare con una libreria relativamente ridotta e un design pulito [Pedrazzini-2004].

La funzionalità richiesta dalla GUI viene realizzata con codice Java object oriented, al posto di miscugli di codice e tecnologie eterogenee. Il vantaggio per lo sviluppatore è evidente: può lavorare con un insieme minimo di utensili e un modello di programmazione semplice. In realtà sviluppa un'applicazione come se questa fosse standalone, diventando client/server unicamente in fase di esecuzione.

Come detto, affinché questo sia possibile il client deve possedere il motore di presentazione (UIEngine). Questo motore, esattamente come un browser per HTML, non contiene codice specifico di un'applicazione e può quindi essere utilizzato con ogni tipo di programma ULC. Siccome lo UIEngine è realizzato completamente in Java, può essere eseguito come applet (quindi senza necessità di installazione), come applicazione standalone, oppure in ambiente JNLP, cioè con il meccanismo di Java Web Start a disposizione del linguaggio Java.

Tecnologie thin client

Il mio rapporto con tecnologie *thin client* sul modello di ULC è iniziato nel 1997. Attraverso la SUPSI, dove insegno e dove a quei tempi mi occupavo di progetti, riuscimmo infatti a farci finanziare un progetto chiamato "generic middle-tier for Web applications", il cui scopo era quello di studiare le modalità per realizzare un framework in grado di gestire in modo generico alcuni aspetti tipici della programmazione server, come sessioni, accesso a un database, e così via. Appena partito il progetto, io e il mio collaboratore Raffaello Giulietti ci rendemmo conto che sarebbe stato più interessante occuparsi dell'aspetto di interazione tra il tier di presentation e il middle tier, piuttosto che di aspetti legati al server, per i quali si iniziavano già a intravedere proposte che avrebbero poi portato agli EJB attuali.
Realizzammo così un prototipo di architettura thin client in Java, spostando la presentation logic sul tier intermedio, prevedendo l'aspetto di comunicazione client/server prevalentemente nelle classi `Listener`, responsabili dell'esecuzione degli eventi generati nella GUI. La trasparenza non era totale, perché alcune istruzioni esplicite dovevano essere inviate dal server al client in fase di setup. Il progetto venne presentato alle giornate JavaDays a Düsseldorf [Giulietti-1999].

In quello stesso periodo iniziava a muovere i primi passi, in altri luoghi e con altri mezzi, l'idea architetturale di ULC, promosso da grossi istituti finanziari con il coinvolgimento in fase di start up della OTI di Erich Gamma. Riprendemmo il progetto con Canoo, nel 2000, dopo che parecchi di coloro che avevano lavorato alle sue prime fasi contribuirono alla fondazione della nuova azienda. Da allora, fu comunque necessario ancora molto lavoro per trasformarlo in prodotto.

Utilizzo di ULC

L'utilizzo delle classi ULC è molto simile all'utilizzo di Swing. In generale basta convertire i nomi delle classi di Swing inizianti con "J" (ad esempio JButton) con identificatori corrispondenti che iniziano con "ULC" (JButton diventa ULCButton). In realtà non sempre è possibile questa conversione, soprattutto perché in ULC è stata data maggiore attenzione all'ortogonalità delle singole API. Inoltre si possono prevedere alcune operazioni particolari, che hanno senso unicamente in un contesto client/server. Ne vediamo alcune nell'esempio che segue.

Confronto con Swing

Per fare un confronto a livello di codice con l'utilizzo di Swing, consideriamo l'esempio di una finestra contenente un pulsante. Ecco la versione Swing della finestra, realizzata con una nuova classe che modelli la finestra stessa ereditando da JFrame.

```
public class JButtonWindow extends JFrame
{
  public JButtonWindow(String title){
    super(title);
    JButton button = new JButton("First Appearance");
    getContentPane().add(button);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pack();
  }
}
```

Questo diventa il codice necessario per istanziare un oggetto e visualizzare la finestra:

```
JFrame frame = new JButtonWindow("JButtonWindow");
frame.setVisible(true);
```

Come detto, per ogni classe Swing che rappresenta un widget, ne esiste una corrispondente ULC. Questo significa che nel nostro caso avremo almeno ULCFrame e ULCButton a disposizione. A livello di import tutto ciò che viene importato in Swing dal package javax.swing, viene sostituito da com.ulcjava.application.

Ecco quindi il codice corrispondente in ULC:

```
public class ULCButtonWindow extends ULCFrame
```

```
{
    public ULCButtonWindow(String title){
        super(title);
        ULCButton button = new ULCButton("First Appearance");
        add(button);
        setDefaultCloseOperation(ULCFrame.TERMINATE_ON_CLOSE);
    }
}
```

Le differenze sono dovute a scelte di semplificazione (add() chiamata direttamente sul frame, invece che sul content pane) o perché, basandosi ULC su Swing, ci sono operazioni che sono implicite (chiamata pack()). Per il resto non ci sono grosse differenze. Importante: il fatto che l'applicazione giri completamente sul server non è visibile nel codice, e nemmeno lo sviluppatore deve preoccuparsene.

Questo è il codice di chiamata

```
ULCFrame frame = new ULCButtonWindow("ULCButtonWindow");
frame.setVisible(true);
```

Aggiunta di eventi

Vediamo un ulteriore aspetto che dimostri quanto la trasparenza riguardo l'architettura client/server sia totale. Aggiungiamo un'azione al pulsante appena creato. Facciamo in modo che il testo del pulsante venga modificato al primo click.

Ecco la versione Swing:

```
class ChangeTextAction implements ActionListener
{
    public void actionPerformed(ActionEvent event){
        JButton source = (JButton)event.getSource();
        source.setText("Changed...");
    }
}
```

Ed ecco la versione ULC:

```
class ChangeTextAction implements IActionListener
{
    public void actionPerformed(ActionEvent event){
        ULCButton source = (ULCButton)event.getSource();
        source.setText("Changed...");
    }
}
```

Se consideriamo che nello scenario di deployment più tipico l'azione viene eseguita sul server, può sorprendere come questo fatto non abbia nessun impatto a livello di codice. L'azione viene eseguita sul server e le modifiche nella metà del widget che riguardano l'interfaccia grafica vengono inviate all'engine grafico del client.

Visto che il deployment di sviluppo è di tipo standalone, il codice sopra dimostra come ULC potrebbe essere facilmente utilizzato al posto di Swing. Solo nel caso in cui, per ragioni di scalabilità, si desiderasse introdurre un'architettura distribuita, si passerebbe a un deployment client/server, senza toccare una sola linea di codice.

Il pattern Half-Object

Il pattern Half-Object è stato presentato per la prima volta da Gerard Meszaros nella pubblicazione *Patterns Languages of Program Design* [Meszaros-1995]. Vediamone gli aspetti essenziali, utili all'architettura di ULC.

Problema

A volte gli oggetti devono comparire in più contesti e spazi di indirizzamento. Come prevedere che la differenza tra uno e più spazi sia trasparente? In altre parole, come rendere trasparente la differenza tra applicazioni singole e applicazioni distribuite?

Motivazione

A causa di vari fattori, che vanno dai costi di esecuzione, alla dimensione, alla necessaria distribuzione fisica, diversità di ambienti, e così via, alcuni sistemi devono utilizzare più spazi di indirizzamento. A volte questi sistemi possono essere suddivisi facilmente in più oggetti, ognuno dei quali può risiedere, sia fisicamente che logicamente, in uno spazio diverso.

Il problema si pone però quando un certo concetto, rappresentato da un oggetto, deve per forza esistere in più spazi di indirizzamento, senza che ci sia la possibilità di scomporlo. Oppure ancora: un oggetto deve interagire con altri oggetti in più spazi di indirizzamento perché necessita di informazioni derivanti da più elementi distribuiti per stabilire il proprio comportamento.

In questi casi non è possibile isolare singole operazioni in un unico spazio. Almeno un oggetto deve esistere in più indirizzamenti contemporaneamente. Deve perciò essere a sua volta suddiviso in uno o più oggetti per spazio di indirizzamento. Questo introduce naturalmente ulteriore complessità, perché evidentemente un oggetto singolo è molto più semplice di due mezzi oggetti con un protocollo di comunicazione tra di loro.

Implementazione

Come separare l'oggetto? Si potrebbe pensare a una metà oggetto che svolge tutti i compiti e una seconda metà che si riferisce alla prima per tutti i compiti da eseguire. Si parlerebbe in

questo caso di pattern Proxy. Se le interazioni tra le due metà sono frequenti, questo porterebbe però a costi di comunicazione troppo elevati.

Un'altra possibilità è quella di mantenere in ogni metà tutte le operazioni che sono necessarie localmente. In questo modo si avrebbe una duplicazione di alcune funzionalità (quelle richieste in entrambi gli spazi di indirizzamento) con la conseguente necessità di sincronizzare le due metà oggetto. È però questa l'implementazione suggerita. In ogni spazio di indirizzamento si realizzano le funzionalità che permettono un'interazione efficiente con gli altri oggetti presenti, con conseguente duplicazione delle funzionalità richieste in entrambi gli spazi. In seguito si definisce il protocollo di comunicazione tra i due oggetti, in modo da coordinare le attività dei due e permettere alle necessarie informazioni di passare da un oggetto all'altro.

Esempi di utilizzo

Oltre al già citato ULC, ecco alcuni esempi di utilizzo del pattern Half-Object (ufficialmente chiamato HOPP, Half Object Plus Protocol):

- Distribuzione fisica: debugger e servizi telefonici distribuiti sulla rete telefonica. Sistemi che accedono a elementi hardware come sensori. Interfacce utenti distanti dall'informazione. Database distribuiti.

- Ambiente di programmazione eterogeneo: interfaccia utente in C++ per debugger implementato in Smalltalk.

Esempio con codice

Proviamo a realizzare un programma per la gestione di richieste FTP che faccia uso del pattern Half-Object per la gestione degli aspetti client/server. Vogliamo partire da una versione standalone, in cui l'intera applicazione è prevista su un'unica macchina, e poi passare alla versione distribuita, in cui però il design rimane pressoché identico. In figura 14.2, ecco come si presenterebbe il design di base della versione standalone.

La classe FTP è quella che mette a disposizione della GUI le funzioni di API per inviare le richieste e poi si occupa di eseguirle interagendo con il filesystem e, più in particolare, con un oggetto di tipo `File`.

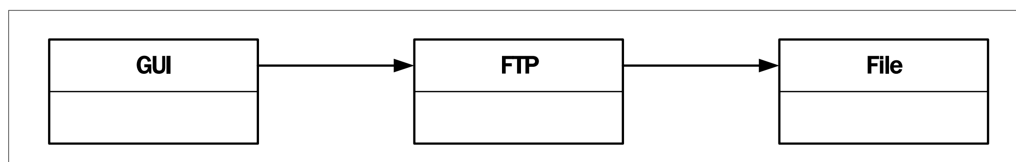

Figura 14.2 – Programma per la gestione di richieste FTP con pattern Half-Object: schema iniziale della versione standalone.

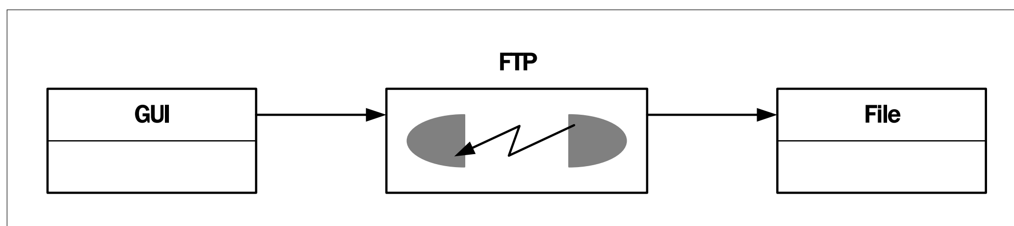

Figura 14.3 – FTP con due metà oggetto.

Architettura client/server

Un programma per FTP ha però senso se le operazioni su file sono permesse da una macchina a un'altra. Se vogliamo mantenere il design logico attuale, questo significa che la classe FTP deve essere divisa in due: una metà nello spazio di indirizzamento del client, per mettere a disposizione le chiamate FTP (dir(), chdir(), get(), exit(), e così via); un'altra metà sul server, per permettere l'effettiva esecuzione delle operazioni. Le due metà di oggetto comunicheranno tra loro attraverso un protocollo privato, non conosciuto nel resto dell'applicazione.

Sincronizzazione

Vogliamo inoltre mantenere uno stato, sia sul client che sul server, che ci permetta di sapere qual è stata l'ultima operazione eseguita con successo, senza per questo dover interagire con l'altra metà oggetto. Lo stato viene modificato sul client, perché è sul client, dopo aver decodificato la risposta del server, che determiniamo se un'operazione è stata effettivamente eseguita in maniera corretta; successivamente il client inizia la sincronizzazione con il server, per fare in modo che anche il server possa mantenere lo stato sull'ultima operazione eseguita.

Lo scopo è quello di permettere a un oggetto che si trova nello stesso spazio di indirizzamento del server di ottenere l'informazione desiderata chiamando getLastOperation(). Stessa cosa deve essere possibile a un oggetto che si trova nello spazio di indirizzamento del client, che, chiamando getLastOperation() non genera nessuna comunicazione verso il server.

Protocollo

Il fatto che il formato del contenuto dei messaggi tra le due metà oggetto sia privato ha il grosso vantaggio di permettere la modifica del protocollo usato per il *tunneling* (contenitore), senza intaccare il resto dell'applicazione. È grazie a questo principio che la libreria di classi ULC descritta in precedenza permette di decidere durante la fase di deployment quale protocollo di tunneling usare (HTTP(S), IIOP, e così via), senza modifiche nel codice dell'applicazione. Ecco, in figura 14.3, come si presenta la versione distribuita dell'applicazione, con la scelta di dividere FTP in due metà oggetto. A livello di implementazione tra le due metà oggetto, dovremo prevedere un proxy (classe ProtocolProxy), che riceve e gestisce le richieste dell'oggetto client e le passa all'oggetto server. Nel nostro caso le richieste partono sempre dal client. La richiesta di cambiamento dello stato di ultima operazione sul client provoca una sincronizzazione con il server, per aggiornare il suo stato (setLastOperation()).

Client

Vediamo inizialmente l'implementazione della classe FTPClient, la metà di oggetto residente nello spazio di indirizzamento del client. Ci limitiamo alle due funzionalità seguenti: chdir() (cambiamento di directory) e dir() (richiesta del contenuto di una directory), con in più la funzionalità getLastOperation().

```
public class FTPClient
{
  private Socket fSocket;
  private BufferedReader fReader;
  private PrintWriter fWriter;
  private String fLastOperation;

  public FTPClient(String server, int port) throws IOException{
    fSocket = new Socket(server,port);
    fReader = new BufferedReader(new InputStreamReader(fSocket.getInputStream()));
    fWriter = new PrintWriter(fSocket.getOutputStream(),true);
  }

  public String chdir(String dir) throws IOException{
    fWriter.println("cd " + dir);
    String response = fReader.readLine();
    if (response.startsWith("OK")){
      setLastOperation("chdir");
      return response.substring(response.indexOf(" "));
    }else{
      return null;
    }
  }

  public String[] dir() throws IOException {
```

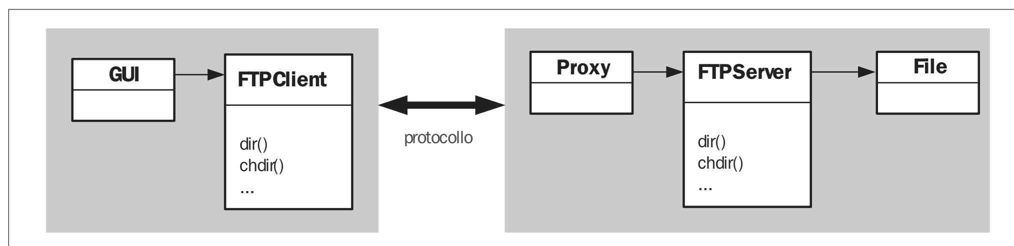

Figura 14.4 – Schema completo dell'applicazione.

```
        fWriter.println("dir");
        fWriter.flush();
        String response = fReader.readLine();
        if (response.startsWith("OK")){
            setLastOperation("dir");
            int quantity = Integer.parseInt(response = fReader.readLine());
            List files = new ArrayList(quantity);
            for (int i=0; i<quantity; i++){
                files.add(response = fReader.readLine());
            }
            return (String[]) files.toArray(new String[files.size()]);
        }
        return null;
    }

    public String getLastOperation(){
        return fLastOperation;
    }

    private void setLastOperation(String lastOperation) {
        fLastOperation = lastOperation;
        fWriter.println("last " + fLastOperation);
    }
}
```

Come si può vedere, un oggetto FTPClient appena creato attiva un socket che gli permette di comunicare con il server. Di questo socket utilizza gli stream di input e output per comunicare con il server. Il client comanda le operazioni. Ogni sua chiamata invia la richiesta alla sua metà server, attraverso un protocollo proprietario.

Le sue funzioni chdir(), dir() e getLastOperation() verranno chiamate dall'utente attraverso un'interfaccia preposta. Come si può vedere, il metodo privato setLastOperation() aggiorna lo stato locale e si sincronizza con il server.

Server

L'implementazione delle funzioni nella metà server viene delegata alla classe FTPServer, che offre le stesse funzionalità di FTPClient. FTPServer si occupa effettivamente di eseguire le istruzioni sul filesystem. Tra le sue funzionalità vi è anche getLastOperation(), che farà capo allo stato residente sul server.

```
public class FTPServer
{
    private File fClientDir = new File(".");
```

```
    private String fLastOperation = null;

    public String chdir(String dir) {
        File newDir = new File(fCurrentDir, dir);
        if (newDir.isDirectory()) {
            fCurrentDir = newDir;
            try {
                return fCurrentDir.getCanonicalPath();
            }catch (IOException e){
                return null;
            }
        }
        return null;
    }

    public String[] dir(){
        return fClientDir.list();
    }

    public String getLastOperation(){
        return fLastOperation;
    }

    protected void setLastOperation(String lastOperation) {
        fLastOperation = lastOperation;
    }
}
```

Abbiamo già visto che, per collegare le due metà oggetto, è necessario avere una classe che faccia da ponte tra le due, gestendo il socket e il protocollo di comunicazione. Questa classe è la ProtocolProxy.

```
public class ProtocolProxy extends Thread
{
    Socket fIncoming;
    FTPServer fFTPServer = new FTPServer();

    public ProtocolProxy(Socket incoming){
        fIncoming = incoming;
    }

    public void run(){
        BufferedReader reader = null;
```

```
    try{
      reader = new BufferedReader(new InputStreamReader(fIncoming.getInputStream()));
      PrintWriter writer = new PrintWriter(fIncoming.getOutputStream(), true);
      while (true) {
        String line = reader.readLine();
          if (line == null){
            break;
          }
          if (line.startsWith("cd")){
            doChangeDir(cutOperation(line), writer);
          }else if (line.startsWith("dir")){
            doDir(writer);
          }else if (line.startsWith("last")) {
            doLastOperation(cutOperation(line));
          }else{
            writer.println("Invalid operation");
          }
        }
      }catch (Exception e){
        e.printStackTrace();
      }finally{
        try {
          reader.close();
        }catch (Exception ignore){}
      }
    }
  }

  private String cutOperation(String operation) throws Exception{
    int start = operation.indexOf(" ");
    if (start == -1){
      throw new Exception("Wrong formatted operation");
    }
    return operation.substring(start).trim();
  }

  private void doChangeDir(String argument, PrintWriter writer){
    String newDir = fFTPServer.chdir(argument);
    if (newDir != null) {
      writer.println("OK: " + newDir);
    }else{
      writer.println("Cannot change into directory: " + argument);
    }
```

```
    }

    private void doDir(PrintWriter writer) {
      String[] fileNames = fFTPServer.dir();
      if (fileNames != null) {
        writer.println("OK");
        writer.println(fileNames.length);
        for (int i = 0; i < fileNames.length; i++) {
          writer.println(fileNames[i]);
        }
      }else{
        writer.println("No Files in this directory");
      }
    }

    private void doLastOperation(String argument) {
      fFTPServer.setLastOperation(argument);
    }
  }
```

La classe è formata da un loop che gestisce le richieste in arrivo dal client e le passa a FTPServer. Ogni richiesta viene gestita in un metodo privato che si occupa di preparare la risposta per FTPClient. Queste sono le istruzioni di esempio per poter far partire il programma sul server, in attesa di connessioni da client.

```
ServerSocket server = new ServerSocket(7007);
while (true){
  try {
    Socket incoming = server.accept();
    new ProtocolProxy(incoming).start();
  }catch(IOException e){
    e.printStackTrace();
  }
}
```

In questo capitolo…

Il pattern Half-Object permette di trattare in modo omogeneo oggetti che devono esistere in più spazi di indirizzamento. Dello stesso oggetto esistono due metà, una sul client e una sul server. Le due metà si mantengono sincronizzate tra di loro, senza che il resto dell'applicazione debba preoccuparsi della loro residenza distribuita.