UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli Studi di Firenze
Dipartimento di Ingegneria dell'Informazione
anno accademico 2024/2025

Relazione ServerTransfer

Autori:
Federico Monetti, Samuel Bruno

N° Matricola:
7080562
7012352

Corso:
Ingegneria del Software

Docente corso:
Enrico Vicario

Indice

1 Introduzione **4**
- 1.1 Obbiettivo . . . 4
- 1.2 Architettura . . . 5
  - 1.2.1 Package Diagram . . . 6
- 1.3 Strumenti e tecnologie utilizzate . . . 6

2 Analisi e Progettazione **7**
- 2.1 Casi d'uso . . . 7
- 2.2 Use Case Templates . . . 8
- 2.3 Class Diagram . . . 10
- 2.4 Mockup . . . 11
- 2.5 Page Navigation Diagram . . . 13
- 2.6 Design Pattern utilizzati . . . 14
  - 2.6.1 Observer . . . 14
  - 2.6.2 Singleton . . . 15
  - 2.6.3 Command + Factory . . . 15

3 Implementazione **16**
- 3.1 Package Client . . . 16
  - 3.1.1 Classe Client . . . 16
- 3.2 Package Server . . . 16
  - 3.2.1 Classe Server . . . 16
  - 3.2.2 Classe User . . . 17
  - 3.2.3 Classe UserAuthenticator . . . 17
  - 3.2.4 Classe ClientHandler . . . 18
  - 3.2.5 Package observers . . . 20
    - Interfaccia Observer . . . 20
    - LoggerObserver . . . 20
    - Interfaccia Observable . . . 20
    - UserActionObservable . . . 21
    - AdminActionObservable . . . 21
    - AbstractObservable . . . 21
  - 3.2.6 Package Commands . . . 22
    - Classe CommandFactory . . . 22
    - Interfaccia Command . . . 23
    - Classe CdCommand . . . 23
    - Classe DownloadCommand . . . 24
    - Classe ListCommand . . . 24
    - Classe UploadCommand . . . 25
    - Classe DeleteCommand . . . 26

4 Testing del codice **27**
4.1 Tecnologie utilizzate e struttura dei test 27
4.2 Codice di Test 27
4.2.1 white-box 27
UserAuthenticatorWhiteBoxTest 28
ServerWhiteBoxTest 29
ObserverWhiteBoxTest 29
4.2.2 black-box 30
AuthBlackBoxTest 30
CommandFactoryBlackBoxTest 31
FileOperationsBlackBoxTest 31
4.3 Risultati dei test 34

Capitolo 1

Introduzione

1.1 Obbiettivo

Questo progetto consiste in un sistema software client-server per la condivisione e il trasferimento di file, sviluppato in linguaggio Java. L'applicazione è concepita per permettere a utenti remoti di accedere a un archivio di file centralizzato: un programma client si connette a un server dedicato, effettua un'autenticazione mediante credenziali e consente il download dei file autorizzati. Il sistema segue un modello simile a un semplice servizio FTP (File Transfer Protocol) personalizzato: il client può infatti collegarsi al server, eseguire il login con username e password, quindi richiedere e scaricare file dal server in base ai permessi assegnati. Gli utenti coinvolti nel sistema sono principalmente gli utenti finali che utilizzano l'applicazione client per accedere ai file condivisi. Ciascun utente deve essere preventivamente registrato (ossia le sue credenziali sono note al server) per poter effettuare il login ed usufruire del servizio. Non è previsto un utilizzo da parte di utenti non autenticati: tutte le funzionalità di download sono disponibili solo dopo il successo del login. Le funzionalità principali offerte dal sistema possono essere riassunte come segue:

- **Autenticazione degli utenti** : All'avvio il client richiede le credenziali all'utente e le invia al server, il quale verifica username e password confrontandoli con quelli memorizzati nell'archivio degli utenti autorizzati. Solo in caso di credenziali valide l'utente viene ammesso e può interagire con il sistema (in caso contrario il server nega l'accesso).

- **Visualizzazione dei file disponibili** : Dopo il login, il client può richiedere al server la lista dei file (o delle directory) a cui l'utente ha accesso. Il server risponde inviando l'elenco dei contenuti disponibili, permettendo così all'utente di sapere quali risorse può scaricare.

- **Download di file** : L'utente può selezionare, tramite un comando, uno dei file disponibili per scaricarlo. Il client invia quindi una richiesta di trasferimento al server, il quale trasmette il contenuto del file verso il client. Il trasferimento avviene in modo affidabile tramite protocollo TCP/IP, assicurando che il file giunga integro a destinazione

- **Supporto multi-utente concorrente** : Il server è in grado di gestire più client contemporaneamente. Utenti diversi possono connettersi da postazioni diverse e usufruire del servizio in parallelo, senza interferire gli uni con gli altri. Ciò significa, ad esempio, che più utenti possono effettuare il download di file differenti allo stesso tempo, con il server che serve ciascuno attraverso connessioni dedicate.

- **Chiusura della connessione** : In qualsiasi momento l'utente può decidere di terminare la sessione. È prevista la funzionalità di logout/uscita sicura: il client invia

un segnale di chiusura e il server rilascia le risorse allocate per quel client, terminando il thread di servizio dedicato. In questo modo la connessione viene chiusa correttamente da entrambe le parti senza lasciare risorse impegnate.

Nel programma sono presenti due tipologie di utente:

- **User**: si può registrare e/o accedere per poi eseguire le operazioni di navigazione tra le cartelle, o il download di file.
- **Admin**: oltre a possedere tutte le operazioni dell'User, può anche aggiungere o rimuovere i file presenti all'interno del server

1.2 Architettura

L'applicazione adotta un'architettura multi-threaded client-server a due livelli:

- **Server**: un singleton che rimane in ascolto sulla porta TCP predefinita (12345), gestisce autenticazione, autorizzazione, gestione dei file e logging degli eventi;
- **Client**: un programma console che si connette al server, effettua il login e invia comandi testuali per navigare directory, elencare o scaricare file.

Il client interagisce con il server centralizzato per accedere a file condivisi. Il server resta in esecuzione su una macchina host, in ascolto su una porta predefinita (12345), mentre uno o più client remoti possono connettersi per effettuare l'autenticazione e utilizzare i servizi offerti. La comunicazione avviene su socket TCP mediante un protocollo testuale. Ad ogni nuova connessione il server crea un thread `ClientHandler` dedicato, abilitando il servizio concorrente di più client.

Ogni `ClientHandler`:

- gestisce autenticazione e registrazione attraverso `UserAuthenticator`, che utilizza un file di testo (`credentials.txt`) per salvare credenziali e ruoli;
- instrada i comandi (*list*, $cd < dir >$, $download < file >$, $upload < file >$, $delete < file >$, *exit*) tramite il pattern *Command*, implementato da una Factory dedicata;
- notifica gli observer (es. `LoggerObserver`) sugli eventi di download, secondo il pattern *Observer*;
- accede e modifica i file condivisi nella cartella `server_files`.

Il trasferimento dei file è affidabile: il server legge i byte del file, li codifica in Base64 e li invia come stringa al client; il client, tramite un listener asincrono, decodifica il payload e salva il file nella cartella locale.

Questa architettura, priva di database relazionali o GUI complesse, è stata progettata per garantire semplicità, chiarezza e facilità di estensione.

1.2.1 Package Diagram

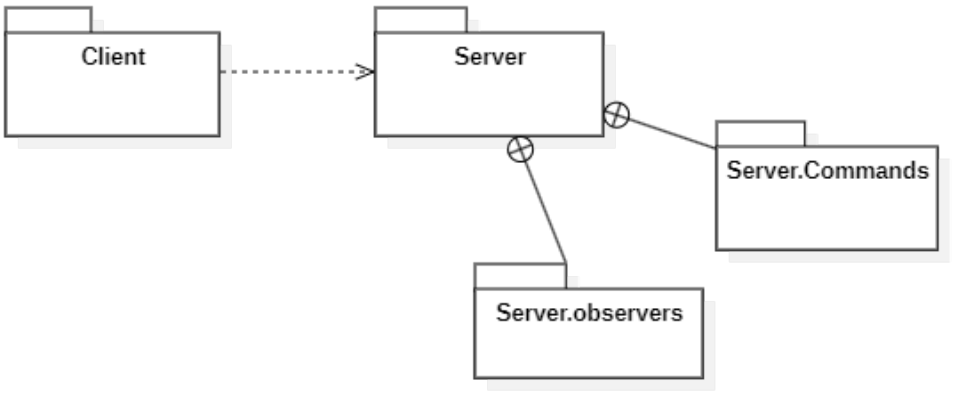

Figura 1.1: Package Diagram

Di seguito una breve descrizione delle responsabilità di ciascun package del progetto:

Client : Contiene la classe principale Client.java che gestisce l'interfaccia testuale, la connessione al server, il thread listener asincrono per le risposte e il salvataggio dei file scaricati in locale.

Server : Racchiude il cuore dell'applicazione server-side, incluse le classi Server.java (singleton), ClientHandler.java (thread per client), UserAuthenticator e User per la gestione dell'autenticazione.

Server.Commands : Implementa il pattern Command: contiene la CommandFactory e tutte le classi comando (ListCommand, DownloadCommand, UploadCommand, DeleteCommand, ecc.), ognuna responsabile dell'elaborazione di un'operazione specifica.

Server.Observers : Definisce il pattern Observer/Observable: include AbstractObservable, UserActionObservable, AdminActionObservable e le relative implementazioni di observer come LoggerObserver, per il logging centralizzato degli eventi di sistema.

1.3 Strumenti e tecnologie utilizzate

Il progetto è stato sviluppato in **Java** utilizzando l'IDE **IntelliJ IDEA**. I diagrammi UML sono stati realizzati con **StarUML** e i mockup delle interfacce (dove presenti) con **Figma**. La stesura della relazione e dei documenti è avvenuta in LaTeX su **Overleaf**.

Capitolo 2

Analisi e Progettazione

Durante la fase di progettazione è stato realizzato un **diagramma UML** delle classi per visualizzare la struttura del sistema e le interazioni tra i vari componenti software. Inoltre, sono stati elaborati un **Use Case Diagram** e alcuni **Use Case Templates** per rappresentare il funzionamento di alcune delle principali operazioni utilizzabili nel programma.

2.1 Casi d'uso

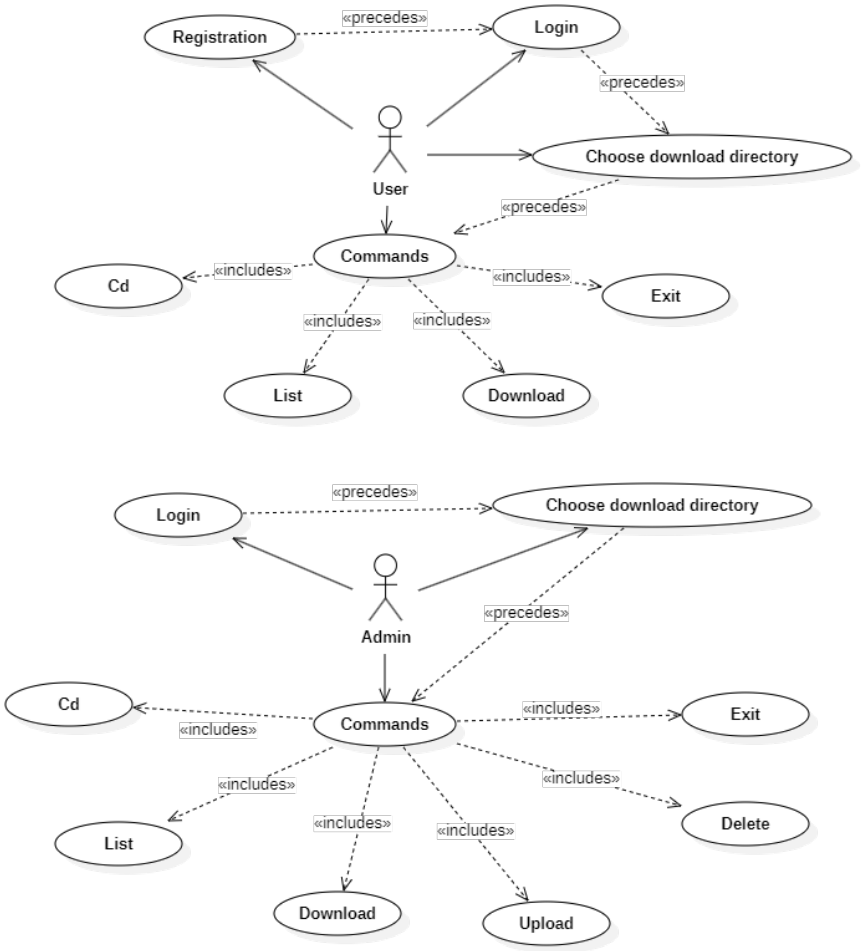

Figura 2.1: Use Case Diagram

2.2 Use Case Templates

In seguito sono mostrati degli Use Case Template che mostrano il flusso di esecuzione di alcuni casi d'uso rappresentati nella Figura 2.1. Sono stati scelti casi considerati più importanti o rappresentativi.

Use Case 1	**Registrazione**
Level	User goal
Description	Un utente effettua la registrazione
Actor	User
Pre-condition	L'utente deve essere nella pagina principale
Normal Flow	1. L'utente inserisce username 2. Inserisce la password 3. Il Server verifica se l'username è già presente 4. Il Server aggiunge le credenziali alla lista delle credenziali 5. Il Server notifica l'avvenuta registrazione
Alternative Flow	1. L'username è già presente, viene notificato un messaggio di errore 2. si ritenta la registrazione
Post-Conditions	L'utente è ora registrato e ha accesso a tutte le sue funizioni

Tabella 2.1: Use Case Template 1

Use Case 2	**Download di un file**
Level	User goal
Description	Un utente fa il download di un File
Actor	User
Pre-condition	L'utente è autenticato (ha completato login o registrazione con successo)
Normal Flow	1. L'utente sceglie la cartella dove scaricare i file 2. L'utente sceglie la cartella all'interno del server dove è presente il file che vuole scaricare 3. esegue il comando di download, specificando il file da voler scaricare 4. il Server cerca il file e lo invia all'Utente 5. Il client salva il file nella directory scelta e viene mostrato un messaggio di conferma
Alternative Flow	1. Il file specificato non esiste tra i file disponibili per il download 2. L'utente viene notificato con un messaggio di errore 3. si ritorna al menu principale
Post-Conditions	L'utente possiede adesso il file scaricato nella cartella scelta

Tabella 2.2: Use Case Template 2

Use Case 3	**Upload di un file**
Level	User goal
Description	Un Admin carica un File sul Server
Actor	Admin
Pre-condition	L'Admin deve aver eseguito il login
Normal Flow	1. L'Admin effettua il comando di upload, specificando il file da voler caricare 2. il Server cerca se il file è già presente 3. Il Server salva il file nella directory dei file scaricabili 4. Viene notificato il caricamento completato
Alternative Flow	1. Il file specificato esiste già tra i file disponibili per il download 2. L'Admin viene notificato con un messaggio di errore 3. si ritorna al menu principale
Post-Conditions	Il Server possiede adesso il nuovo file caricato dall'Admin

Tabella 2.3: Use Case Template 3

2.3 Class Diagram

Il diagramma delle classi riportato successivamente rappresenta le classi implementate e come queste interagiscono tra loro.

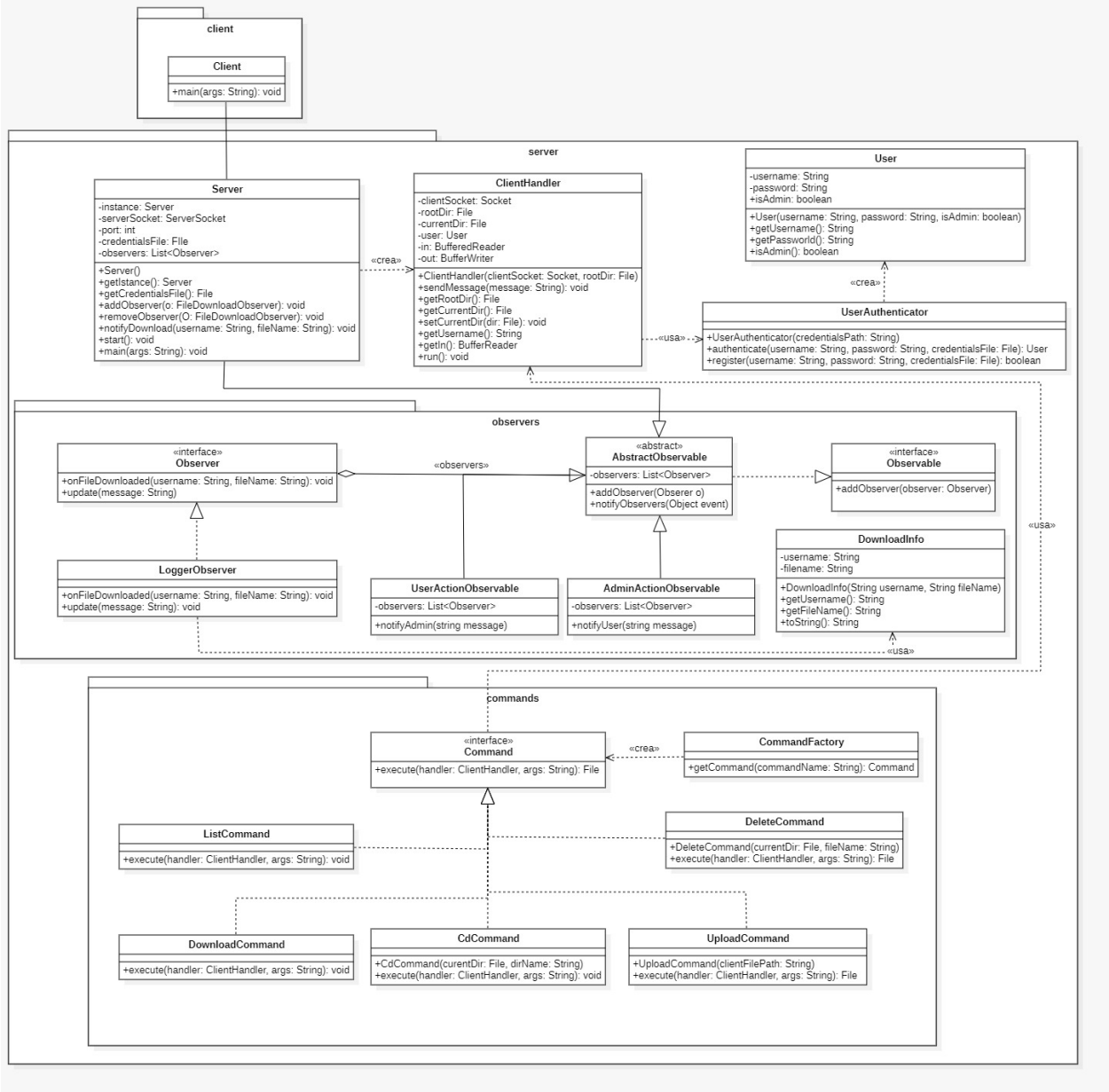

Figura 2.2: Diagramma UML delle classi

L'implementazione delle varie classi è spiegata e mostrata nel Capitolo 3

2.4 Mockup

Per facilitare la comprensione del flusso di utilizzo dell'applicazione, sono stati realizzati alcuni mockup delle schermate principali, rappresentanti un'ipotetica GUI per questo programma. In questa sezione vengono presentati i layout più significativi:

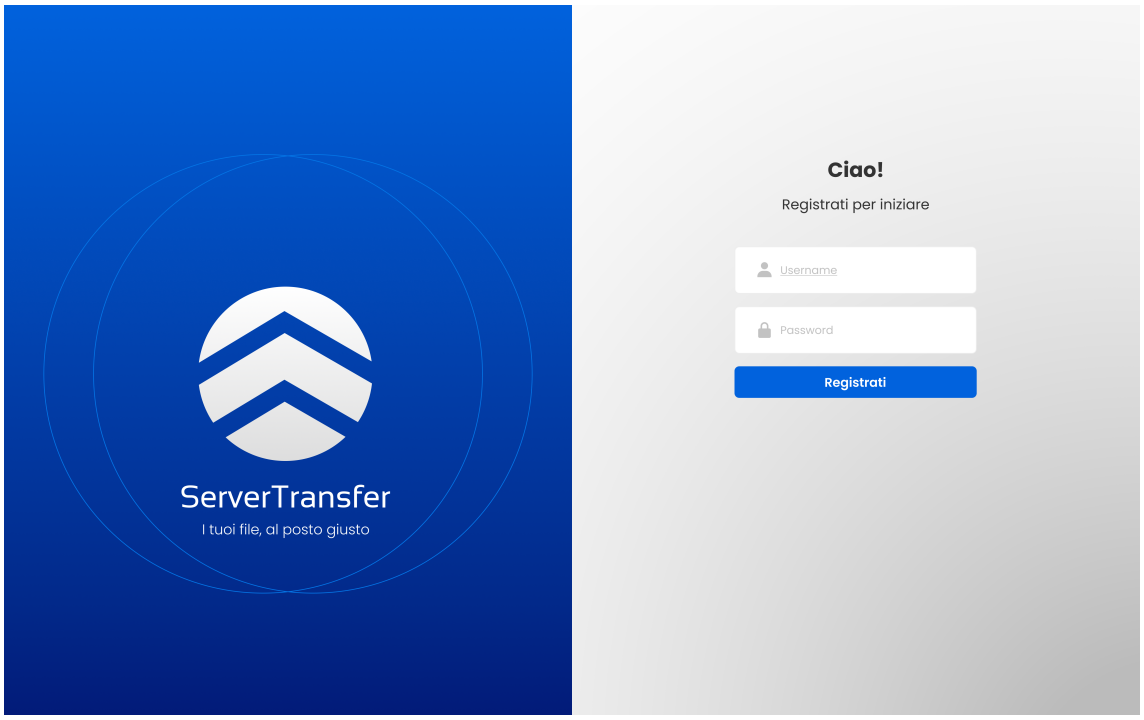

Figura 2.3: Mockup della schermata di registrazione

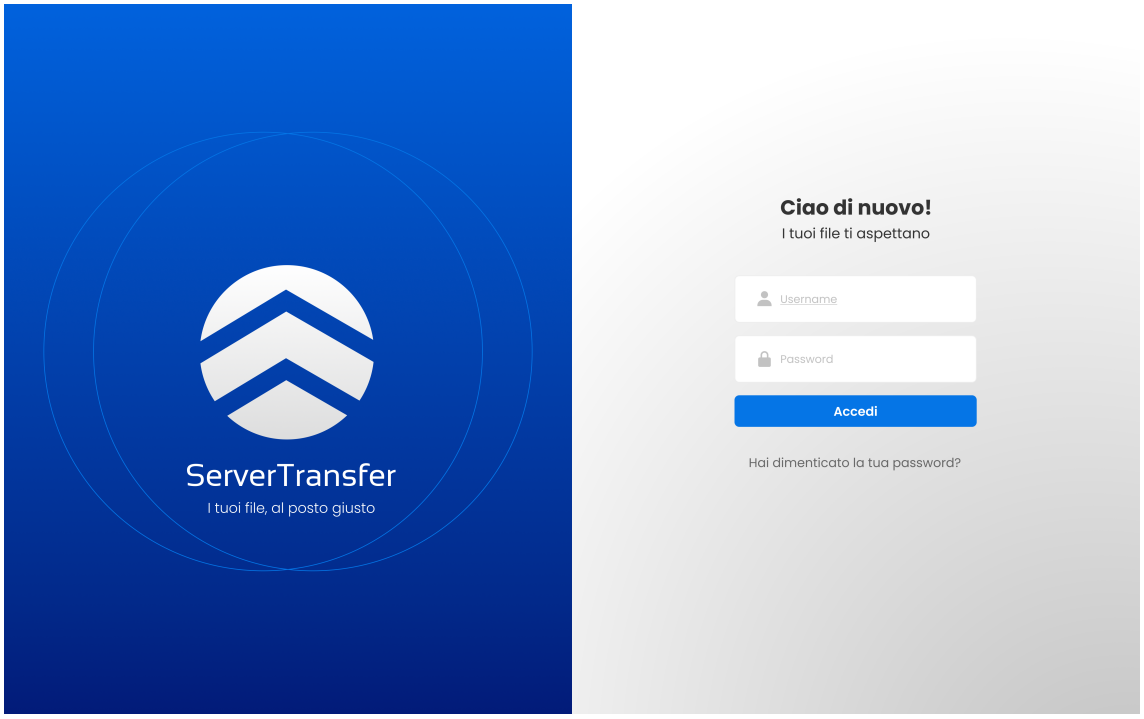

Figura 2.4: Mockup della schermata di accesso

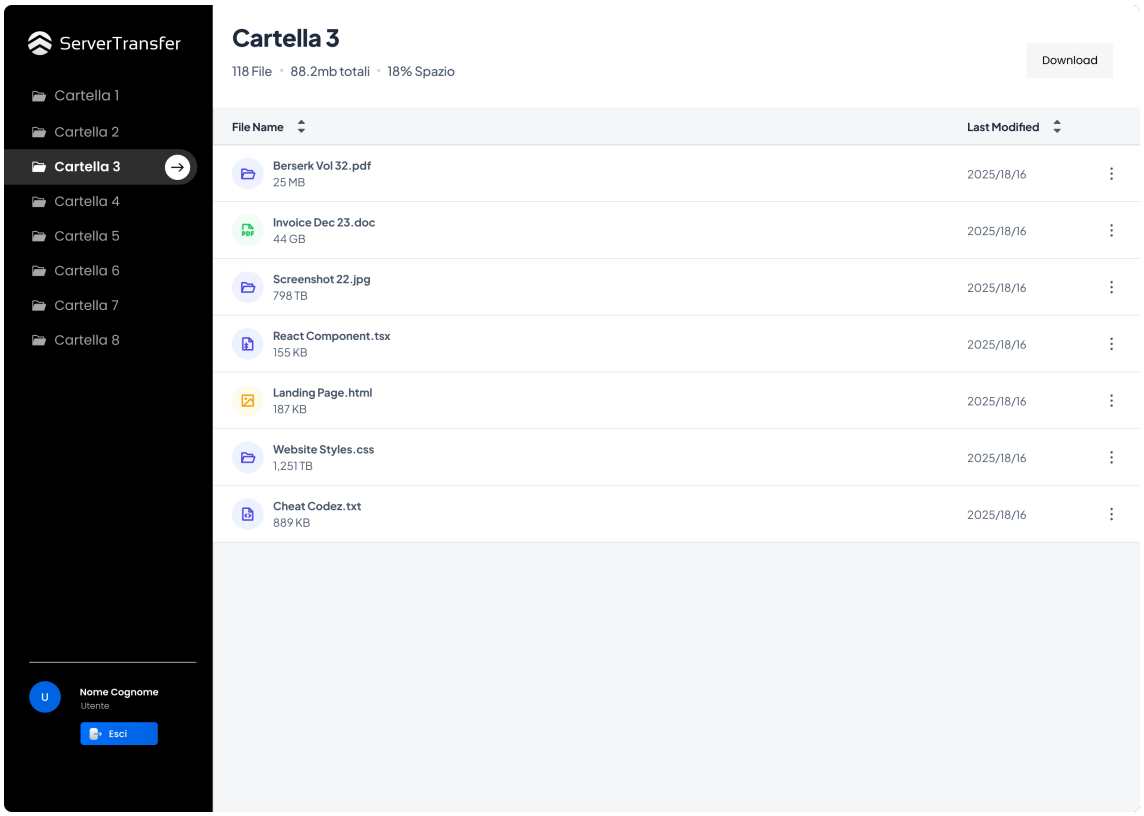

Figura 2.5: Mockup della Dashboard dell'User

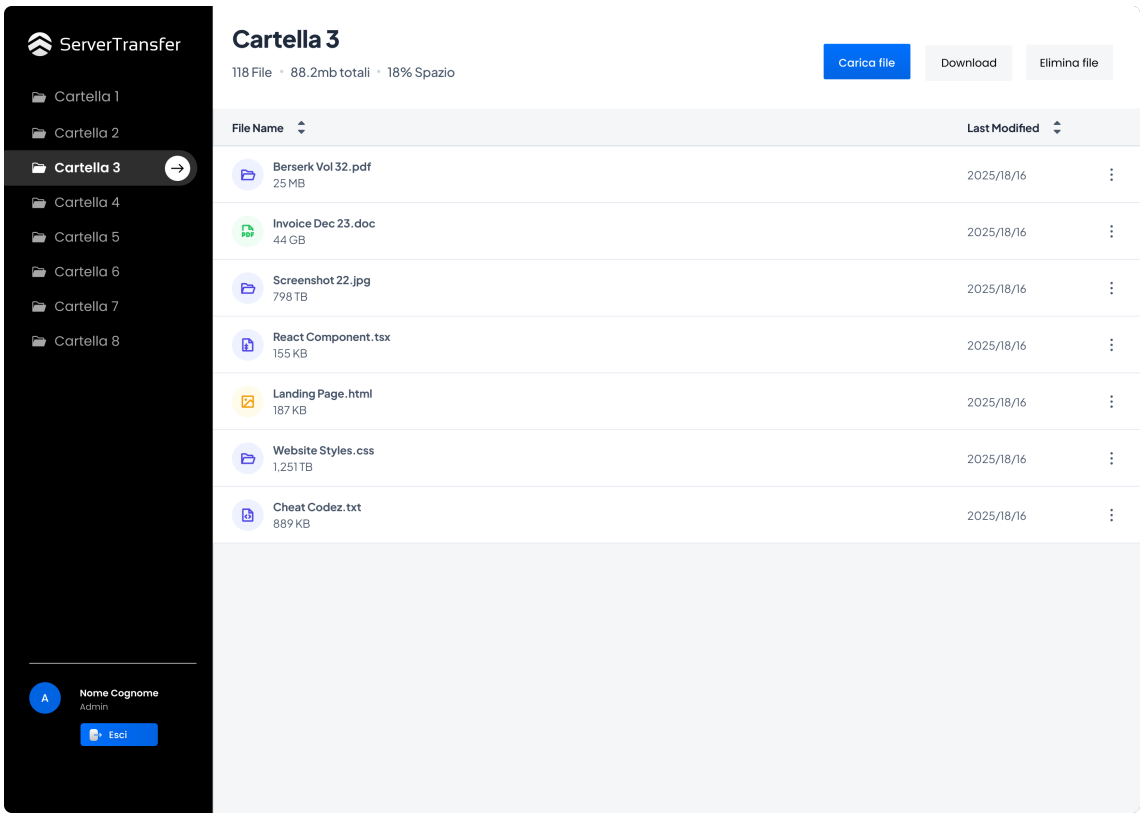

Figura 2.6: Mockup della Dashboard dell'Admin

2.5 Page Navigation Diagram

Immaginando di rendere disponibile un'interfaccia grafica completa all'utente abbiamo rappresentato la logica di navigazione tra alcune pagine del sistema. Si ha un Diagramma per l'User e uno per l'Admin.

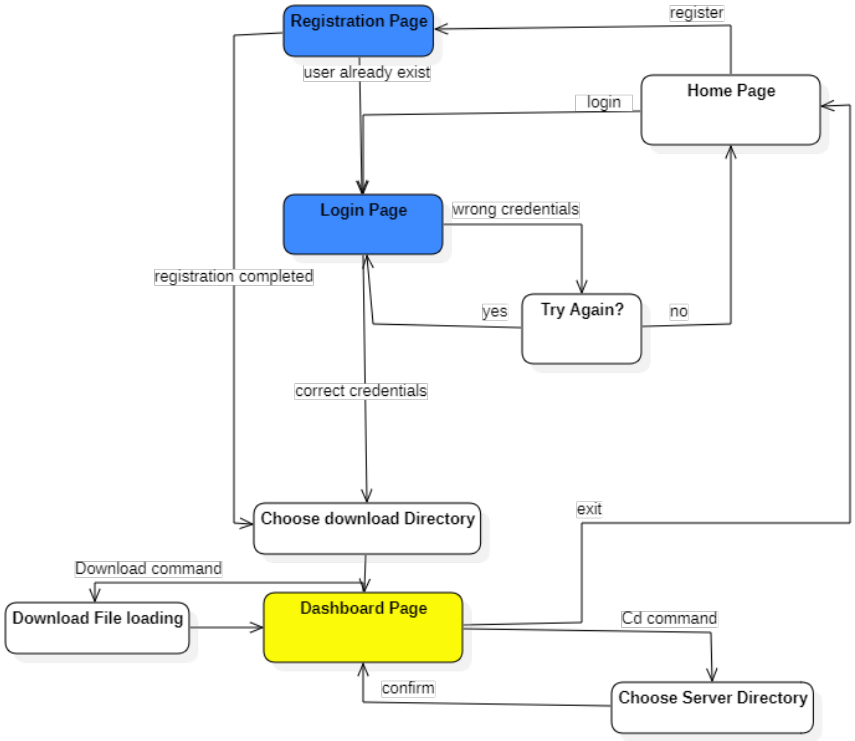

Figura 2.7: User Page Navigation Diagram

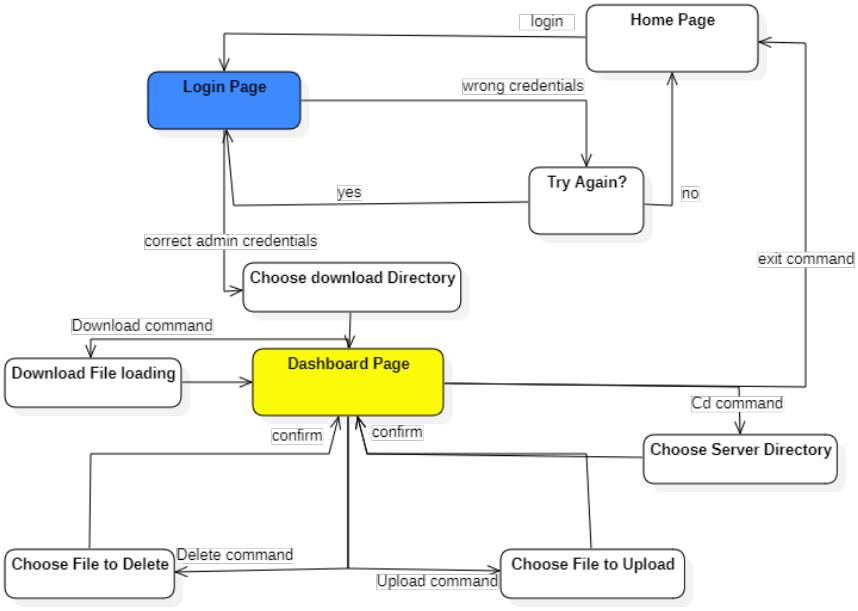

Figura 2.8: Admin Page Navigation Diagram

Le pagine colorate nelle immagini chiamate Registration Page, Login Page, Dashboard Page sono mostrate nei mockups in Sezione 2.4.

2.6 Design Pattern utilizzati

Nel corso dell'implementazione sono stati applicati alcuni design pattern noti, al fine di rendere il codice più modulare, riusabile e manutenibile. In particolare, si sono impiegati i seguenti pattern architetturali: Singleton, Command + Factory e Observer. Di seguito se ne descrive l'utilizzo e le motivazioni.

2.6.1 Observer

Nel progetto il pattern *Observer* viene utilizzato per separare la logica delle operazioni dalle azioni di *logging* e monitoraggio. Si distinguono due "famiglie" di Observable:

AbstractObservable Classe astratta che realizza concretamente il contratto definito da `Observable`, occupandosi della gestione interna degli observer:

- Mantiene una lista privata di `Observer` a cui notificare gli eventi.
- Espone in pubblico il metodo

  `void addObserver(Observer o)`

  per consentire la registrazione di nuovi listener.
- Definisce in modo **protetto** il metodo

  `protected void notifyObservers(Object event)`

  invocabile solo dalle sue sottoclassi o dalle classi del medesimo package, che:
  - Se `event instanceof DownloadInfo`, chiama `onFileDownloaded(event)` su ciascun observer.
  - Altrimenti, invoca `update(event)` passando l'oggetto evento così com'è.
- Spostando qui la logica di notifica (e rimuovendola dall'interfaccia), si ottiene un incapsulamento completo: l'interfaccia `Observable` rimane unicamente un contratto di registrazione, mentre la classe astratta controlla chi può effettivamente emettere gli eventi.

AdminActionObservable e UserActionObservable Classi concrete che estendono `AbstractObservabl` e forniscono metodi pubblici specifici per ciascun dominio di appartenenza, ovvero il dominio User e il dominio Admin:

- Non mantengono liste proprie: ereditano la gestione degli `Observer` da `AbstractObservable`.
- Espongono in pubblico un metodo di dominio:
  - `void notifyAdmin(String message)` in `AdminActionObservable`
  - `void notifyUser(String message)` in `UserActionObservable`

  che internamente invocano il `protected notifyObservers(Object)` ereditato.
- Il metodo pubblico di notifica riceve sempre un singolo parametro `String`, che viene propagato come evento generico a tutti gli observer registrati, tramite la chiamata a `Observer.update(Object)`.

In entrambi i casi:

- Gli `Observer` (ad es. `LoggerObserver`) si registrano chiamando `addObserver(...)` sull'istanza specifica.

- Quando si verifica l'evento amministrativo o utente, viene chiamato il metodo pubblico (`notifyAdmin` o `notifyUser`) che invia il messaggio di evento a tutti gli observer tramite `notifyObservers`.

2.6.2 Singleton

Il pattern Singleton è stato adottato per la classe Server, che rappresenta l'istanza centrale del server in ascolto. È fondamentale che esista una sola istanza globale del server durante l'esecuzione, poiché questa gestisce:

- La lista di observer registrati.
- Il socket di ascolto.
- L'accesso al file delle credenziali.

L'implementazione prevede:

- Costruttore privato.
- Attributo statico privato instance.
- Metodo `getInstance()` che crea e restituisce l'unica istanza.

Questo garantisce un punto di accesso globale al server e impedisce la creazione di istanze multiple, che causerebbero conflitti nella gestione delle connessioni e risorse condivise.

2.6.3 Command + Factory

Il pattern Command è stato utilizzato per incapsulare ogni comando del client in un oggetto dedicato che ne gestisce l'esecuzione. Tutti i comandi implementano l'interfaccia Command, che espone il metodo `execute(ClientHandler handler)`.

I comandi implementati sono:

- `ListCommand`: elenca i file nella directory corrente.
- `CdCommand`: consente la navigazione tra directory.
- `DownloadCommand`: gestisce il download di un file.
- `UploadCommand`, `DeleteCommand` (per gli admin): gestisce la rimozione o l'aggiunta di file dal server

Per creare dinamicamente l'oggetto comando corretto in base al testo inviato dal client, viene usato il pattern Factory:

- La classe CommandFactory espone il metodo statico `getCommand(String name)`, che restituisce l'istanza del comando associato alla stringa ricevuta ("list", "cd", "download"...).

Questo approccio elimina la necessità di lunghi blocchi if/else o switch, semplifica l'aggiunta di nuovi comandi e mantiene separata la logica di parsing da quella esecutiva.

Capitolo 3

Implementazione

Nel diagramma delle classi sono stati identificati due package principali, client e server.. Di seguito sono descritte le principali classi e le loro responsabilità:

3.1 Package Client

3.1.1 Classe Client

Classe principale dell'applicazione client, contenente il main. All'avvio acquisisce dall'utente l'indirizzo del server (se non prestabilito) e tenta la connessione. Una volta connesso, gestisce l'interazione con l'utente locale: ad esempio, chiede di inserire username e password, li invia al server e attende la risposta. Se il login ha successo, presenta le operazioni disponibili (elencare file, scaricarne uno o uscire) e quindi invia i comandi appropriati al server, ricevendo e trattando le risposte. In caso di download, il Client riceve dallo stream di rete i dati del file e li salva su disco locale (eventualmente mostrando all'utente l'avanzamento). In pratica, questa classe coordina la UI e la comunicazione di rete.

3.2 Package Server

3.2.1 Classe Server

Classe principale del lato server. Contiene il metodo main che avvia il server sulla porta specificata e inizializza le risorse necessarie (es. caricamento delle credenziali autorizzate in memoria). Implementa il pattern Singleton per garantire che esista una sola istanza globale del server durante l'esecuzione: questo è fondamentale per gestire correttamente le connessioni, gli observer registrati e l'accesso centralizzato a risorse condivise. Il server apre un ServerSocket e rimane in ascolto di connessioni entranti. Per ogni nuova connessione accettata, istanzia un oggetto ClientHandler dedicato, che gestisce la comunicazione con il relativo client in un thread separato. Questo approccio consente una gestione concorrente di più client.

```
1 public void start() {
2     try {
3         ServerSocket serverSocket = new ServerSocket(port);
4         System.out.println("Server in ascolto sulla porta " +
              port);
5         while (true) {
```

```
6               Socket clientSocket = serverSocket.accept();
7               new Thread(new ClientHandler(clientSocket, rootDir)
                    ).start();
8           }
9       } catch (IOException e) {
10              System.out.println("Errore nel server: " + e.
                    getMessage());
11          } finally {
12              if (serverSocket != null && !serverSocket.isClosed
                    ()) {
13                  try {
14                      serverSocket.close();
15                  } catch (IOException e) { }
16              }
17          }
18      }
```

Listing 3.1: Metodo start() del server

3.2.2 Classe User

La classe User rappresenta un'entità utente all'interno del sistema, contenente le informazioni essenziali per l'autenticazione e la gestione dei privilegi. È utilizzata dal server per identificare l'utente connesso e verificare i permessi di accesso ai comandi.

Utilizzo nel sistema:

- Gli oggetti User vengono creati in fase di login, dopo aver letto le credenziali da un file o da una struttura dati interna.
- L'attributo isAdmin consente di abilitare o disabilitare comandi sensibili come upload o delete, accessibili solo agli amministratori.

3.2.3 Classe UserAuthenticator

Componente è responsabile della gestione delle credenziali utente lato server. Fornisce funzionalità sia per l'autenticazione di utenti esistenti che per la registrazione di nuovi, operando su un file di testo che funge da archivio persistente.

Espone i metodi:

- `authenticate(username, password)` che verifica se le credenziali fornite corrispondono a un utente registrato.
- `register(username, password, isAdmin)` che registra un nuovo utente verificando che l'username non esista già.

Responsabilità:

- Carica all'avvio del server la lista degli utenti autorizzati leggendo da un file di configurazione (ad esempio un file di testo con username, password e ruolo).
- Memorizza i dati utente in un file di testo per consentire controlli rapidi e sicuri. Ogni riga del file contiene $< username > < password > < ruolo >$
- Incapsula completamente la logica di accesso alle credenziali, permettendo al resto del sistema (es. ClientHandler) di autenticare utenti senza conoscere i dettagli dell'archiviazione.

```
public User authenticate(String username, String password) {
    try (BufferedReader reader = new BufferedReader(new
        FileReader(credentialsPath))) {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.trim().split("\\s+");
            if (parts.length >= 3 &&
                parts[0].trim().equals(username.trim()) &&
                parts[1].trim().equals(password.trim())) {
                boolean isAdmin = parts[2].trim().
                    equalsIgnoreCase("admin");
                return new User(username.trim(), password.trim
                    (), isAdmin);
            }
        }
    } catch (IOException e) {
        System.out.println("Errore durante l'autenticazione: "
            + e.getMessage());
    }
    return null;
}
```

Listing 3.2: Metodo authenticate() del UserAuthenticator

3.2.4 Classe ClientHandler

Questa classe si occupa di gestire la comunicazione con un singolo client. Implementa l'interfaccia Runnable (o estende Thread), in modo che ogni istanza possa essere eseguita in parallelo rispetto alle altre. Al suo interno, il metodo run() contiene il loop di gestione della sessione: ricezione dei comandi dal client (tramite stream di input sul socket) e chiamata dei servizi appropriati sul server (ad esempio metodi di autenticazione o di invio file). ClientHandler funge dunque da controller per la singola connessione, interpretando i messaggi secondo il protocollo applicativo e utilizzando le componenti di business logic (autenticazione, accesso file) per soddisfare le richieste.

```
UserAuthenticator authenticator = new UserAuthenticator("
    credentials.txt");
boolean authenticated = false;

while (!authenticated) {
    String risposta;
    while (true) {
        sendMessage("Sei registrato? (si/no):");
        risposta = in.readLine();
        if (risposta == null || risposta.trim().isEmpty())
            continue;
        if (risposta.equalsIgnoreCase("si") || risposta.
            equalsIgnoreCase("no")) break;
    }

    if (risposta.equalsIgnoreCase("si")) {
        out.write("Inserisci username:\n"); out.flush();
        String usernameInput = in.readLine();
        out.write("Inserisci password:\n"); out.flush();
        String passwordInput = in.readLine();
```

```
18
19          user = authenticator.authenticate(usernameInput,
                passwordInput);
20          if (user != null) {
21              authenticated = true;
22              out.write("Login effettuato con successo!\n");
23              out.write("Benvenuto " + user.getUsername() + (user
                    .isAdmin() ? " [ADMIN]" : "") + "\n");
24              out.flush();
25          } else {
26              out.write("Credenziali errate.\n"); out.flush();
27          }
28      } else {
29          out.write("Registrazione - Inserisci username:\n"); out
                .flush();
30          String newUser = in.readLine();
31          out.write("Inserisci password:\n"); out.flush();
32          String newPass = in.readLine();
33
34          if (authenticator.register(newUser, newPass, false)) {
35              user = authenticator.authenticate(newUser, newPass)
                    ;
36              authenticated = true;
37              out.write("Registrazione completata con successo!\n
                    ");
38              out.write("Benvenuto " + user.getUsername() + "\n")
                    ;
39              out.flush();
40          } else {
41              out.write("Errore: username gia' esistente.\n");
42              out.flush();
43          }
44      }
45  }
```

Listing 3.3: Gestione dell'autenticazione nel `ClientHandler`

```
1  String commandLine;
2  while ((commandLine = in.readLine()) != null) {
3      String trimmed = commandLine.trim();
4
5      if (trimmed.equalsIgnoreCase("exit")) {
6          sendMessage("Chiusura connessione. Arrivederci!");
7          break;
8      }
9
10     Command command = CommandFactory.getCommand(trimmed,
           currentDir, user.isAdmin());
11     if (command != null) {
12         currentDir = command.execute(this);
13     } else {
14         sendMessage("Comando non valido oppure non hai l'
               autorizzazione.");
15     }
16 }
```

Listing 3.4: Loop principale di gestione dei comandi del `ClientHandler`

3.2.5 Package observers

Interfaccia Observer

L'interfaccia Observer definisce il contratto per tutti gli oggetti osservatori, che devono reagire a determinati eventi nel sistema. Include due metodi:

- update(Object obj): chiamato per notifiche generiche (es. azioni degli utenti o eventi di sistema).
- onFileDownloaded(Object obj): chiamato specificamente quando un utente scarica un file, per registrare o monitorare questa azione.

Questo consente di centralizzare e modulare il comportamento di logging o tracciamento.

```
1 public interface Observer {
2     void update(Object obj);
3     void onFileDownloaded(Object obj);
4 }
```

Listing 3.5: Implementazione dell'interfaccia degli Observer

LoggerObserver

LoggerObserver è un'implementazione concreta dell'interfaccia Observer, responsabile del logging a console delle azioni rilevanti

```
1  public class LoggerObserver implements Observer {
2    @Override
3    public void update(Object message) {
4        System.out.println("[LOG] " + message);
5    }
6
7    @Override
8    public void onFileDownloaded(Object obj) {
9      DownloadInfo info = (DownloadInfo) obj;
10     System.out.printf("[DOWNLOAD] Utente %s ha scaricato il
           file %s\n", info.getUsername(), info.getFileName());
11   }
12 }
```

Listing 3.6: Implementazione dell'interfaccia degli Observer

Interfaccia Observable

L'interfaccia Observable rappresenta la parte osservabile nel pattern Observer, ovvero i soggetti che generano eventi (come le azioni di utenti o amministratori).

Definisce:

- Una lista di Observer che si registrano per ricevere notifiche.
- Il metodo addObserver() per aggiungere osservatori.

```
1 public interface Observable {
2     void addObserver(Observer observer);
3 }
```

Listing 3.7: Implementazione dell'interfaccia degli Observable

UserActionObservable

UserActionObservable è un implementazione concreta dell'interfaccia Observable, responsabile di notificare la lista di observer registrati quando un User esegue una qualsiasi operazione.

```
public class UserActionObservable extends AbstractObservable {
    public void notifyUser(String message) {
        notifyObservers(message);
    }
}
```

Listing 3.8: Implementazione dell'interfaccia degli Observer

AdminActionObservable

AdminActionObservable è un implementazione concreta dell'interfaccia Observable, responsabile di notificare la lista di observer registrati quando un Admin esegue una qualsiasi operazione.

```
public class AdminActionObservable extends AbstractObservable {
    public void notifyAdmin(String message) {
        notifyObservers(message);
    }
}
```

Listing 3.9: Implementazione dell'interfaccia degli Observer

AbstractObservable

`AbstractObservable` è la classe astratta che gestisce il meccanismo di registrazione e notifica per tutti i soggetti osservabili. In particolare:

- Mantiene internamente una lista di `Observer`.
- Fornisce il metodo pubblico

  `void addObserver(Observer o)`

  per consentire l'iscrizione dei listener.
- Definisce un metodo di notifica con visibilità `protected`, in modo che solo le sottoclassi e le classi all'interno del medesimo package possano emettere eventi:

```
protected void notifyObservers(Object event) {
    for (Observer o : observers) {
        if (event instanceof DownloadInfo) {
            o.onFileDownloaded(event);
        } else {
            o.update(event);
        }
    }
}
```

3.2.6 Package Commands

Il package Server.Commands incapsula la logica relativa all'esecuzione dei comandi testuali inviati dal client al server. Ogni comando (es. list, cd, upload, download, delete) è modellato come una classe separata che implementa un'interfaccia comune, promuovendo modularità, estensibilità e aderenza al design pattern Command.

Questa struttura consente di:

- separare la gestione dei comandi dalla logica di rete;
- aggiungere facilmente nuovi comandi (basta creare una nuova classe che implementa Command);
- gestire autorizzazioni e controlli specifici per ogni comando.

Classe CommandFactory

CommandFactory è una factory class responsabile dell'istanziazione dinamica dei comandi sulla base della stringa ricevuta dal client. Il metodo:

```
1  public static Command getCommand(String commandLine, File
       currentDir, boolean isAdmin){
2          String[] parts = commandLine.split(" ");
3          String command = parts[0];
4
5          switch (command.toLowerCase()) {
6              case "list":
7                  return new ListCommand(currentDir);
8              case "cd":
9                  if (parts.length > 1) {
10                     return new CdCommand(currentDir, parts[1]);
11                 }
12                 break;
13             case "download":
14                 if (parts.length > 1) {
15                     return new DownloadCommand(currentDir,
                           parts[1]);
16                 }
17                 break;
18             case "upload":
19                 if (isAdmin && parts.length > 1) {
20                   return new UploadCommand(parts[1]);
21                 }
22                 break;
23             case "delete":
24                 if (isAdmin && parts.length > 1) {
25                     return new DeleteCommand(currentDir, parts
                           [1]);
26                 }
27                 break;
28             case "exit":
29                 break;
30         }
31         return null;
32     }
```

Listing 3.10: metodo getCommand della CommandFactory

- Analizza il comando testuale (commandLine) ricevuto dal client.
- Verifica la validità degli argomenti e se l'utente ha i permessi necessari (es. alcuni comandi come upload e delete sono riservati agli admin).
- Restituisce un'istanza della classe comando corrispondente (ListCommand, CdCommand, DownloadCommand, ecc.), oppure null se il comando non è valido.

La CommandFactory incapsula la logica di selezione del comando, evitando if-else o switch sparsi nel codice del server, facilitando la manutenibilità.

Interfaccia Command

L'interfaccia Command definisce un contratto per tutte le classi comando. Contiene un unico metodo:

```
1 public interface Command {
2     File execute(ClientHandler handler);
3 }
```

Listing 3.11: Interfaccia per i comandi

- handler: l'oggetto ClientHandler associato alla sessione client attuale.
- Il valore restituito è una cartella (tipo File) che può rappresentare una modifica allo stato corrente del client (es. cambio directory).

Questo approccio consente al server di eseguire comandi in modo polimorfico, senza conoscere i dettagli dell'implementazione specifica.

Classe CdCommand

Implementazione concreta dell'Interfaccia Command. Implementa il comando per navigare all'interno delle directory dell'archivio dei file:

```
1  public File execute(ClientHandler handler) {
2      File newDir = new File(currentDir, dirName);
3      if (newDir.exists() && newDir.isDirectory()) {
           handler.sendMessage("Directory cambiata in: " + newDir.
           getName());
4          try {
5              UserActionObservable observable = new
                   UserActionObservable();
6              observable.addObserver(new LoggerObserver());
7              observable.notifyObservers("L'utente ha cambiato la
                    directory in: " + newDir.getAbsolutePath());
8          } catch (Exception e) {
9              System.err.println("Errore logging cd: " + e.
                   getMessage());
10         }
11         return newDir;
12     } else {
13         handler.sendMessage("Directory non trovata.");
14         return currentDir;
15     }
16 }
```

Listing 3.12: metodo di esecuzione del comando cd

Classe DownloadCommand

Implementazione concreta dell'Interfaccia Command. Implementa il comando per scaricare in locale i file presenti all'interno dell'archivio del server:

```
public File execute(ClientHandler handler) {
    File file = new File(currentDir, fileName);
    try {
        if (!file.exists() || !file.isFile()) {
            handler.sendMessage("File non trovato.");
            return currentDir;
        }
        byte[] fileBytes = Files.readAllBytes(file.toPath());
        String encoded = Base64.getEncoder().encodeToString(
            fileBytes);
        handler.sendMessage("FILE_CONTENT:" + file.getName() +
            ":" + encoded);
        try {
            DownloadObservable observable = new
                DownloadObservable();
            observable.addObserver(new LoggerObserver());
            observable.notifyObservers(handler.getUsername(),
                file.getName());
        } catch (Exception e) {
            System.err.println("Errore nel logging degli
                observers: " + e.getMessage());
        }
    } catch (IOException e) {
        handler.sendMessage("Errore nel download del file: " +
            e.getMessage());
        e.printStackTrace(); // stampa lato server
    }
    return currentDir;
}
```

Listing 3.13: metodo di esecuzione del comando download

Classe ListCommand

Implementazione concreta dell'Interfaccia Command. Implementa il comando per visualizzare la lista di file scaricabili all'interno della directory del server corrente:

```
public File execute(ClientHandler handler) {
    File[] files = currentDir.listFiles();
    if (files == null || files.length == 0) {
    handler.sendMessage("La directory e' vuota.");
    } else {
        StringBuilder sb = new StringBuilder();
        for (File file : files) {
            sb.append(file.getName()).append(file.isDirectory()
                ? "/" : "").append("\n");
        }
        String list = sb.toString().trim(); // rimuove l'ultimo
            \n
        handler.sendMessage(list);
    }

    try {
```

```
15          UserActionObservable observable = new
                UserActionObservable();
16          observable.addObserver(new LoggerObserver());
17          observable.notifyObservers("Utente ha visualizzato il
                contenuto di: " + currentDir.getAbsolutePath());
18      } catch (Exception e) {
19          System.err.println("Errore logging list: " + e.
                getMessage());
20      }
21
22      return currentDir;
23  }
```

Listing 3.14: metodo di esecuzione del comando list

Classe UploadCommand

Implementazione concreta dell'Interfaccia Command. Implementa il comando, esclusivo per l'Admin, per aggiungere nuovi file all'interno dell'archivio del server:

```
1   public File execute(ClientHandler handler) {
2       // Estrai solo il nome del file
3       String fileName = new File(clientFilePath).getName();
4
5       // Cartella di destinazione fissa
6       File dest = new File(handler.getRootDir(), fileName);
7
8       // Se gia' esiste sul server
9       if (dest.exists()) {
10          handler.sendMessage("Il file '" + fileName + "' esiste
                gia' sul server.");
11          return handler.getCurrentDir();
12      }
13
14      // Prompt al client per inviare i dati
15      handler.sendMessage("Pronto per ricevere il file.");
16
17      try {
18          // Leggi la riga intera inviata dal client
19          String uploadLine = handler.getIn().readLine();
20          if (uploadLine == null || !uploadLine.startsWith("
                FILE_UPLOAD:")) {
21              handler.sendMessage("Formato di upload non valido."
                    );
22              return handler.getCurrentDir();
23          }
24
25          // Splitta in tre parti: [0]="FILE_UPLOAD", [1]=
                nomeFile, [2]=base64Data
26          String[] parts = uploadLine.split(":", 3);
27          if (parts.length < 3) {
28              handler.sendMessage("Dati di upload corrotti.");
29              return handler.getCurrentDir();
30          }
31          String base64Data = parts[2];
32
33          // Decodifica e salva
```

```
34          byte[] fileBytes = Base64.getDecoder().decode(
                base64Data);
35          Files.write(dest.toPath(), fileBytes);
36
37          handler.sendMessage("Upload completato con successo: "
                + fileName);
38
39          // Logging osservatori
40          AdminActionObservable obs = new AdminActionObservable()
                ;
41          obs.addObserver(new LoggerObserver());
42          obs.notifyObservers("Admin ha caricato: " + dest.
                getAbsolutePath());
43      } catch (Exception e) {
44          handler.sendMessage("Errore durante l'upload: " + e.
                getMessage());
45          System.err.println("Errore in UploadCommand: " + e.
                getMessage());
46      }
47      return handler.getCurrentDir();
48  }
```

Listing 3.15: metodo di esecuzione del comando upload

Classe DeleteCommand

Implementazione concreta dell'Interfaccia Command. Implementa il comando, esclusivo per l'Admin, per rimuovere eventuali file dall'archivio del server:

```
1   public File execute(ClientHandler handler) {
2       File file = new File(currentDir, fileName);
3       if (!file.exists()) {
4           handler.sendMessage("File non trovato.");
5       } else if (file.delete()) {
6           handler.sendMessage("File eliminato.");
7           try {
8               AdminActionObservable observable = new
                    AdminActionObservable();
9               observable.addObserver(new LoggerObserver());
10              observable.notifyObservers("Admin ha eliminato il
                    file: " + file.getAbsolutePath());
11          } catch (Exception e) {
12              System.err.println("Errore logging delete: " + e.
                    getMessage());
13          }
14      } else {
15          handler.sendMessage("Tentativo di eliminazione fallito.
                ");
16      }
17      return currentDir;
18  }
```

Listing 3.16: metodo di esecuzione del comando delete

Capitolo 4

Testing del codice

4.1 Tecnologie utilizzate e struttura dei test

Per garantire l'affidabilità e la correttezza del software sviluppato, è statp sviluppato del codice di testing, adottando una strategia mista che combina sia test **white-box** che **black-box**. L'obiettivo è stato quello di verificare sia la corretta implementazione interna del codice che il corretto comportamento esterno rispetto alle specifiche funzionali.

I test sono stati realizzati utilizzando il framework **JUnit 5**, integrato all'interno del progetto tramite il sistema di build **Maven**. L'esecuzione automatica dei test è stata gestita attraverso il plugin **Maven Surefire**, che consente di integrare la fase di testing nel ciclo di build e verificare l'effettiva copertura delle funzionalità implementate. Per una maggiore leggibilità e precisione nell'espressione delle asserzioni, è stata inoltre utilizzata la libreria **AssertJ**, che offre un set di metodi fluenti e intuitivi per la verifica delle condizioni nei test.

Nel dettaglio:

- I test **white-box** sono stati utilizzati per verificare il corretto funzionamento di singoli metodi, condizioni logiche e percorsi di esecuzione interni. Questo ha permesso di individuare e correggere potenziali bug a livello di implementazione.
- I test **black-box** si sono concentrati invece sul comportamento esterno delle componenti principali, validando l'interazione tra input e output senza considerare i dettagli interni dell'implementazione.

Questo approccio ha permesso di ottenere una copertura dei test bilanciata e completa, assicurando il corretto funzionamento delle funzionalità principali e una maggiore robustezza del sistema.

4.2 Codice di Test

4.2.1 white-box

Questi test accedono alla struttura interna del codice, verificando comportamenti specifici, percorsi condizionali e struttura dati. Il codice di test di tipo white-box è stato suddiviso in tre classi di testing:

UserAuthenticatorWhiteBoxTest

Obiettivo: Verificare il comportamento interno della classe UserAuthenticator.
Cosa verifica:

- Che il costruttore crei il file di credenziali se non esiste.
- Che il metodo register scriva correttamente la riga nel file.
- Che authenticate restituisca un oggetto User coerente con i dati memorizzati.
- Che authenticate ritorni null con password errata.

```
1  class UserAuthenticatorWhiteBoxTest {
2      private static final String CRED_FILE = "
           test_credentials_wb.txt";
3      private UserAuthenticator auth;
4      @BeforeEach
5      void init() throws IOException {
6          Files.deleteIfExists(Path.of(CRED_FILE));
7          auth = new UserAuthenticator(CRED_FILE);
8      }
9      @AfterEach
10     void cleanup() throws IOException {
11         Files.deleteIfExists(Path.of(CRED_FILE));
12     }
13     @Test
14     void constructorCreatesFileIfMissing() {
15         assertThat(Files.exists(Path.of(CRED_FILE)))
16                 .as("Il costruttore deve creare il file delle
                       credenziali")
17                 .isTrue();
18     }
19     @Test
20     void registerAppendsCorrectLine() throws IOException {
21         auth.register("federico", "pass", true);
22         String content = Files.readString(Path.of(CRED_FILE));
23         assertThat(content).contains("federico pass admin");
24     }
25     @Test
26     void authenticateReturnsCorrectUserObject() {
27         auth.register("samuel", "pw", false);
28         User u = auth.authenticate("samuel", "pw");
29         assertThat(u)
30                 .satisfies(user -> {
31                     assertThat(user.getUsername()).isEqualTo("
                           samuel");
32                     assertThat(user.isAdmin()).isFalse();
33                 });
34     }
35     @Test
36     void authenticateReturnsNullOnBadPwd() {
37         auth.register("egidio", "pwd", false);
38         assertThat(auth.authenticate("egidio", "wrong")).isNull
               ();
39     }
40 }
```

Listing 4.1: classe di test UserAuthenticatorWhiteBoxTest

ServerWhiteBoxTest

Obiettivo: Verificare il corretto funzionamento del singleton Server.
Cosa verifica:

- Che getInstance() restituisca sempre la stessa istanza.
- Che il file delle credenziali sia correttamente inizializzato.

```
class ServerWhiteBoxTest {
    @Test
    void singletonBehaviorAndCredentialsFilePath() {
        Server s1 = Server.getInstance();
        Server s2 = Server.getInstance();
        assertThat(s1).isSameAs(s2);
        String credPath = s1.getCredentialsFile().getPath();
        assertThat(credPath)
                .as("Il server deve avere un file di
                    credenziali configurato")
                .isNotEmpty();
    }
}
```

Listing 4.2: classe di test ServerWhiteBoxTest

ObserverWhiteBoxTest

Obiettivo: Controllare che gli observer vengano notificati correttamente in caso di download.
Cosa verifica:

- Che l'aggiunta di un observer funzioni.
- Che la chiamata a notifyDownload() invochi effettivamente onFileDownloaded() sugli observer registrati.

```
class ObserverWhiteBoxTest {
    private static class SpyObserver extends LoggerObserver {
        boolean notified = false;
        @Override
        public void onFileDownloaded(Object obj) {
            if (obj instanceof DownloadInfo) {
                notified = true;
            }
        }
    }
    @Test
    void notifyDownloadInvokesObserverMethod() {
        Server srv = Server.getInstance();
        SpyObserver spy = new SpyObserver();
        srv.addObserver(spy);

        srv.notifyDownload("utenteTest", "report.pdf");
        assertThat(spy.notified).as("L'Observer deve essere
            notificato in caso di download").isTrue();
    }
}
```

Listing 4.3: classe di test ObserverWhiteBoxTest

4.2.2 black-box

Questi test verificano il comportamento osservabile del sistema senza conoscere la struttura interna del codice, simulando il punto di vista di un utente o altro sistema esterno. Il codice di test di tipo black-box è stato suddiviso in tre classi di testing:

AuthBlackBoxTest

Obiettivo: Testare la registrazione e l'autenticazione degli utenti tramite l'interfaccia pubblica di UserAuthenticator.
Cosa verifica:

- La possibilità di registrare un utente e accedere con credenziali corrette.
- Il blocco della registrazione di un utente con username già esistente.
- Il fallimento dell'autenticazione con password sbagliata o username inesistente.

```
1  class AuthBlackBoxTest {
2      private static final String TEST_CRED_FILE = "
           test_credentials_bb.txt";
3      private UserAuthenticator auth;
4      @BeforeEach
5      void setUp() throws IOException {
6          // Assicuriamoci di partire sempre da un file vuoto
7          Files.deleteIfExists(Path.of(TEST_CRED_FILE));
8          auth = new UserAuthenticator(TEST_CRED_FILE);
9      }
10     @AfterEach
11     void tearDown() throws IOException {
12         Files.deleteIfExists(Path.of(TEST_CRED_FILE));
13     }
14     @Test
15     void registerNewUserAndLoginSuccess() {
16         assertTrue(auth.register("aaa", "pwd123", false));
17         User u = auth.authenticate("aaa", "pwd123");
18         assertNotNull(u);
19         assertEquals("aaa", u.getUsername());
20         assertFalse(u.isAdmin());
21     }
22     @Test
23     void cannotRegisterSameUserTwice() {
24         assertTrue(auth.register("alice", "secret", true));
25         assertFalse(auth.register("alice", "secret", true));
26     }
27     @Test
28     void loginWithWrongPasswordFails() {
29         auth.register("testLogin", "pwd", false);
30         assertNull(auth.authenticate("testLogin", "wrongpwd"));
31     }
32     @Test
33     void loginNonexistentUserFails() {
34         assertNull(auth.authenticate("pippo", "nopass"));
35     }
36 }
```

Listing 4.4: classe di test AuthBlackBoxTest

CommandFactoryBlackBoxTest

Obiettivo: Verificare la corretta creazione dei comandi in base al ruolo utente (admin o normale).
Cosa verifica:

- Gli utenti normali possono eseguire solo list, cd e download.
- Gli admin possono anche upload e delete.
- Comandi non validi restituiscono null.

```
class CommandFactoryBlackBoxTest {
    private final File root = new File("server_files");
    @Test
    void regularUserCanUseOnlyListCdDownload() {
        assertTrue(CommandFactory.getCommand("list", root,
           false) instanceof ListCommand);
        assertTrue(CommandFactory.getCommand("cd subdir", root,
            false) instanceof CdCommand);
        assertTrue(CommandFactory.getCommand("download file.txt
           ", root, false) instanceof DownloadCommand);
    }
    @Test
    void regularUserCannotUploadOrDelete() {
        assertNull(CommandFactory.getCommand("upload file.txt",
            root, false));
        assertNull(CommandFactory.getCommand("delete file.txt",
            root, false));
    }
    @Test
    void adminCanUseUploadAndDelete() {
        assertTrue(CommandFactory.getCommand("upload new.txt",
           root, true) instanceof UploadCommand);
        assertTrue(CommandFactory.getCommand("delete old.txt",
           root, true) instanceof DeleteCommand);
    }
    @Test
    void unknownCommandsReturnNull() {
        assertNull(CommandFactory.getCommand("testcomando",
           root, true));
        assertNull(CommandFactory.getCommand("", root, false));
    }
}
```

Listing 4.5: classe di test CommandFactoryBlackBoxTest

FileOperationsBlackBoxTest

Obiettivo: Testare direttamente le funzionalità principali legate ai file dal punto di vista utente.
Cosa verifica:

- Il download effettivo dei file da server_files/ a downloaded_files/.
- L'upload corretto da file_to_upload/ a server_files/ solo da admin.
- La rimozione di file da server_files/ solo da parte di un admin.

```java
class FileOperationsBlackBoxTest {
    private final File root = new File("server_files");
    private final User admin = new User("a","p",true);
    private final User user  = new User("u","p",false);
    void downloadCommandSendsBase64Content() throws Exception {
        // 1) Assicuriamoci che il file esista
        Path src = Path.of("server_files/file.txt");
        assertTrue(Files.exists(src), "server_files/file.txt
            deve esistere");

        // 2) Recuperiamo il comando
        Command cmd = CommandFactory.getCommand("download file.
            txt", root, false);
        assertNotNull(cmd);
        assertTrue(cmd instanceof DownloadCommand,
                "getCommand su 'download' deve restituire
                    DownloadCommand");

        // 3) Creiamo il nostro handler di test che registra i
            messaggi
        ClientHandlerForTesting handler = new
            ClientHandlerForTesting(root);

        // 4) Eseguiamo il comando
        File after = cmd.execute(handler);

        // 5) La directory corrente non cambia
        assertEquals(root.getCanonicalFile(), after.
            getCanonicalFile());

        // 6) Verifichiamo fra i messaggi registrati quello con
             FILE_CONTENT
        List<String> msgs = handler.getMessages();
        assertFalse(msgs.isEmpty(), "handler deve aver ricevuto
             almeno un messaggio");

        boolean found = msgs.stream()
                .anyMatch(m -> m.startsWith("FILE_CONTENT:file.
                    txt:"));
        assertTrue(found, "Deve esserci un messaggio che inizia
             con 'FILE_CONTENT:file.txt:'");
    }
    void uploadAddsFileOnlyForAdmin() throws Exception {
        // Assicuriamoci che esista il file di test da "
            caricare"
        Path up = Path.of("file_to_upload/upload_test.txt");
        assertTrue(Files.exists(up), "file_to_upload/
            upload_test.txt deve esistere");

        // Prepara i dati Base64 per quel file
        byte[] bytes = Files.readAllBytes(up);
        String base64 = Base64.getEncoder().encodeToString(
            bytes);

        // Costruiamo il comando upload
        Command cmdAdmin = CommandFactory.getCommand("upload
            upload_test.txt", root, true);
```

```
44          assertNotNull(cmdAdmin, "Admin deve poter ottenere l'
                UploadCommand");
45
46          // Creiamo il nostro handler di test, che risponde con
                la riga di upload
47          class UploadTestHandler extends ClientHandler {
48              UploadTestHandler(File rootDir) {
49                  super((Socket) null, rootDir);
50                  // preparo il reader con la riga "FILE_UPLOAD:
                        upload_test.txt:<base64>"
51                  String line = "FILE_UPLOAD:upload_test.txt:" +
                        base64 + "\n";
52                  this.in = new BufferedReader(new StringReader(
                        line));
53              }
54              @Override
55              public void sendMessage(String message) {
56                  // no-op: ignoriamo i prompt
57              }
58          }
59          // Usa il handler per eseguire
60          UploadTestHandler handler = new UploadTestHandler(root)
                ;
61          File after = cmdAdmin.execute(handler);
62          // Ora il file deve esistere in server_files
63          assertTrue(Files.exists(Path.of("server_files/
                upload_test.txt")),
64                  "Il file deve finire in server_files");
65          assertEquals(root.getCanonicalFile(), after.
                getCanonicalFile());
66
67          // Puliamo per non lasciare tracce
68          Files.deleteIfExists(Path.of("server_files/upload_test.
                txt"));
69      }
70          void deleteRemovesFileOnlyForAdmin() throws Exception {
71          Path toDelete = Path.of("server_files/tmp.txt");
72          Files.writeString(toDelete, "temp");
73          assertTrue(Files.exists(toDelete));
74
75          // Utente normale non ottiene il comando
76          assertNull(CommandFactory.getCommand("delete tmp.txt",
                root, false));
77
78          // Comando delete per admin
79          Command cmd = CommandFactory.getCommand("delete tmp.txt
                ", root, true);
80          assertNotNull(cmd);
81
82          ClientHandlerForTesting adminHandler = new
                ClientHandlerForTesting(root);
83          File afterDelete = cmd.execute(adminHandler);
84
85          assertFalse(Files.exists(toDelete), "Il file deve
                essere rimosso da server_files");
86          assertEquals(root.getCanonicalFile(), afterDelete.
                getCanonicalFile());
87      }
```

Listing 4.6: classe di test FileOperationsBlackBoxTest

4.3 Risultati dei test

Il codice dei test descritto precedentemente viene eseguito correttamente senza nessun errore come si può vedere:

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running blackbox.AuthBlackBoxTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.076 s - in blackbox.AuthBlackBoxTest
[INFO] Running blackbox.CommandFactoryBlackBoxTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 s - in blackbox.CommandFactoryBlackBoxTest
[INFO] Running blackbox.FileOperationsBlackBoxTest
[LOG] Admin ha caricato: C:\Users\monet\GitHub\ServerTransfer\server_files\upload_test.txt
[LOG] Admin ha eliminato il file: C:\Users\monet\GitHub\ServerTransfer\server_files\tmp.txt
[DOWNLOAD] L'utente User non valido ha scaricato il file: file.txt
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.04 s - in blackbox.FileOperationsBlackBoxTest
[INFO] Running whitebox.ObserverWhiteBoxTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.064 s - in whitebox.ObserverWhiteBoxTest
[INFO] Running whitebox.ServerWhiteBoxTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 s - in whitebox.ServerWhiteBoxTest
[INFO] Running whitebox.UserAuthenticatorWhiteBoxTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.02 s - in whitebox.UserAuthenticatorWhiteBoxTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 17, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.408 s
[INFO] Finished at: 2025-06-09T21:12:27+02:00
[INFO] ------------------------------------------------------------------------

Process finished with exit code 0
```

Figura 4.1: Risultati dei Test