Software Engineering Audit Report

Relazione_ServerTransfer.pdf

CAPRA

February 25, 2026

Contents

1 Document Context 2

2 Executive Summary 3

3 Strengths 4

4 Expected Feature Coverage 4

5 Summary Table 7

6 Issue Details 7

6.1 Architecture (2 issues) 7

6.2 Requirements (7 issues) 8

6.3 Testing (3 issues) 11

7 Priority Recommendations 12

8 Traceability Matrix 12

9 Terminological Consistency 13

1 Document Context

Project Objective

ServerTransfer is a client-server file transfer application developed in Java that enables remote users to authenticate and access a centralized file archive. The system implements a custom FTP-like service where clients connect to a dedicated server, perform login with credentials, and download authorized files based on assigned permissions. The application supports concurrent multi-user access with role-based functionality for regular users and administrators.

Main Use Cases

- UC-1 – Registration: A user registers by providing username and password; the server verifies uniqueness and adds credentials to the authorized users list.
- UC-2 – Login: An authenticated user provides credentials; the server verifies them against stored data and grants access upon success.
- UC-3 – List Files: An authenticated user requests the list of files and directories available in the current server directory.
- UC-4 – Change Directory (Cd): A user navigates between directories on the server to locate desired files.
- UC-5 – Download File: An authenticated user selects and downloads a file from the server to a local directory via reliable TCP/IP transfer.
- UC-6 – Upload File: An admin uploads a new file to the server; the system verifies the file does not already exist before saving.
- UC-7 – Delete File: An admin removes a file from the server's file archive.
- UC-8 – Exit: A user terminates the session; the server releases allocated resources and closes the connection cleanly.

Functional Requirements

- User authentication via username and password verification against a credentials file.
- User registration with duplicate username prevention.
- File listing in the current server directory with distinction between files and directories.
- Directory navigation using the `cd` command with validation of directory existence.
- File download with Base64 encoding for reliable transmission over TCP/IP sockets.
- File upload functionality restricted to admin users with duplicate file detection.
- File deletion functionality restricted to admin users.
- Concurrent multi-user support with dedicated thread per client connection.
- Secure session termination with proper resource cleanup.
- Role-based access control distinguishing between regular users and administrators.
- Logging of user actions and file operations through the Observer pattern.

Non-Functional Requirements

- Multi-threaded server architecture supporting concurrent client connections.
- Reliable file transfer using TCP/IP protocol with Base64 encoding.
- Singleton pattern implementation for centralized server instance management.
- Modular command architecture using Command and Factory patterns for extensibility.
- Persistent credential storage in text file format (`credentials.txt`).

- Console-based user interface for client application.
- Event-driven logging system using Observer pattern for monitoring downloads and administrative actions.
- Server listening on predefined port 12345.

Architecture

The application follows a two-tier multi-threaded client-server architecture. The **Server** component is implemented as a Singleton that listens on TCP port 12345, manages authentication via `UserAuthenticator`, and creates a dedicated `ClientHandler` thread for each incoming connection. The **Client** is a console application that connects to the server, performs login/registration, and sends textual commands. Communication occurs via TCP sockets using a text-based protocol. The server employs the **Command** pattern with a **CommandFactory** to dynamically instantiate command objects (`ListCommand`, `DownloadCommand`, `UploadCommand`, `DeleteCommand`, `CdCommand`). The **Observer** pattern is used for event notification through `DownloadObservable`, `UserActionObservable`, and `AdminActionObservable` classes that notify `LoggerObserver` instances. File transfer uses Base64 encoding for reliable transmission. The architecture is organized into packages: `Client`, `Server`, `Server.Commands`, and `Server.Observers`.

Testing Strategy

A comprehensive testing strategy combining white-box and black-box approaches was implemented using JUnit 5 with Maven Surefire for automated execution and AssertJ for fluent assertions. **White-box tests** verify internal implementation details: `UserAuthenticatorWhiteBoxTest` validates credential file creation, user registration, and authentication logic; `ServerWhiteBoxTest` confirms Singleton behavior and credentials file initialization; `ObserverWhiteBoxTest` verifies observer notification mechanisms. **Black-box tests** validate external behavior: `AuthBlackBoxTest` tests user registration and login workflows; `CommandFactoryBlackBoxTest` verifies role-based command creation and authorization; `FileOperationsBlackBoxTest` tests file download, upload, and deletion operations. All 17 tests execute successfully with zero failures, confirming correct implementation of functional and non-functional requirements.

2 Executive Summary

Quick Overview

Total issues: **12** — HIGH: **2** MEDIUM: **8** LOW: **2**

Average confidence: **94%**

Executive Summary: Audit of Relazione_ServerTransfer.pdf

This document is a Software Engineering report for a university project implementing a client-server file transfer system with user authentication, role-based access control, and multi-threaded request handling. The report includes requirements specification (use cases, actors, functional descriptions), architectural design (class diagrams, design patterns, command pattern), implementation details (code listings), and testing documentation. The purpose is to provide comprehensive documentation of the system's design and validation.

The audit identified 12 issues across three categories. In **Architecture**, the Observer pattern implementation does not align with the documented design: observable objects are instantiated per command rather than reused, and the Command interface signature is inconsistent between the class diagram, textual description, and actual code listings. In **Requirements**, the most critical finding is a fundamental contradiction between the introduction (only pre-registered users allowed) and the detailed use cases (self-registration at runtime). Additional patterns include incomplete use case coverage (only 3 of 8 operations have detailed templates), missing alternative flows and error scenarios in use case descriptions, and a mismatch between the GUI mockups presented and the console-based implementation actually delivered. Actor naming is also inconsistent (ExpectedUser/NormalUser in diagrams versus User/Admin in text). In **Testing**, the primary concern is the absence of end-to-end integration tests covering the full client-server protocol and socket communication, and no verification of multi-threaded concurrency behavior despite this being a stated architectural objective.

Critical Areas (HIGH Severity): Two issues require immediate attention. ISS-003 represents a fundamental contradiction in the functional requirements that must be resolved before the system can be properly validated. TST-001 indicates that the core system behavior (connection handling, protocol interaction, multi-threaded request processing) is not tested, leaving the main functionality unverified.

Overall Quality Assessment: The document demonstrates reasonable structure and covers the required sections of a Software Engineering report. However, it suffers from significant internal inconsistencies between requirements, design, and implementation, incomplete specification of error handling and edge cases, and insufficient test coverage of critical system behaviors. These issues limit confidence in the correctness and completeness of both the specification and the validation.

Priority Actions:

1. Resolve the contradiction between the authentication model (ISS-003) by clarifying whether the system requires pre-registered users or supports runtime registration, then update all affected sections (introduction, use cases, implementation) consistently.
2. Implement end-to-end integration tests (TST-001) that exercise the full client-server protocol over sockets, including login/registration flows, command execution, and multi-user scenarios to verify the main system behavior.
3. Reconcile the Command interface definition (ISS-002) by aligning the class diagram, textual description, and code listings, and complete the use case specification (ISS-005, ISS-006) by adding detailed templates with preconditions, alternative flows, and error handling for all eight operations shown in the use case diagram.

3 Strengths

- **Comprehensive Design Documentation:** The document provides well-structured design artifacts including Use Case Diagrams, Use Case Templates, Class Diagrams, Package Diagrams, and Page Navigation Diagrams. These UML diagrams clearly illustrate the system architecture, component interactions, and user workflows, demonstrating thorough planning before implementation.
- **Appropriate Design Pattern Application:** The project correctly implements three established design patterns—Observer, Singleton, and Command+Factory—with clear justification for each choice. The patterns are well-integrated into the architecture (e.g., Observer for decoupling logging from core logic, Singleton for centralized server management, Command+Factory for dynamic command handling).
- **Multi-threaded Concurrent Architecture:** The system demonstrates solid architectural design with a multi-threaded client-server model where each client connection spawns a dedicated ClientHandler thread. This enables concurrent user support without interference, addressing a key functional requirement.
- **Dual Testing Approach:** The project includes both white-box and black-box testing strategies across multiple test classes (UserAuthenticatorWhiteBoxTest, ServerWhiteBoxTest, AuthBlackBoxTest, CommandFactoryBlackBoxTest, FileOperationsBlackBoxTest), indicating comprehensive test coverage methodology.
- **Clear Package Organization:** The codebase is logically organized into well-defined packages (Client, Server, Server.Commands, Server.Observers) with clearly documented responsibilities, promoting modularity and maintainability.
- **User Interface Mockups:** The inclusion of detailed UI mockups for Registration, Login, and Dashboard pages (both User and Admin variants) demonstrates consideration of user experience and provides clear visual specifications for the intended application interface.

4 Expected Feature Coverage

Of 7 expected features: 4 present, 2 partial, 1 absent. Average coverage: **68%**.

Feature	Status	Coverage	Evidence
Unit testing framework implementation	Present	80% (4/5)	The document describes JUnit 5 and Maven Surefire usage, shows assertion-based tests (assertThat, assertTrue, assertNull, etc.), defines multiple dedicated test classes (UserAuthenticatorWhiteBoxTest, ServerWhiteBoxTest, ObserverWhiteBoxTest, AuthBlackBoxTest, CommandFactoryBlackBoxTest, FileOperationsBlackBoxTest), and distinguishes white-box (unit-like) and black-box tests. Mock-style isolated testing is shown via SpyObserver and custom ClientHandlerForTesting/UploadTestHandler, but there is no explicit mention of broader integration testing beyond component-level black-box tests.
Use of UML Diagrams for system modeling	Present	100% (8/8)	The report includes a Use Case Diagram with actors and use cases, a UML Class Diagram, and a Package Diagram describing Client, Server, Server.Commands, and Server.Observers. It provides multiple UI mockups (registration, login, User and Admin dashboards) and Page Navigation Diagrams for User and Admin, explicitly showing relationships between actors and use cases and navigation flows. These diagrams are used to clarify functional requirements and use standard UML-like notation (use case, class, package, flowchart).
Identification and definition of system actors	Present	100% (8/8)	The document clearly identifies two user roles, User and Admin, and describes their responsibilities (User can register/login, navigate folders, download; Admin can also upload/delete). Interactions with system components are detailed via use cases, class descriptions (User, UserAuthenticator, ClientHandler), and commands (list, cd, download, upload, delete). Use Case Templates and navigation diagrams structure user requirements and functional differences per role, and list concrete user actions to guide development.

Feature	**Status**	**Coverage**	**Evidence**
Definition and Documentation of Use Cases	Present	100% (5/5)	Section 2.2 provides Use Case Templates for Registrazione, Download di un file, and Upload di un file, each with Description (user goal), Actor, Pre-condition, Normal Flow, Alternative Flow, and Post-Conditions. The Use Case Diagram shows relationships such as «precedes» and «includes» between Registration, Login, Choose download directory, and Commands and its subcases (Cd, List, Download, Upload, Delete, Exit), clearly defining interactions, goals, flows, and relationships.
User interface and interaction design principles	Partial	57% (4/7)	The document includes detailed UI mockups for registration, login, User dashboard, and Admin dashboard, with structured input fields (Username, Password) and role-specific actions (Carica file, Download, Elimina file). Page Navigation Diagrams for User and Admin show clear navigation elements and flows between pages. Distinct user roles and their functionalities are described. However, there is no description of a step-by-step reservation creation process (domain is file transfer, not reservations), no explicit documentation of input validation mechanisms in the UI, and no mention of dynamic UI updates based on user selections.
Separation of concerns in software architecture	Partial	40% (2/5)	The architecture section and Package Diagram describe separation into Client, Server, Server.Commands, and Server.Observers, and the roles of classes like ClientHandler, User, UserAuthenticator, and command classes, demonstrating modular design and clear component responsibilities. However, there is no explicit MVC or Service/DAO layering: no distinct View, Controller, Service, or DAO packages are defined, and interactions between such layers are not documented.

Feature	Status	Coverage	Evidence
Data Access Object (DAO) pattern	Absent	0% (0/7)	The system uses a simple file-based storage (credentials.txt and server_files directory) accessed directly by classes like UserAuthenticator and command implementations. There are no DAO interfaces, no CRUD methods encapsulating SQL, no separate DAO classes per entity, and no discussion of DAO patterns or testing strategies for DAOs.

5 Summary Table

Category	HIGH	MEDIUM	LOW	Total
Architecture	0	2	0	2
Requirements	1	5	1	7
Testing	1	1	1	3
Total	**2**	**8**	**2**	**12**

6 Issue Details

6.1 Architecture (2 issues)

ISS-001 — MEDIUM [71%] — Page 20 The described Observer pattern design (with DownloadObservable, UserActionObservable, AdminActionObservable and Server extending DownloadObservable) is not fully consistent with the implementation and with how tests use observers. Tests interact directly with Server.addObserver/notifyDownload, while the commands instantiate new observable objects on each call instead of reusing a shared observable, which weakens the intended decoupling and centralization of logging.

> *DownloadObservable ‘e una classe che rappresenta un oggetto osservabile specializzato nel contesto del download, ma non implementa direttamente l’interfaccia Observable tradizionale. ... Inoltre, DownloadObservable viene estesa dalla classe Server, che ne eredita la funzionalit‘a di notifica e la integra nel contesto pi‘u ampio della gestione server, mantenendo cos‘ı un controllo pi‘u fine sui meccanismi di comunicazione degli eventi relativi al download. ... ObserverWhiteBoxTest Obiettivo: Controllare che gli observer vengano notificati correttamente in caso di do- wnload. ... Server srv = Server.getInstance (); SpyObserver spy = new SpyObserver (); srv.addObserver(spy); ... srv.notifyDownload ("utenteTest", "report.pdf");*

Recommendation: Align the implementation and tests with the described Observer architecture. Two concrete options: (1) Centralize observables in Server: keep Server extending DownloadObservable and add fields for UserActionObservable and AdminActionObservable inside Server. Expose methods like addDownloadObserver, addUserActionObserver, addAdminActionObserver, and have commands call these shared observables instead of creating new instances each time. Update ObserverWhiteBoxTest to verify notifications through these centralized observables. (2) If you prefer the current per-command instantiation, update the design section to reflect that observables are created on demand and are not shared via Server, and adjust the class diagram accordingly. In both cases, ensure that tests cover the actual pattern you use (e.g., a test that executes a command and verifies that its observable notifies a registered LoggerObserver), so that the documented design and the code/tests remain consistent.

ISS-002 — MEDIUM [100%] — Page 21 The class diagram and the textual description of the Command interface are inconsistent with the code examples and with each other. The diagram shows ‘+execute(handler: ClientHandler, args: String): File‘, the text shows ‘File execute(ClientHandler handler);‘, while the ClientHandler loop (Listing 3.4) calls ‘command.execute(this, parts);‘ and the concrete commands in Listings 3.10–3.14 have signatures like ‘File execute(ClientHandler handler)‘. This inconsistency makes it unclear what the actual command interface is and how arguments are passed, which affects the correctness of the design documentation.

> *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 3.9: metodo getCommand della CommandFactory*

Recommendation: Align the Command interface and all related descriptions: 1) Decide the definitive method signature for Command (e.g., ‘File execute(ClientHandler handler, String[] args)‘ or ‘File execute(ClientHandler handler)‘ with arguments embedded in the command object). 2) Update: - The UML class diagram (Section 2.3) to show the correct signature. - The textual description of the interface in Section 3.2.6. - The code snippets for concrete commands so they all implement the same signature. - The ClientHandler loop (Listing 3.4) so that the call to ‘execute‘ matches the interface. 3) Briefly explain in the text how command arguments are passed (via constructor vs via args parameter) so the design is coherent and understandable for the examiner.

6.2 Requirements (7 issues)

ISS-003 — HIGH [100%] — Page 4 The introduction states that all download functionality is only available after successful login, but the use case diagram and templates introduce a full registration flow (Use Case 1) and allow users to register and then immediately access all functions. This creates a contradiction between the high-level requirement (only pre-registered users, credentials already known to the server) and the detailed behavior (self-registration at runtime).

> *Non ‘e previsto un utilizzo da parte di utenti non autenticati: tutte le funzionalit‘a di download sono disponibili solo dopo il successo del login.*

Recommendation: Clarify and align the business rule about user registration across the document. Decide one of the following and update all sections consistently: - Option A: Only pre-registered users exist. Then remove or heavily restrict the "Registrazione" use case and related flows in ClientHandler (lines 28–44 of Listing 3.3), and state explicitly that credentials are provisioned offline by an admin. - Option B: Self-registration is allowed. Then update Section 1.1 to remove "Ciascun utente deve essere preventivamente registrato (ossia le sue credenziali sono note al server)" and instead state that users can create an account via the registration use case. Also clarify whether admins can be created via registration or only via manual editing of credentials.txt. Make sure the use case diagram, Use Case 1 template, and the textual description of UserAuthenticator all reflect the same registration policy.

ISS-004 — MEDIUM [100%] — Page 7 The use case diagram introduces two actors (ExpectedUser, NormalUser) that are not used or explained anywhere else in the document. The textual description and templates instead use "User" and "Admin" as roles. This inconsistency in actor naming and roles makes it unclear which actor can perform which use case and how admin-only operations (upload, delete) relate to the diagram.

> *[IMAGE 1]: 1. Diagram Type: Use Case Diagram 2. Elements: - Actors: ExpectedUser, NormalUser - Use Cases: Registration, Login, Choose download directory, Cd, Commands, Exit, List, Download, Upload, Delete*

Recommendation: Unify the actor model across the entire document: - Decide on a consistent set of actors, e.g., "User" and "Admin". - Update the use case diagram to replace "ExpectedUser" and "NormalUser" with these actors, and show explicitly which use cases are available to each (e.g., Admin extends User and has Upload/Delete). - In the Use Case Templates, explicitly state the actor for each use case and ensure it matches the diagram (e.g., Upload/Delete: Actor = Admin; List/Download/Cd: Actor = User or Admin). - Add a short textual subsection describing each actor and its permissions, and ensure it matches the implementation (User.isAdmin, CommandFactory checks).

ISS-005 — MEDIUM [100%] — Page 7 The use case templates cover only three use cases (Registration, Download, Upload) while the use case diagram defines additional important operations (List, Cd, Delete, Exit, Choose download directory, Commands). This leaves several core behaviors without detailed requirements (preconditions, normal flow, alternative flows, postconditions), which weakens traceability and completeness of the functional specification.

> *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 pi'u importanti o rappresentativi.*

Recommendation: Extend the Use Case Templates section so that every use case in Figure 2.1 has at least a minimal template. In particular, add templates for: - List (visualizzazione dei file disponibili) - Cd (navigazione tra directory) - Delete (eliminazione file da parte dell'admin) - Upload (already present but ensure it matches the diagram semantics) - Exit (logout/chiusura connessione) - Choose download directory (if kept as a separate use case) For each new template, specify: Actor, clear preconditions (e.g., user authenticated, correct role), normal flow, alternative/error flows (e.g., directory not found, permission denied), and postconditions. This will make the requirements set complete and easier to map to tests.

ISS-006 — MEDIUM [100%] — Page 8 The Download use case mixes client-side and server-side responsibilities and omits several important alternative flows and constraints. It does not specify what happens if the user has no permission for the file, if the network connection fails mid-transfer, if Base64 decoding or file writing fails on the client, or if the chosen local directory is invalid. Similar omissions appear in the Upload and Registration use cases, where only a single simple error case is described.

> *Use Case 2 Download di un file ... Pre-condition L'utente 'e 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 'e presente il file che vuole scaricare 3. esegue il comando di download, specificando il file da vo- ler 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*

Recommendation: For all main transactional use cases (Download, Upload, Registration, Delete): 1) Separate clearly what the client does and what the server does in the normal flow (e.g., client sends command, server validates permissions, server reads file, server sends data, client decodes and writes file). 2) Add alternative flows for: - Permission errors (user not allowed to access a file or command). - Network or I/O failures (file not readable, disk full, connection lost). - Invalid parameters (invalid path, empty filename, invalid local directory). 3) For each alternative flow, state the system behavior: what is rolled back, what error message is shown, and what the user can do next (retry, return to menu, abort session). Update the postconditions to reflect both success and failure cases (e.g., on failure, no partial file remains on server/client, state is unchanged).

ISS-007 — MEDIUM [100%] — Page 12 The document presents detailed GUI mockups and page navigation diagrams (Registration Page, Login Page, Dashboard, etc.), but the implemented system is described as a console client with textual commands. There is no explicit requirement stating that a GUI must be implemented, and the mockups are only "ipotetica". This can confuse the scope of the project and the examiner about which behaviors are actually implemented versus only envisioned.

> *Per facilitare la comprensione del flusso di utilizzo dell'applicazione, sono stati realizza- ti alcuni mockup delle schermate principali, rappresentanti un'ipotetica GUI per questo programma.*

Recommendation: Clarify the status of the GUI in the requirements and analysis: - In the introduction or in Section 2.4, explicitly state that the current implementation is console-based and that the mockups represent a possible future GUI, not part of the delivered software. - In the Page Navigation Diagrams section, add a short note that these diagrams abstract the logical flow of operations (registration, login, download, etc.) and are not tied to an implemented web/desktop UI. - Optionally, add a short subsection "Out of scope" listing the GUI as future work. This will avoid misunderstandings about missing GUI code and keep the focus on the implemented business logic and protocol.

ISS-008 — MEDIUM [100%] — Page 27 The testing section claims to verify external behavior against functional specifications, but there is no explicit mapping between use cases/requirements and the implemented tests. Some important behaviors from the use case diagram (e.g., Cd navigation error cases, Exit/logout behavior, registration flow as a whole, permission errors for non-admins) are only partially or indirectly tested. This weakens traceability from requirements to tests.

> *L'obiettivo ‘e stato quello di verificare sia la corretta implementazione interna del codice che il corretto comportamento esterno rispetto alle specifiche funzionali.*

Recommendation: Add a concise traceability table or paragraph mapping tests to use cases/requirements. For each main use case (Registration, Login, List, Cd, Download, Upload, Delete, Exit): - Indicate which test class and test methods cover it (e.g., Registration/Login → AuthBlackBoxTest.*; Command permissions → CommandFactoryBlackBoxTest.*; File operations → FileOperationsBlackBoxTest.*). - Identify any use cases or important alternative flows that are not covered (e.g., Cd to non-existing directory, Exit command behavior, registration failure path) and, if time permits, add at least one black-box test per uncovered critical behavior. This will make the link between requirements and tests explicit and strengthen the justification that the system has been adequately validated.

ISS-009 — LOW [100%] — Page 4 The text claims that file transfer is "affidabile" and that the file arrives "integro" at destination, but there are no explicit non-functional requirements or mechanisms described to ensure integrity beyond using TCP and Base64. There is no mention of checksums, file size verification, or retry logic. This overstates the guarantees provided by the current design and may be considered an unsubstantiated requirement.

> *Il trasferimento avviene in modo affidabile tramite protocollo TCP/IP, assicurando che il file giunga integro a destinazione*

Recommendation: Either weaken or justify the integrity claim: - Option A (simpler): Rephrase the requirement to something like "Il trasferimento avviene tramite TCP/IP; in assenza di errori di rete, il file viene trasferito correttamente" and avoid strong guarantees about integrity. - Option B: If you want to keep the integrity guarantee, add a brief description of a simple integrity mechanism (e.g., sending file size and verifying it on the client, or computing a checksum/MD5 and comparing it after transfer) and, if possible, a small test that verifies mismatch detection. Also consider adding a short non-functional requirements subsection (performance, concurrency, reliability) to make such claims explicit and aligned with the actual implementation.

6.3 Testing (3 issues)

TST-001 — HIGH [99%] — Page 29 Core end-to-end client–server interaction flows (login/registration + command loop over sockets) are not tested; all black-box tests exercise components in isolation, not the full protocol between Client and Server/ClientHandler. This leaves the main system behavior (connection handling, text protocol, multi-threaded interaction) unverified.

> *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 'e stato suddiviso in tre classi di testing: AuthBlackBoxTest ... CommandFactoryBlackBoxTest ... FileOperationsBlackBoxTest*

Recommendation: Add at least one integration / end-to-end test that starts a real Server (or a ServerSocket on a random free port) and a Client (or a lightweight client stub) that: (1) connects via socket, (2) performs the full authentication dialogue (both registration and login), and (3) sends a sequence of commands (list, cd, download, upload, delete, exit) over the text protocol. Verify on the client side the exact messages received (e.g., login success, error messages, FILE_CONTENT prefix) and on the server side that the expected files are created/removed and that the connection is closed correctly. You can keep these tests small (e.g., JUnit test that runs server in a background thread) but they should cover at least one successful and one failing session to validate the protocol as implemented in ClientHandler and Client together.

TST-002 — MEDIUM [82%] — Page 4 Multi-user / concurrency behavior, which is a stated objective of the architecture, is not tested. All tests run components in a single-threaded context and do not verify that multiple ClientHandler threads can operate correctly and independently on shared resources (credentials file, server_files directory, observers).

> *Supporto multi-utente concorrente : Il server 'e in grado di gestire pi'u client contemporaneamente. Utenti diversi possono connettersi da postazioni diverse e usufruire del servizio in parallelo, senza interferire gli uni con gli altri. ... Ad ogni nuova connessione il server crea un thread ClientHandler dedicato, abilitando il servizio concorrente di pi'u client.*

Recommendation: Add at least one focused concurrency test that starts two or more ClientHandler instances in parallel (using threads in a JUnit test) sharing the same Server and rootDir. For example: (1) create two ClientHandler test doubles that both perform list and download on different files at the same time, and assert that each receives the correct responses without interference; (2) for admin operations, run two UploadCommand or DeleteCommand instances concurrently on different files and verify that both complete successfully and that the final state of server_files is consistent. You do not need heavy load tests; a small number of parallel threads is enough to demonstrate that the multi-threaded design works as intended and to support the architectural claim of "Supporto multi-utente concorrente".

TST-003 — LOW [71%] — Page 5 The client-side behavior for saving downloaded files and handling the FILE_CONTENT protocol is not directly tested. FileOperationsBlackBoxTest verifies that DownloadCommand sends a FILE_CONTENT message, but there is no test that exercises the Client logic that decodes Base64 and writes the file to disk.

> *Il trasferimento dei file ‘e 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. ... FileOperationsBlackBoxTest Obiettivo: Testare direttamente le funzionalit‘a 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.*

Recommendation: Add a small unit or integration test for the Client class (or for the listener component that processes server messages) that feeds it a simulated FILE_CONTENT message, e.g. "FILE_CONTENT:test.txt:<base64>" over a fake input stream, and asserts that: (1) the file test.txt is created in the expected local directory, and (2) its content matches the original bytes. You can implement this by extracting the message-handling logic into a separate method (e.g., handleServerMessage(String msg)) that is easy to call from a JUnit test. This will close the loop on the download use case by testing both server-side sending and client-side saving.

7 Priority Recommendations

The following actions are considered priority:

1. **ISS-003** (p. 4): Clarify and align the business rule about user registration across the document.
2. **TST-001** (p. 29): Add at least one integration / end-to-end test that starts a real Server (or a ServerSocket on a random free port) and a Client (or a lightweight clie...

8 Traceability Matrix

Of 16 traced use cases: 12 fully covered, 0 without design, 4 without test.

ID	Use Case	Design	Test	Gap
UC-1	Registration	✓	✓	—
UC-2	Login	✓	✓	—
UC-3	Choose download directory	✓	×	No automated test explicitly verifies the client-side selection of the local download directory or its integration with ...
UC-4	Cd	✓	✓	There is no test that executes CdCommand to verify directory change behavior and observer logging.
UC-5	List	✓	✓	There is no test that runs ListCommand.execute() to assert the actual directory listing output and logging.
UC-6	Download	✓	✓	—
UC-7	Upload	✓	✓	—
UC-8	Delete	✓	✓	—
UC-9	Exit	✓	×	No automated test covers the exit command behavior, including server-side session termination and resource cleanup.

ID	Use Case	Design	Test	Gap
UC-10	Commands (composite use case including Cd, List, Download, Upload, Delete, Exit)	✓	✓	There is no integrated test that drives the full ClientHandler command loop with a sequence of mixed commands as in real...
UC-11	Visualizzazione dei file disponibili	✓	✓	There is no test that asserts the exact content/format of the directory listing returned by ListCommand.
UC-12	Autenticazione degli utenti	✓	✓	—
UC-13	Supporto multi-utente concorrente	✓	×	No concurrency or load tests validate simultaneous connections, thread safety, or isolation between multiple ClientHandl...
UC-14	Chiusura della connessione (logout/uscita sicura)	✓	×	There is no automated test verifying that logout/exit correctly releases server resources and terminates client threads.
UC-15	Gestione ruoli User/Admin e autorizzazioni comandi	✓	✓	—
UC-16	Logging e monitoraggio delle azioni (Observer pattern)	✓	✓	Observer behavior is only partially tested (download via Server.

9 Terminological Consistency

Found **10** terminological inconsistencies (4 major, 6 minor).

Group	Variants found	Severity	Suggestion
End-user actor naming	ExpectedUser, NormalUser, User, utenti finali, utente	MAJOR	Use a single, consistent term for the non-admin actor; align the UML diagram, use case templates, and textual description on either "User" or "NormalUser" (prefer "User" to match class names and most of the text).
Admin role naming	Admin, admin, amministratori	MINOR	Use a single capitalization style for the admin role across the document; prefer "Admin" to match the class name and actor name.

Group	Variants found	Severity	Suggestion
Entry screen naming	pagina principale, Home Page, home, schermata di accesso, schermata di registrazione	MAJOR	Use one consistent term for the login/registration starting point; if the system is console-based, avoid "pagina" and prefer a neutral term like "schermata" or "interfaccia" consistently.
Download destination naming	cartella dove scaricare i file, directory scelta, Choose download Directory, download directory, cartella locale	MINOR	Standardize the naming of the download destination folder; use a single term such as "cartella di download" or "download directory" consistently.
Server file storage naming	cartella server files, directory dei file scaricabili, archivio dei file, server_files, archivio del server	MINOR	Use a single term for the server-side file area; prefer "directory del server" or "server_files" and keep it consistent in text and tests.
Dashboard naming	Dashboard Page, Dashboard, dashboard Page, Dashboard dell'User, Dashboard dell'Admin	MINOR	Use a single, consistent label for the user dashboard; prefer "Dashboard" or "Dashboard Page" and apply it uniformly for both User and Admin.
Authenticator class naming	UserAuthenticator, UseAuthenticator, classe UserAuthenticator, Componente 'e responsabile della gestione delle credenziali utente	MAJOR	Use one consistent name for the authentication component; align the class diagram and text on "UserAuthenticator" (which matches the implementation and listings).
Observer notification method naming	notifyObservers, notifyObserver, notifyDownload, metodo notify(), notifica l'avvenuta registrazione	MAJOR	Use a single method name for observer notification; either consistently use "notifyObservers" or "notifyDownload" in both text and code descriptions.
Command abstraction naming	interfaccia Command, Command (interface), classi comando, oggetto comando, classe comando	MINOR	Standardize the naming of the command abstraction; use "interfaccia Command" or "Command interface" consistently and avoid mixing with "classi comando" when referring to the same concept.
Textual command naming	comandi testuali, comando, command, Download command, Cd command, exit command	MINOR	Use a single term for the textual protocol/commands; choose either "comandi testuali" or "command" and apply it consistently.