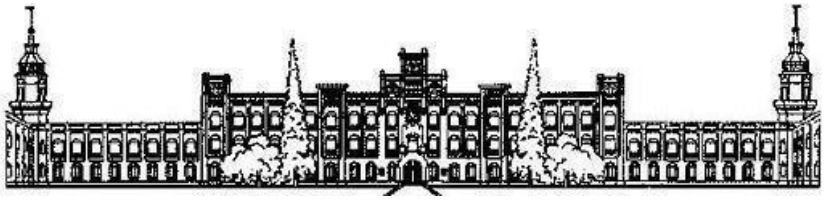

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №9

з дисципліни «Технології розроблення програмного забезпечення»

Тема: «Взаємодія компонентів системи.»

Виконав:
студент групи ІА-32
Діденко Я.О

Перевірив:
Мягкий М.Ю

Київ 2025

Тема: Взаємодія компонентів системи.
Мета: Вивчити види взаємодії додатків (Client-Server, Peer-to-Peer, Service-oriented Architecture), та реалізувати в проєктованій системі одну із архітектур.

18. **Shell (total commander)** (state, prototype, factory method, template method, interpreter, client-server)

Оболонка повинна вміти виконувати основні дії в системі – перегляд файлів папок в файлової системі, перемикання між дисками, копіювання, видалення, переміщення об'єктів, пошук.

Теоретичні Відомості

Для розробки розподілених систем існує кілька фундаментальних архітектурних підходів. Розглянемо основні види, що вивчались у рамках лабораторної роботи.

1. Архітектура «Клієнт-Сервер» (Client-Server)

Це централізована модель, що розподіляє ролі між учасниками мережі. **Сервер** є постачальником ресурсів або послуг і пасивно очікує на запити. **Клієнт** є споживачем, який ініціює з'єднання та відправляє запити до сервера для отримання даних або виконання операцій. Вся взаємодія відбувається через сервер, а прямий зв'язок між клієнтами відсутній. Залежно від обсягу логіки на стороні клієнта розрізняють "тонких" (більшість логіки на сервері) та "товстих" (більшість логіки на клієнті) клієнтів.

2. Однорангова архітектура (Peer-to-Peer, P2P)

Це децентралізована модель, в якій відсутній виділений центральний сервер. Кожен вузол мережі, що називається **піром** (peer), є рівноправним і може одночасно виконувати функції як клієнта, так і сервера: запитувати ресурси в інших вузлів і надавати власні. Така структура забезпечує високу стійкість до відмов, оскільки вихід з ладу одного вузла не впливає на працездатність решти мережі. Прикладами є файлообмінні мережі (BitTorrent) та криптовалюти.

3. Сервіс-орієнтована архітектура (Service-Oriented Architecture, SOA)

Це підхід до розробки, при якому функціональність системи будується з набору незалежних, слабо пов'язаних **сервісів**. Кожен сервіс є автономним компонентом, що реалізує конкретну бізнес-функцію і доступний по мережі через стандартизований інтерфейс (зазвичай на базі веб-технологій, як-от REST або SOAP). Сервіси можуть бути розроблені та розгорнуті незалежно один від одного, що забезпечує гнучкість та можливість повторного використання функціоналу в різних додатках.

Клієнт-серверна архітектура.

Клієнт-серверні додатки являють собою найпростіший варіант розподілених додатків, де виділяється два види додатків: клієнти (представляють додаток користувачеві) і сервери (використовується для зберігання і обробки даних). Розрізняють тонкі клієнти і товсті клієнти. Тонкий клієнт – клієнт, який повністю всі операції (або більшість, пов'язаних з логікою роботи програми) передає для обробки на сервер, а сам зберігає лише візуальне уявлення одержуваних від сервера відповідей. Грубо кажучи, тонкий клієнт – набір форм відображення і канал зв'язку з сервером. Прикладом тонкого клієнта є класичні Web-застосунки. У такому варіанті використання майже все навантаження лягає на сервер або групу серверів. Перевагою таких моделей є простота розгортання, тому що оновлювати потрібно лише сервери і в результаті клієнти з наступними запитами автоматично будуть працювати з оновленою системою.

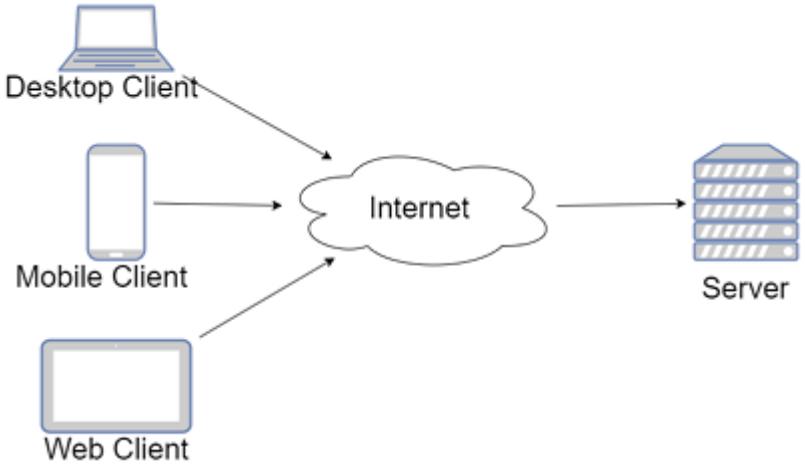

рис 1. Структура клієнт-серверної архітектури

Товстий клієнт – антипод тонкого клієнта, більшість логіки обробки даних містить на стороні клієнта. Це сильно розвантажує сервер. Сервер в таких 99 випадках зазвичай працює лише як точка доступу до деякого іншого ресурсу (наприклад, бази даних) або сполучна ланка з іншими клієнтськими комп'ютерами. Перевагою такого підходу є менші вимоги до серверної частини. Також перевагою, при певному підході до реалізації, є можливість працювати клієнтам без тимчасового доступу до серверу. Прикладом товстого клієнта можна назвати мобільні застосунки, або

десктоп застосунки. Наприклад, Evernote, Viber, MS Outlook, комп’ютерні антивіруси, ігри, що потребують інсталяції (The Sims, GTA, …) та інші. Проміжним варіантом можна назвати SPA (Single Page Application) – це товсті Web-клієнти, які при старті кожен раз завантажуються з сервера, а надалі працюють з сервером через web-API. З одного боку більшу частину логіки вони відпрацьовують на клієнтський стороні, за рахунок чого зменшується серверне навантаження. Також оновлення простіше ніж для товстих клієнтів. Але такі застосунки не працюють, якщо сервер не доступний.
Клієнт-серверна взаємодія, як правило, організовується за допомогою 3-х рівневої структури: клієнтська частина, загальна частина, серверна частина.
Оскільки велика частина даних загальна (класи, використовувані системою), їх прийнято виносити в загальну частину (middleware) системи.
Клієнтська частина містить візуальне відображення і логіку обробки дії користувача; код для встановлення сеансу зв'язку з сервером і виконання відповідних викликів.
Серверна частина містить основну логіку роботи програми (бізнес-логіку) або ту її частину, яка відповідає зберіганню або обміну даними між клієнтом і сервером або клієнтами.

Хід роботи

1. Загальний опис виконаної роботи

Основна мета полягала в розробці розподіленого файлового менеджера, що функціонує за **клієнт-серверною архітектурою**. Замість монолітного додатку, вся логіка взаємодії з файловою системою була винесена на серверну сторону, тоді як клієнтська частина відповідає виключно за графічний інтерфейс та взаємодію з користувачем.

Такий підхід дозволяє чітко відокремити логіку представлення (UI) від бізнес-логіки (файлові операції). Сервер виступає як єдина точка доступу до ресурсів, що забезпечує централізоване управління та безпеку, а клієнт надає користувачу можливість віддалено керувати файлами на сервері. Взаємодія між компонентами відбувається по мережі за допомогою TCP-сокетів, а дані передаються у форматі JSON.

Програмна система складається з наступних модулів та компонентів:

- **Модуль shared (Загальна бібліотека)**
  Це базовий "контракт" або протокол, який визначає структуру даних для обміну між клієнтом і сервером. Він використовується як залежність в обох частинах системи. Містить:
  - **CommandType:** Перелік (enum), що визначає всі можливі операції (GET_DRIVES, COPY, DELETE тощо).
  - **Request:** Клас-DTO для інкапсуляції запиту від клієнта. Містить тип команди та необхідні параметри (наприклад, шляхи до файлів).
  - **Response:** Клас-DTO для відповіді від сервера. Містить статус виконання операції (успіх/помилка), повідомлення про помилку та дані (наприклад, список файлів).
  - **FileSystemEntry:** Клас-DTO для опису одного елемента файлової системи.
- **Модуль server (Серверна частина)**
  Це консольний додаток, який виконує всю роботу з файлами та каталогами.
  - **Клас FileServer:** Головний клас, що запускає сервер. Він створює ServerSocket для прослуховування порту та ExecutorService (пул потоків) для обробки кількох клієнтських підключень одночасно.
  - **Клас ClientHandler:** Обробник, що виконується в окремому потоці для кожного підключеного клієнта. Він зчитує JSON-рядок запиту, десеріалізує його в об'єкт Request, викликає відповідний метод для роботи з файловою системою, формує об'єкт Response, серіалізує його в JSON та відправляє назад клієнту.
- **Модуль client (Клієнтська частина)**
  Це десктопний додаток на Java Swing, що надає користувацький інтерфейс.
  - **Клас NetworkClient:** Допоміжний клас, що повністю інкапсулює логіку мережевої взаємодії. Його метод send() приймає об'єкт Request, встановлює з'єднання з сервером, відправляє запит та повертає отриманий об'єкт Response.
  - **Клас MainFrame:** Головне вікно програми. Воно містить усі графічні елементи (списки файлів, кнопки, поля) та обробники

подій. При діях користувача (клік по кнопці, подвійний клік по папці) MainFrame створює відповідний об'єкт Request, викликає метод net.send(), а потім оновлює інтерфейс на основі даних з отриманого Response. Клієнт не виконує жодних прямих операцій з файлами.

2. Опис класів програмної системи.

2.1. Модуль shared (Загальні класи)

Цей модуль є фундаментом для взаємодії між клієнтом та сервером. Він містить класи-DTO (Data Transfer Objects), що описують структуру даних, якими обмінюються компоненти.

Клас Request.java

- **Опис:** Клас Request інкапсулює будь-який запит, що надсилається від клієнта до сервера. Він є уніфікованим контейнером, що містить тип команди та необхідні для її виконання дані.
- **Характеристики:**
    - Є POJO (Plain Old Java Object) класом, що реалізує інтерфейс Serializable.
    - Містить поле command типу CommandType для ідентифікації операції.
    - Містить поля path1 та path2 для передачі шляхів до файлів або каталогів. Наприклад, для команди COPY, path1 буде джерелом, а path2 - призначенням.

```java
package ua.edu.university.shared;

import java.io.Serializable;

public class Request implements Serializable {  11 usages
    private CommandType command;  2 usages
    private String path1;  2 usages
    private String path2;  2 usages

    public Request() {}  no usages

    public Request(CommandType command, String path1, String path2) {  5 usages
        this.command = command;
        this.path1 = path1;
        this.path2 = path2;
    }

    public CommandType getCommand() { return command; }  2 usages
    public String getPath1() { return path1; }  4 usages
    public String getPath2() { return path2; }  1 usage
}
```

рис. 2.1 - клас Request.java

Клас Response.java

- **Опис:** Клас Response представляє уніфіковану відповідь від сервера клієнту. Він повідомляє про результат виконання операції та передає запитувані дані.
- **Характеристики:**
    - Є POJO класом, що реалізує інтерфейс Serializable.
    - Містить булеве поле success для позначення успішності операції.
    - Містить поле errorMessage для передачі тексту помилки у разі невдачі.
    - Містить поля drives (список дисків) та entries (список файлів/каталогів), які заповнюються в залежності від типу запиту.

```java
package ua.edu.university.shared;

import java.io.Serializable;
import java.util.List;

public class Response implements Serializable {  15 usages
    private boolean success;  2 usages
    private String errorMessage;  2 usages
    private List<String> drives;  2 usages
    private List<FileSystemEntry> entries;  2 usages

    public boolean isSuccess() { return success; }  5 usages
    public String getErrorMessage() { return errorMessage; }  5 usages
    public List<String> getDrives() { return drives; }  2 usages
    public List<FileSystemEntry> getEntries() { return entries; }  3 usages

    public void setSuccess(boolean success) { this.success = success; }  3 usages
    public void setErrorMessage(String errorMessage) { this.errorMessage = errorMessage; }  2 usages
    public void setDrives(List<String> drives) { this.drives = drives; }  1 usage
    public void setEntries(List<FileSystemEntry> entries) { this.entries = entries; }  1 usage
}
```

рис. 2.2 - клас Responce.java

Інші класи модуля:

- **CommandType.java:** Перелік (enum), що містить константи для всіх можливих команд. Це усуває необхідність використовувати "магічні рядки" та робить код більш надійним.
- **FileSystemEntry.java:** Клас-DTO, що описує один елемент файлової системи: його ім'я, розмір, тип (файл/каталог) та дату останньої зміни.

2.2. Модуль server (Серверна частина)

Цей модуль містить логіку для прийому запитів та виконання файлових операцій.

Клас FileServer.java

- **Опис:** Клас FileServer є точкою входу та головним керуючим компонентом серверної частини. Він відповідає за запуск сервера, прослуховування мережевого порту та делегування обробки клієнтських підключень.
- **Характеристики:**

- Містить метод main(), що ініціалізує ServerSocket на фіксованому порту (8888).
- Використовує ExecutorService для створення пулу потоків. Це дозволяє обробляти запити від кількох клієнтів одночасно, не блокуючи один одного.
- У нескінченному циклі очікує нових підключень (serverSocket.accept()) і для кожного створює новий екземпляр ClientHandler, передаючи його на виконання в пул потоків.

```
package ua.edu.university.server;

import ...

public class FileServer {

    private static final int PORT = 8888;  1 usage

    public static void main(String[] args) {
        ExecutorService pool = Executors.newCachedThreadPool();
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("SERVER STARTED");
            while (true) {
                Socket client = serverSocket.accept();
                pool.execute(new ClientHandler(client));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

рис. 2.3 - клас FileServer.java

Клас ClientHandler.java

- **Опис:** Клас ClientHandler реалізує логіку обробки одного конкретного клієнтського запиту. Екземпляр цього класу створюється для кожного нового підключення і виконується в окремому потоці.

- **Характеристики:**
  - Реалізує інтерфейс Runnable.
  - У методі run() зчитує вхідний потік даних від клієнта, десеріалізує JSON-рядок в об'єкт Request.
  - Викликає приватний метод handle(), передаючи йому об'єкт Request.
  - Результат роботи (об'єкт Response) серіалізує в JSON і відправляє назад клієнту.
  - Метод handle() містить switch-case конструкцію, яка аналізує command з об'єкта Request і викликає відповідний метод для роботи з файловою системою (отримання списку дисків, копіювання, видалення тощо).

```java
package ua.edu.university.server;

import ...

public class ClientHandler implements Runnable {  1 usage
    private final Socket socket;  4 usages
    private final Gson gson = new Gson();  2 usages

    public ClientHandler(Socket socket) {  1 usage
        this.socket = socket;
    }

    @Override
    public void run() {
        try (
                BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
                PrintWriter out = new PrintWriter(socket.getOutputStream(),  autoFlush: true)
        ) {
            String json = in.readLine();
            if (json == null) {
                return;
            }

            Request req = gson.fromJson(json, Request.class);
            Response res = handle(req);
            out.println(gson.toJson(res));

        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try { socket.close(); } catch (IOException ignored) {}
        }
```

```
47 @    private Response handle(Request req) {  1 usage
48          Response res = new Response();
49
50          try {
51              switch (req.getCommand()) {
52
53                  case GET_DRIVES: {
54                      List<String> drives = Arrays.stream(File.listRoots()) Stream<File>
55                              .map(File::getAbsolutePath) Stream<String>
56                              .collect(Collectors.toList());
57                      res.setDrives(drives);
58                      break;
59                  }
60
61                  case GET_DIRECTORY_CONTENT: {
62                      Path path = Paths.get(req.getPath1());
63                      List<FileSystemEntry> entries = new ArrayList<>();
64
65                      if (path.getParent() != null) {
66                          entries.add(new FileSystemEntry( name: "..",  size: 0,  isDirectory: true,  lastModified: 0));
67                      }
68
69                      try (DirectoryStream<Path> dir = Files.newDirectoryStream(path)) {
70                          for (Path p : dir) {
71                              File f = p.toFile();
72                              entries.add(new FileSystemEntry(
73                                      f.getName(),
74                                      f.length(),
75                                      f.isDirectory(),
76                                      f.lastModified()
77                              ));
```

```
84                  case CREATE_DIRECTORY: {
85                      Path dirPath = Paths.get(req.getPath1());
86                      Files.createDirectories(dirPath);
87                      break;
88                  }
89
90                  case COPY: {
91                      Path source = Paths.get(req.getPath1());
92                      Path target = Paths.get(req.getPath2());
93                      copyRecursively(source, target);
94                      break;
95                  }
96
97                  case DELETE: {
98                      Path toDelete = Paths.get(req.getPath1());
99                      deleteRecursively(toDelete);
100                     break;
101                 }
102
103                 default:
104                     throw new UnsupportedOperationException("Unknown command: " + req.getCommand());
105             }
106
107             res.setSuccess(true);
108         } catch (Exception e) {
109             res.setSuccess(false);
110             res.setErrorMessage(e.getClass().getSimpleName() + ": " + e.getMessage());
111         }
112
113         return res;
```

```
116     private void copyRecursively(Path source, Path target) throws IOException {  1 usage
117         if (!Files.exists(source)) {
118             throw new FileNotFoundException("Source not found: " + source);
119         }
120
121         if (Files.isDirectory(source)) {
122             Files.createDirectories(target);
123
124             try (Stream<Path> stream = Files.walk(source)) {
125                 stream.forEach( Path p -> {
126                     Path relative = source.relativize(p);
127                     Path dest = target.resolve(relative);
128                     try {
129                         if (Files.isDirectory(p)) {
130                             Files.createDirectories(dest);
131                         } else {
132                             Files.copy(p, dest, StandardCopyOption.REPLACE_EXISTING);
133                         }
134                     } catch (IOException ex) {
135                         throw new UncheckedIOException(ex);
136                     }
137                 });
138             } catch (UncheckedIOException ex) {
139                 throw ex.getCause();
140             }
141
142         } else {
143             Files.createDirectories(target.getParent());
144             Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);
145         }
146     }
```

```
148     private void deleteRecursively(Path path) throws IOException {  1 usage
149         if (!Files.exists(path)) {
150             return;
151         }
152
153         Files.walkFileTree(path, new SimpleFileVisitor<Path>() {
154             @Override
155             public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
156                 Files.delete(file);
157                 return FileVisitResult.CONTINUE;
158             }
159
160             @Override  3 usages
161             public FileVisitResult postVisitDirectory(Path dir, IOException exc) throws IOException {
162                 Files.delete(dir);
163                 return FileVisitResult.CONTINUE;
164             }
165         });
166     }
167 }
```

рис. 2.4 - клас ClientHandler.java

2.3. Модуль client (Клієнтська частина)

Цей модуль реалізує графічний інтерфейс користувача та логіку взаємодії з сервером.

Клас NetworkClient.java

- **Опис:** Клас NetworkClient інкапсулює всю низькорівневу логіку мережевої комунікації. Він надає простий та зручний API для відправки запитів на сервер, ховаючи деталі роботи з сокетами та серіалізацією.
- **Характеристики:**
    - Має метод send(Request req), який є єдиною точкою взаємодії з сервером для решти клієнтського коду.
    - Всередині методу send() створюється новий Socket для кожного запиту, що забезпечує простоту реалізації без управління довготривалими з'єднаннями.
    - Використовує бібліотеку Gson для перетворення об'єктів Request в JSON перед відправкою та Response з JSON після отримання.
    - Обробляє IOException та інші помилки з'єднання, повертаючи об'єкт Response з відповідним повідомленням про помилку.

```java
package ua.edu.university.client;

import ...

public class NetworkClient {  2 usages
    private final String ip;  2 usages
    private final int port;  2 usages
    private final Gson gson = new Gson();  2 usages

    public NetworkClient(String ip, int port) {  1 usage
        this.ip = ip;
        this.port = port;
    }

    public Response send(Request req) {  5 usages
        try (
                Socket socket = new Socket(ip, port);
                PrintWriter out = new PrintWriter(socket.getOutputStream(),  autoFlush: true);
                BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()))
        ) {
            out.println(gson.toJson(req));
            return gson.fromJson(in.readLine(), Response.class);

        } catch (Exception e) {
            Response r = new Response();
            r.setSuccess(false);
            r.setErrorMessage(e.getMessage());
            return r;
        }
    }
}
```

рис. 2.5 - клас NetworkClient.java

Клас MainFrame.java

- **Опис:** Клас MainFrame є головним вікном програми і виступає в ролі "Клієнта" у найвищому розумінні. Він відповідає за побудову UI, обробку дій користувача та відображення даних, отриманих від сервера.
- **Характеристики:**
    - Створює та управляє єдиним екземпляром NetworkClient протягом усього сеансу роботи.
    - Містить компоненти Swing (JList, JButton, JComboBox) для візуалізації файлової системи.
    - Має обробники подій (ActionListener, MouseListener), які на дії користувача (наприклад, клік на кнопку "Delete") створюють відповідний об'єкт Request.
    - Використовує екземпляр NetworkClient для відправки запиту на сервер.

- Після отримання об'єкта Response аналізує його та оновлює стан UI: або відображає список файлів, або показує діалогове вікно з повідомленням про помилку.

```
16  public class MainFrame extends JFrame {
17
18      private final DefaultListModel<FileSystemEntry> model = new DefaultListModel<>();  3 usages
19      private final JList<FileSystemEntry> list = new JList<>(model);  6 usages
20      private final JComboBox<String> driveBox = new JComboBox<>();  5 usages
21      private final JTextField pathField = new JTextField();  3 usages
22
23      private final JButton btnNewFolder = new JButton( text: "New folder");  2 usages
24      private final JButton btnDelete = new JButton( text: "Delete");  2 usages
25      private final JButton btnCopy = new JButton( text: "Copy to...");  2 usages
26
27      private final NetworkClient net = new NetworkClient( ip: "127.0.0.1",  port: 8888);  5 usages
28
29      private String currentPath = "";  9 usages
30
31      public MainFrame() {  1 usage
32          setTitle("Client - Total Commander");
33          setSize( width: 900,  height: 600);
34          setDefaultCloseOperation(EXIT_ON_CLOSE);
35          setLocationRelativeTo(null);
36
37          setLayout(new BorderLayout());
38
39          JPanel top = new JPanel(new BorderLayout( hgap: 5,  vgap: 5));
40          top.setBorder(BorderFactory.createEmptyBorder( top: 5,  left: 5,  bottom: 5,  right: 5));
41
42          top.add(driveBox, BorderLayout.WEST);
43
44          pathField.setEditable(false);
45          top.add(pathField, BorderLayout.CENTER);
```

```
        JPanel buttonsPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT,  hgap: 5,  vgap: 0));
        buttonsPanel.add(btnNewFolder);
        buttonsPanel.add(btnCopy);
        buttonsPanel.add(btnDelete);
        top.add(buttonsPanel, BorderLayout.EAST);

        add(top, BorderLayout.NORTH);

        list.setFont(new Font( name: "Monospaced", Font.PLAIN,  size: 14));
        add(new JScrollPane(list), BorderLayout.CENTER);

        driveBox.addActionListener( ActionEvent e -> {
            String d = (String) driveBox.getSelectedItem();
            if (d != null) {
                loadDir(d);
            }
        });

        list.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseClicked(MouseEvent e) {
                if (e.getClickCount() == 2) {
                    FileSystemEntry sel = list.getSelectedValue();
                    if (sel == null) return;
                    if (sel.isDirectory()) {
                        Path p = Paths.get(currentPath, sel.getName()).normalize();
                        loadDir(p.toString());
                    }
                }
            }
```

```
    btnNewFolder.addActionListener( ActionEvent e -> {
        if (currentPath == null || currentPath.isEmpty()) return;

        String name = JOptionPane.showInputDialog(
                parentComponent: MainFrame.this,
                message: "Folder name:",
                title: "Create folder",
                JOptionPane.PLAIN_MESSAGE
        );
        if (name == null) return;
        name = name.trim();
        if (name.isEmpty()) return;

        String fullPath = Paths.get(currentPath, name).toString();
        Response r = net.send(new Request(CommandType.CREATE_DIRECTORY, fullPath,  path2: null));
        if (!r.isSuccess()) {
            JOptionPane.showMessageDialog( parentComponent: this, r.getErrorMessage(),  title: "Error", JOptionPane.ERROR_MESSAGE);
            return;
        }
        loadDir(currentPath);
    });
```

рис. 2.6 - клас MainFrame.java

Результати виконання коду

Надані скріншоти демонструють послідовне тестування функціоналу розробленого файлового менеджера. Тестування охоплює навігацію, створення, копіювання та видалення каталогів, підтверджуючи коректність взаємодії між клієнтом та сервером.

Етап 1: Навігація та створення каталогу в корені диска

- **Перегляд вмісту диска (рис. 3.1):**
  Робота починається з перегляду кореневого каталогу диска C:\. Клієнтський додаток відправляє серверу запит на отримання списку файлів та каталогів для цього шляху і відображає отриманий результат.

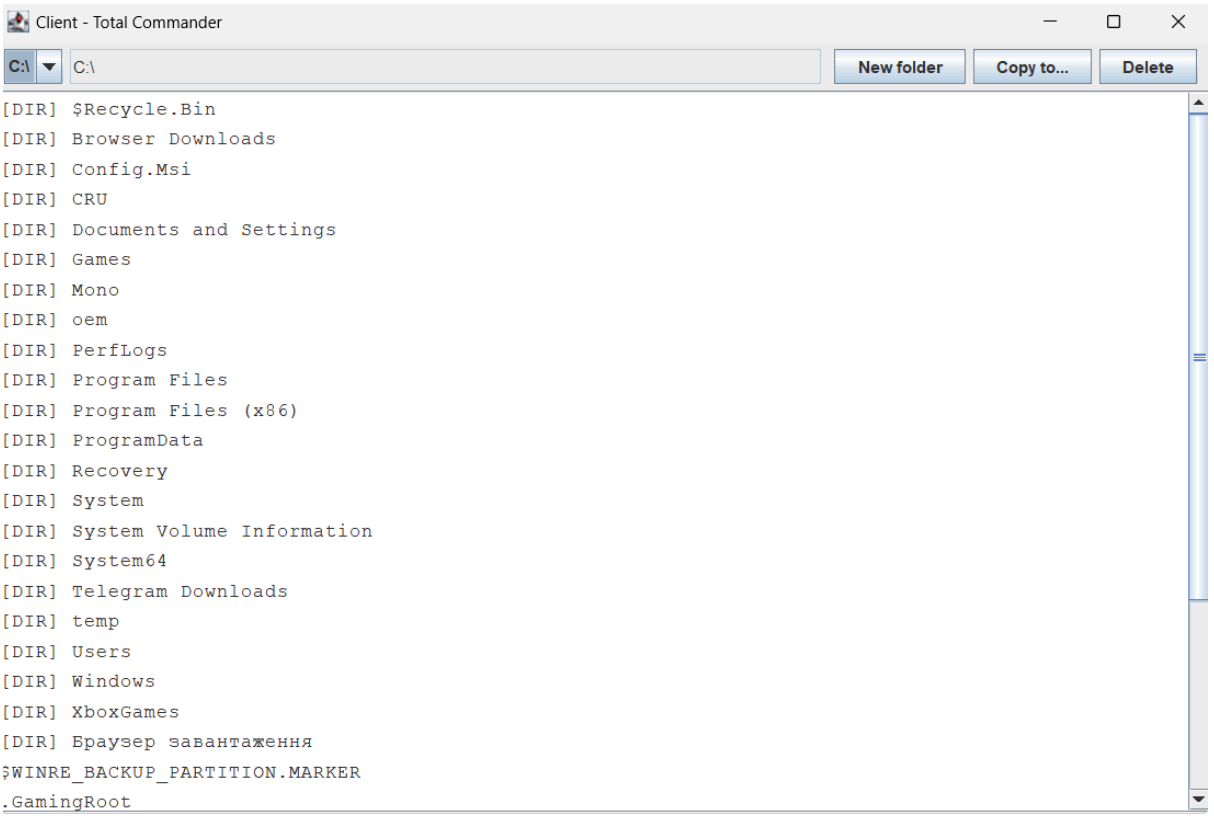

Рис. 3.1 - Перегляд кореневого каталогу

- **Процес створення нового каталогу (рис. 3.2):**
  Користувач ініціює створення нового каталогу, натиснувши кнопку "New folder". Відкривається діалогове вікно, в якому вводиться ім'я NewFolder.

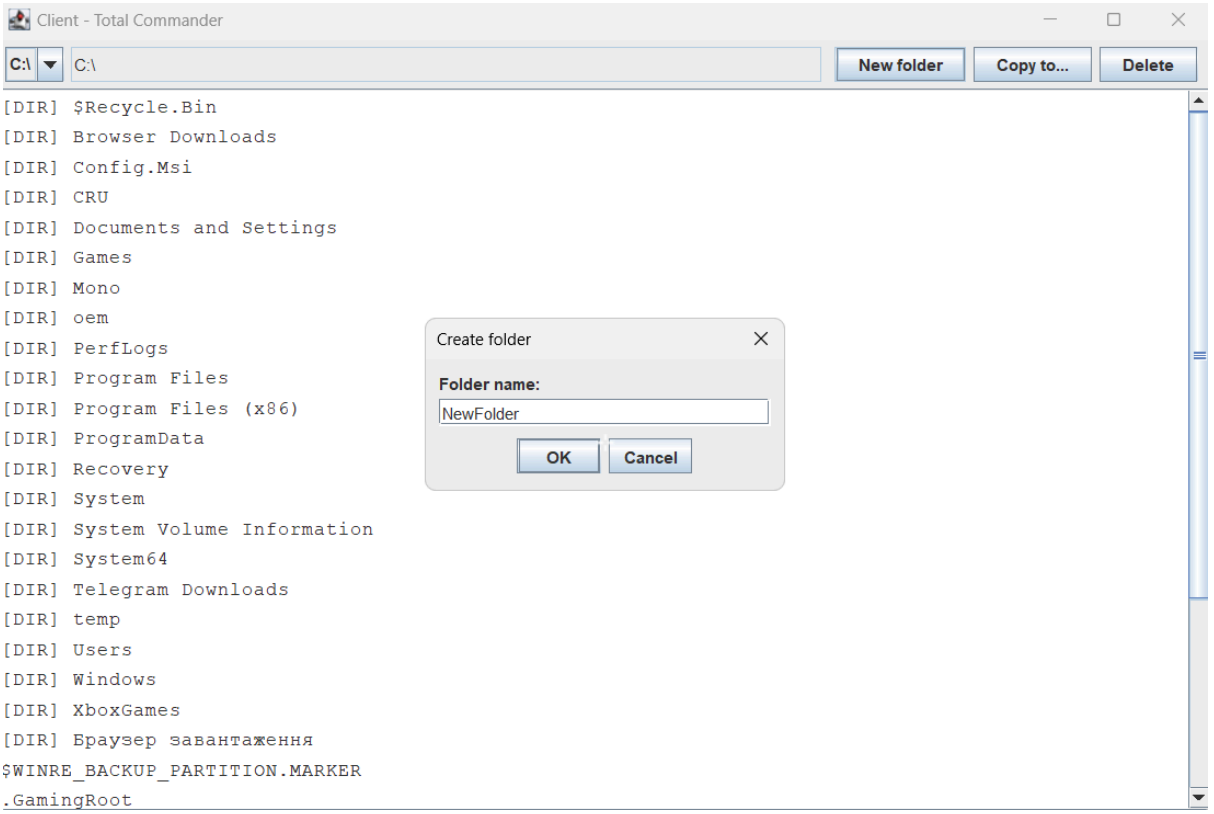

Рис. 3.2 - Введення імені для нового каталогу

- **Результат створення (рис. 3.3):**
  Після підтвердження клієнт відправляє на сервер команду CREATE_DIRECTORY зі шляхом C:\NewFolder. Сервер створює каталог, і клієнт, оновивши список, відображає щойно створений каталог NewFolder у корені диска.

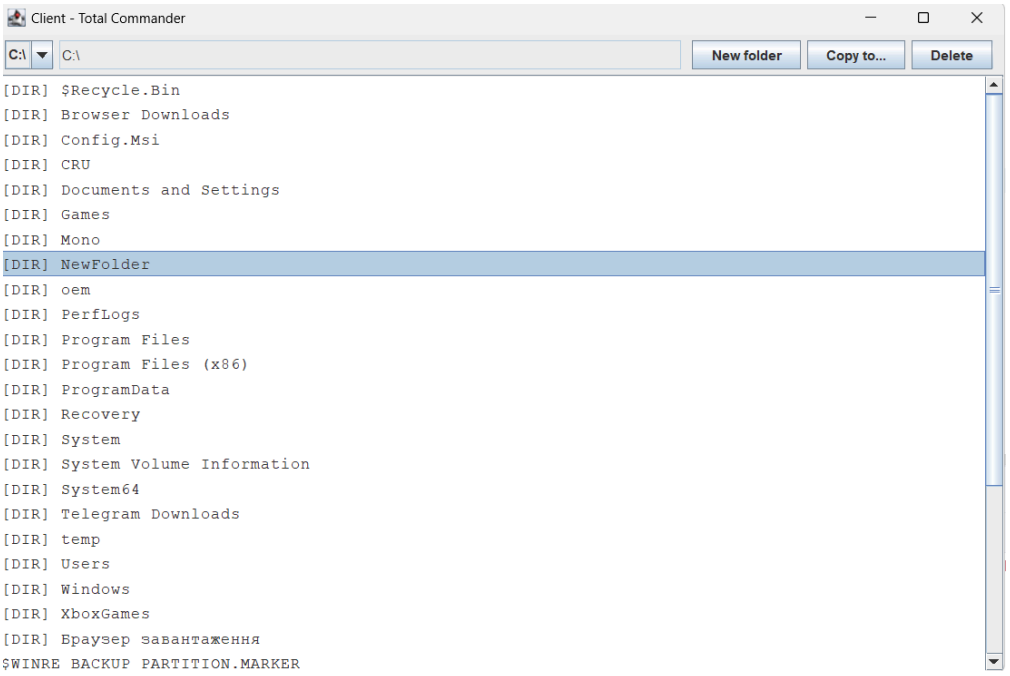

Рис. 3.3 - Каталог NewFolder успішно створений

Етап 2: Тестування операції копіювання

- **Ініціація копіювання (рис. 3.4):**
  Користувач вибирає створений каталог C:\NewFolder і натискає кнопку "Copy to...". Відкривається діалогове вікно JFileChooser, де в якості цільового каталогу обирається C:\Games.

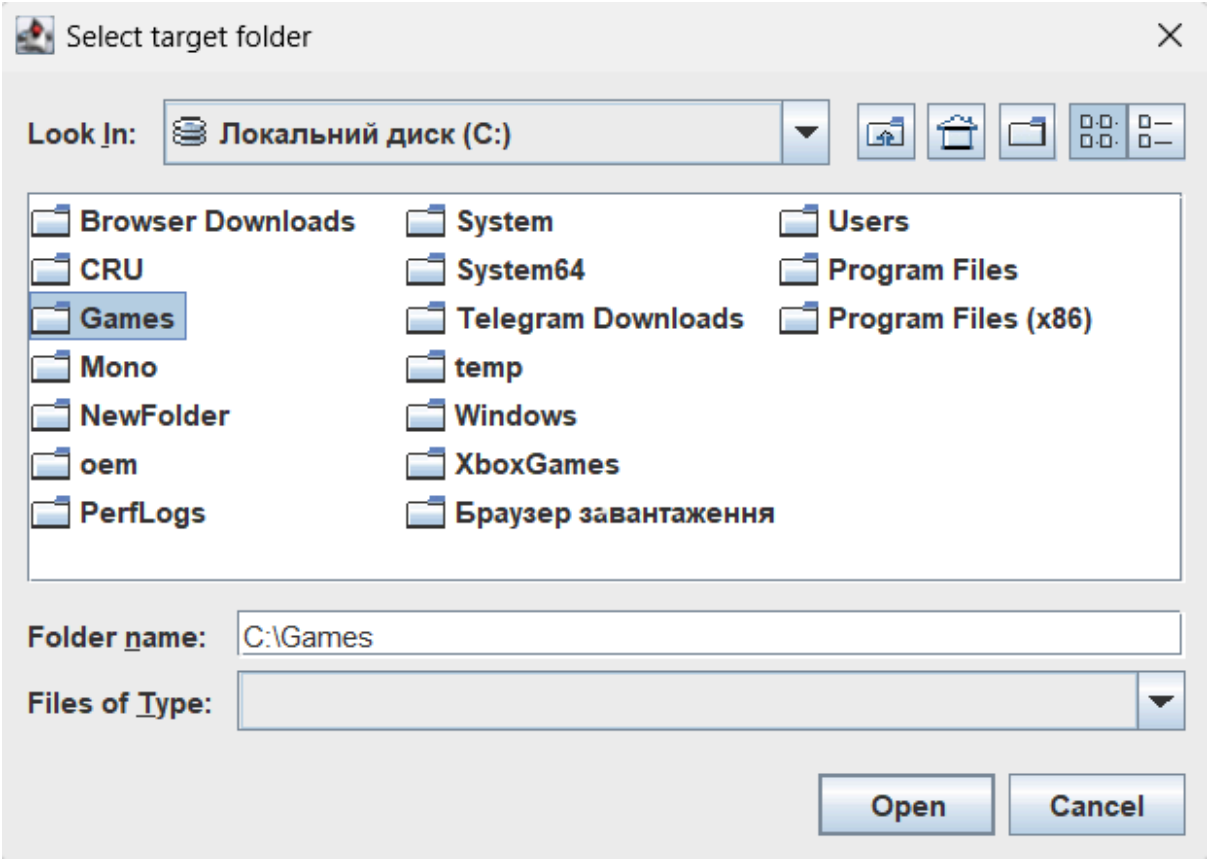

Рис. 3.4 - Вибір цільового каталогу C:\Games для копіювання

- **Результат копіювання (рис. 3.5):**
  Клієнт відправляє на сервер команду COPY з вихідним шляхом C:\NewFolder та цільовим C:\Games\NewFolder. Сервер виконує операцію. Після цього користувач переходить у каталог C:\Games, і оновлений список показує, що копія каталогу NewFolder була успішно створена всередині C:\Games.

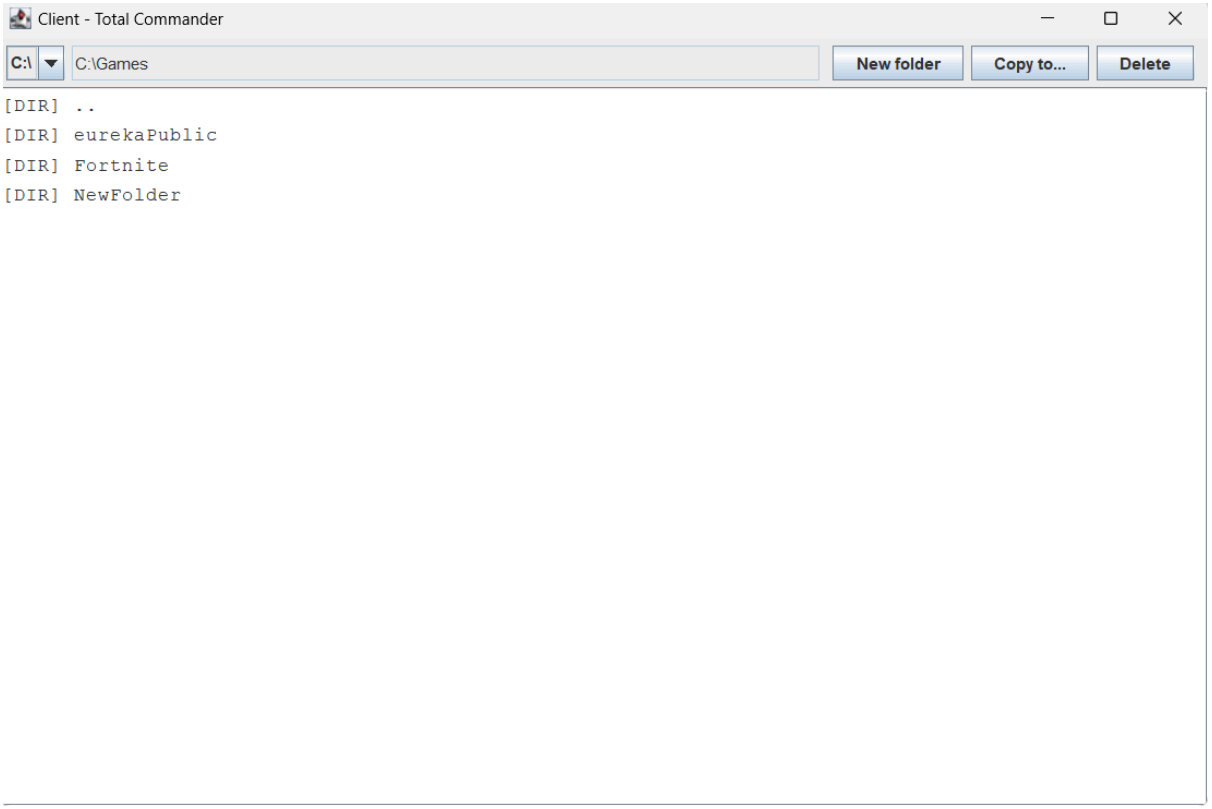

Рис. 3.5 - Каталог NewFolder скопійовано до C:\Games

Етап 3: Тестування операції видалення

- **Підтвердження видалення (рис. 3.6):**
  Перебуваючи в каталозі C:\Games, користувач вибирає скопійований каталог NewFolder і натискає "Delete". Програма запитує підтвердження для уникнення випадкових дій.

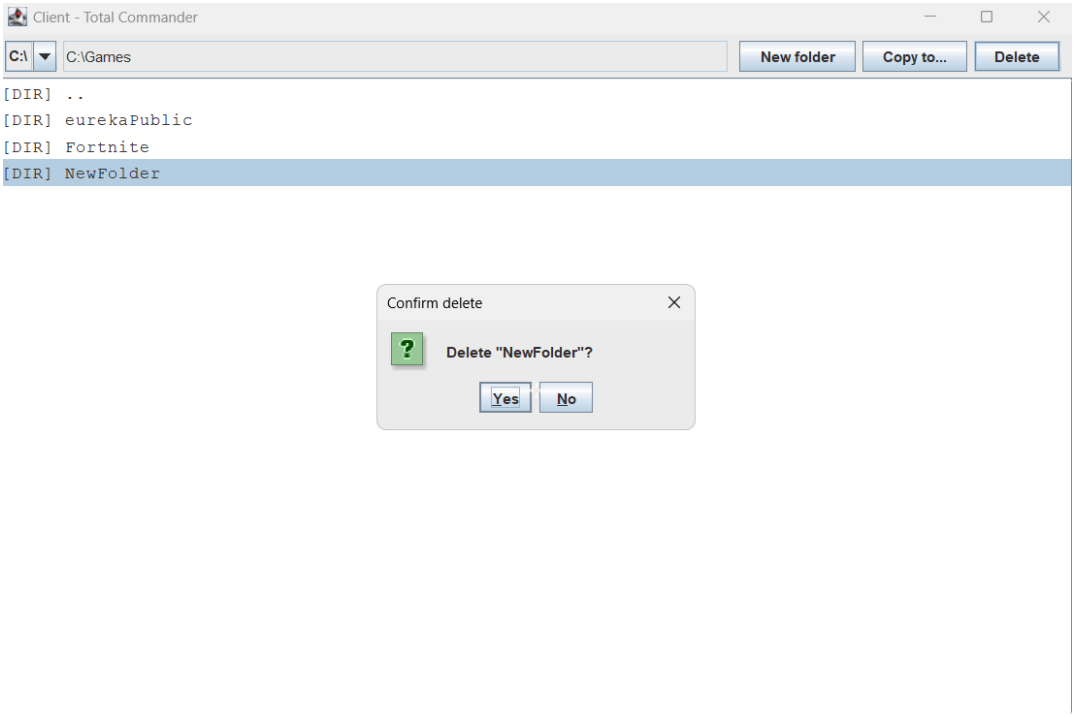

Рис. 3.6 - Підтвердження видалення каталогу NewFolder

- **Результат видалення (рис. 3.7):**
  Після підтвердження клієнт відправляє на сервер команду DELETE для шляху C:\Games\NewFolder. Сервер видаляє каталог. Клієнт оновлює список файлів для C:\Games, з якого зникає видалений каталог NewFolder.

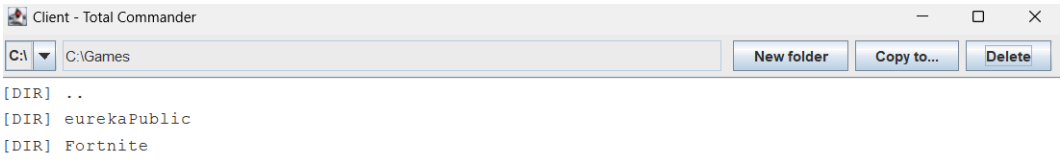

Рис. 3.7 - Каталог NewFolder успішно видалено з C:\Games

3. Діаграма класів

Для візуалізації архітектури розробленої системи та взаємозв'язків між її компонентами була побудована UML-діаграма класів.

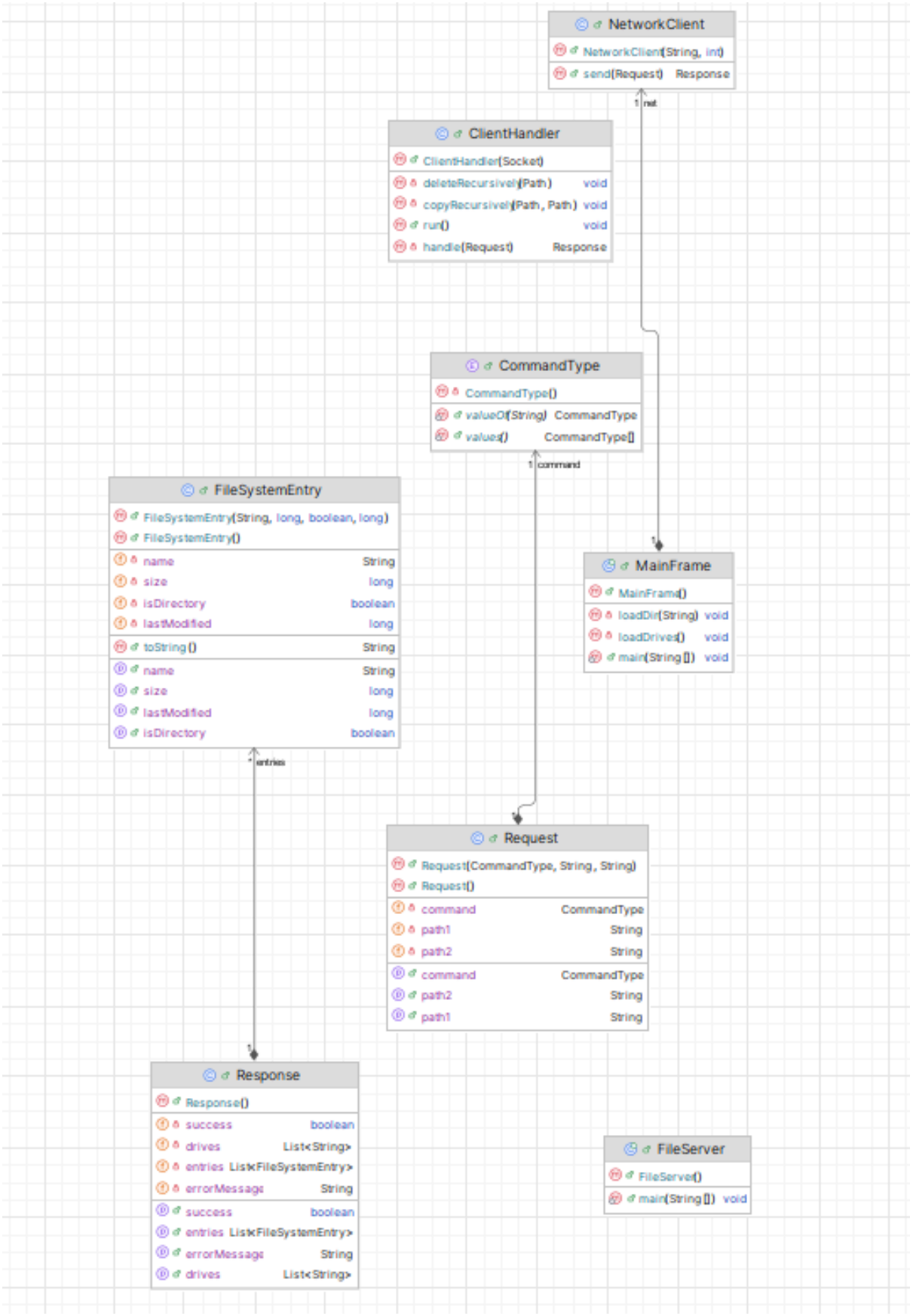

рис 3.1 - діаграма класів (повна)

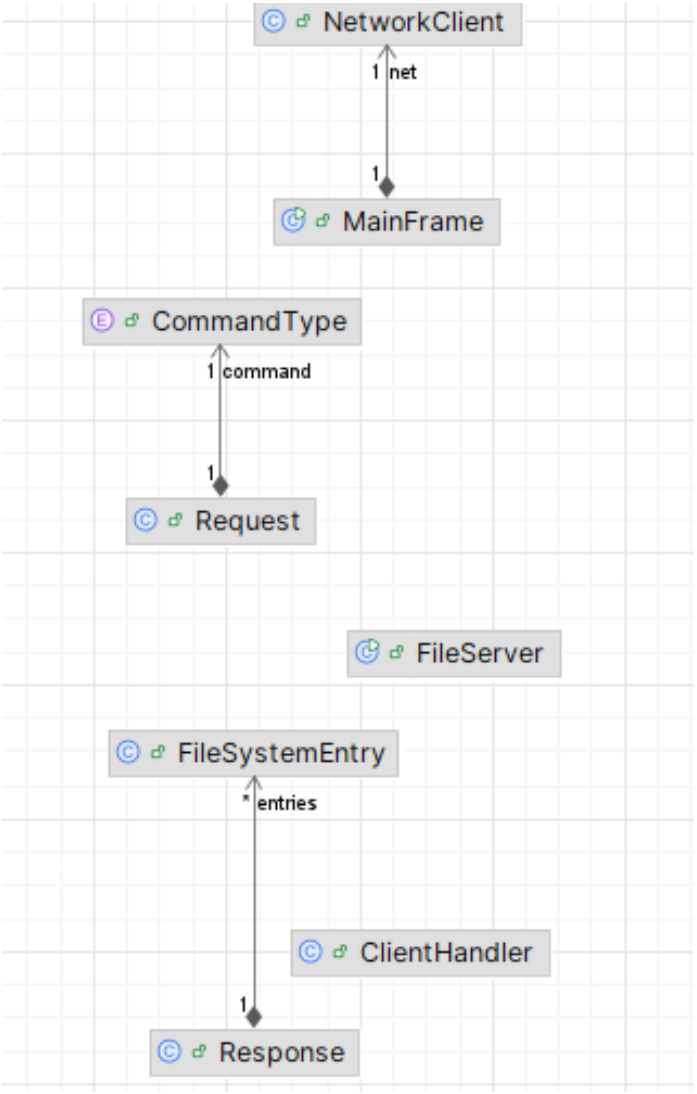

рис 3.2 - діаграма класів (спрощена)

1. Ключові компоненти патерну:

Наведена UML-діаграма ілюструє клієнт-серверну архітектуру розробленого файлового менеджера. В основі взаємодії лежать класи з **спільного модуля (shared)**, що визначають протокол обміну даними. Клас Request інкапсулює запит від клієнта, обов'язково містить CommandType (перелік можливих операцій) та необхідні параметри у вигляді шляхів path1 і path2. У відповідь сервер надсилає об'єкт Response, який містить статус операції success, повідомлення про помилку errorMessage та, за потреби, список рядків drives або список об'єктів FileSystemEntry. FileSystemEntry - це модель, що описує один елемент файлової системи.

- **Серверна частина** представлена класом FileServer, який виступає точкою входу, прослуховує мережевий порт і для кожного нового підключення створює екземпляр ClientHandler. Клас ClientHandler виконується в окремому потоці, відповідаючи за повний цикл обробки одного клієнтського запиту: він десеріалізує отриманий Request, аналізує його CommandType у методі handle і виконує відповідну файлову операцію, наприклад, copyRecursively або deleteRecursively. Після цього він формує об'єкт Response і відправляє його назад клієнту.

- **Клієнтська частина** складається з головного вікна MainFrame та мережевого компонента NetworkClient. MainFrame відповідає за графічний інтерфейс та обробку дій користувача. Він не взаємодіє з мережею напряму, а делегує це завдання об'єкту NetworkClient. При дії користувача MainFrame створює Request і передає його методу send класу NetworkClient. NetworkClient інкапсулює всю логіку з'єднання, серіалізації/десеріалізації та повертає готовий об'єкт Response. Отримавши відповідь, MainFrame оновлює свій стан, наприклад, перезавантажуючи список файлів через методи loadDir або loadDrives. Таким чином, UI-логіка чітко відокремлена від мережевої взаємодії.

4. Висновки

Висновки: В ході виконання даної лабораторної роботи було успішно вивчено та реалізовано клієнт-серверну архітектуру для вирішення задачі розробки розподіленого файлового менеджера. Була створена чітка тримодульна архітектура, що складається з серверної частини (server), клієнтської (client) та спільної бібліотеки класів (shared). Особливу увагу було приділено розподілу відповідальності: впровадження серверного компонента, що складається з класів FileServer та ClientHandler, дозволило централізувати всю складну логіку взаємодії з файловою системою в одному місці, тоді як відповідальність за представлення даних та взаємодію з користувачем була успішно делегована клієнтській частині, представленій класами MainFrame та NetworkClient. Практична реалізація продемонструвала ключові переваги обраної архітектури: чітке розділення відповідальності між логікою представлення (UI) та бізнес-логікою;

високу гнучкість системи, що дозволяє незалежно модифікувати та розгортати клієнт і сервер за умови дотримання спільного контракту, визначеного в модулі shared та надійність і потенціал для масштабування, оскільки серверна частина здатна обробляти декілька клієнтських запитів одночасно завдяки використанню пулу потоків.

Контрольні запитання

1. Що таке клієнт-серверна архітектура?
Це централізована модель взаємодії, в якій програмні компоненти чітко розділені на дві ролі: **клієнти** та **сервери**. Клієнт є ініціатором взаємодії - він відправляє запити на отримання даних або виконання операцій. Сервер є постачальником ресурсів - він постійно очікує на запити, обробляє їх, виконує необхідну логіку та повертає клієнту відповідь.

2. Розкажіть про сервіс-орієнтовану архітектуру.
Сервіс-орієнтована архітектура (SOA) - це підхід до розробки програмного забезпечення, при якому додаток будується з набору незалежних, слабо пов'язаних компонентів, що називаються сервісами. Кожен сервіс реалізує конкретну бізнес-функцію (наприклад, "авторизація користувача", "обробка платежу") і доступний для інших компонентів по мережі через стандартизований інтерфейс.

3. Якими принципами керується SOA?
SOA керується наступними основними принципами:

- **Слабка зв'язаність (Loose Coupling):** Сервіси є незалежними; зміна одного сервісу не повинна ламати інші.
- **Стандартизований контракт:** Кожен сервіс має чітко визначений інтерфейс, що описує, як з ним взаємодіяти.
- **Автономність:** Сервіси самостійно керують своєю логікою та даними.
- **Повторне використання:** Сервіси створюються з розрахунком на те, що їх можна буде використовувати у різних додатках.
- **Прозорість розташування:** Клієнт не знає, де фізично знаходиться сервіс, а звертається до нього за логічною адресою.

4. Як між собою взаємодіють сервіси в SOA?
Сервіси взаємодіють шляхом обміну повідомленнями через мережу за

стандартизованими протоколами, найчастіше **HTTP**. Дані зазвичай передаються у форматах **XML (з протоколом SOAP)** або **JSON (з підходом REST)**. У складних системах взаємодія може відбуватися через центрального посередника - корпоративну сервісну шину (ESB), яка керує маршрутизацією та перетворенням повідомлень.

5. Як розробники взнають про існуючі сервіси і як робити до них запити?

Розробники дізнаються про сервіси за допомогою **реєстру сервісів (Service Registry)** - це спеціальне сховище, що містить інформацію про доступні сервіси та їх адреси. Щоб зрозуміти, як робити запити, розробники використовують **контракт сервісу** - формальний опис його інтерфейсу. Для SOAP-сервісів це **WSDL**-файл, а для REST-сервісів - **OpenAPI (Swagger)** специфікація.

6. У чому полягають переваги та недоліки клієнт-серверної моделі?

- **Переваги:**
  - **Централізація:** Легко керувати даними, оновлювати логіку та забезпечувати безпеку.
  - **Контроль:** Чіткий контроль доступу до ресурсів на сервері.
  - **Масштабованість:** Можна збільшувати потужність сервера, не змінюючи клієнтів.
- **Недоліки:**
  - **Єдина точка відмови:** Якщо сервер не працює, вся система недоступна.
  - **"Вузьке місце":** Сервер може бути перевантажений при великій кількості клієнтів.
  - **Витрати:** Підтримка потужного та надійного сервера може бути дорогою.

7. У чому полягають переваги та недоліки однорангової моделі взаємодії?

- **Переваги:**
  - **Відмовостійкість:** Відсутня єдина точка відмови; мережа працює, доки є активні вузли.
  - **Масштабованість:** Додавання нових вузлів збільшує загальну продуктивність мережі.

    - **Низька вартість:** Немає потреби в утриманні центрального сервера.
- **Недоліки:**
    - **Складність пошуку даних:** Знайти потрібний ресурс у великій мережі може бути складно.
    - **Безпека:** Складно контролювати доступ до даних та їх цілісність.
    - **Нестабільність:** Доступність ресурсів залежить від того, чи онлайн вузли, що їх надають.

8. Що таке мікро-сервісна архітектура?
Мікросервісна архітектура - це різновид сервіс-орієнтованої архітектури, де великий додаток розбивається на набір **дуже маленьких, повністю незалежних сервісів**. Кожен мікросервіс відповідає за одну вузьку бізнес-можливість, має власну базу даних і може бути розроблений, розгорнутий та масштабований абсолютно незалежно від інших.

9. Які протоколи використовуються для обміну даними в мікросервісній архітектурі?

- **Синхронні:** Найчастіше використовується легкий протокол **HTTP/HTTPS** з архітектурним стилем **REST**. Також популярним є **gRPC** для високопродуктивної внутрішньої взаємодії.
- **Асинхронні:** Для забезпечення слабкої зв'язаності та відмовостійкості широко використовуються **протоколи обміну повідомленнями** (наприклад, AMQP, MQTT) через брокери повідомлень, такі як **RabbitMQ** або **Apache Kafka**.

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, коли ми в проєкті між шаром веб-контролерів та шаром доступу до даних реалізуємо шар бізнес-логіки у вигляді сервісів?
Ні, це не є SOA в її класичному розумінні. Це приклад добре структурованої багатошарової (N-tier) архітектури монолітного додатку з використанням патерну "Сервісний шар". Ключова відмінність полягає в тому, що в SOA сервіси є незалежними, розподіленими компонентами, що взаємодіють по мережі.