МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ „КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім.І.Сікорського”

Факультет прикладної математики Кафедра програмного забезпечення комп’ютерних систем

**КУРСОВА РОБОТА**

з дисципліни «Об’єктно-орієнтоване програмування»

на тему

**Шаблони проектування в ООП. Файловий менеждер**

Виконав студент

ІІ курсу групи КП-61

Рухайло Павло Олегович

залікова книжка КП-6124

Керівник роботи

доцент, к.т.н. Заболотня Т.М.

Оцінка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(дата, підпис)

КИЇВ 2018

**ЗМІСТ**

**ВСТУП……………………………………………………………………………..3**

**1. СТРУКТУРНА ОРГАНІЗАЦІЯ РОЗРОБЛЕНОГО**

**ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ…………………………………………...5**

**1.1 Модульна організація програми……………………………………..5**

**1.2 Функціональні характеристики……………………………………..7**

**1.3 Опис реалізованих класів…………………………………………….8**

**2. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ……………………………………………….29**

**2.1. Обґрунтування вибору та опис шаблонів проектування для реалізації програмного забезпечення автомату………………………………29**

**2.2 Результати роботи програми………………………………………....36**

**ВИСНОВКИ……………………………………………………………………….39**

**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ…………………………………..40**

**ВСТУП**

Дана курсова робота призначена розробці файлового менеджера для сімейства операційних систем Ubuntu 12.04 або вище. Задачами файлового менеджера є надання користувачеві графічного інтерфейсу для маніпуляцій з файловою системою компʼютера, такими як перегляд вмісту директорій, перегляд і виконування різних типів файлів (зображення, відео- та аудіо-файли, документи, текстові файли) і так далі.

*Об’єктом* дослідження є структура файлової системи сімейства операційних систем Ubuntu та процес взаємодії з нею.

*Метою роботи* є розроблення програмного забезпечення обліку робочого часу працівників з використанням шаблонів проектування.

Для досягнення визначеної мети необхідно виконати такі *завдання*:

* абстрагувати об’єкти предметної галузі;
* розробити структурну організацію ПЗ за допомогою застосування основних принципів ООП та шаблонів проектування;
* визначити та описати функціональні характеристики програми;
* обґрунтувати вибір шаблонів проектування, використаних для побудови програми;
* розробити дизайн інтерфейсу користувача;
* виконати реалізацію програмного забезпечення відповідно до вимог технічного завдання;
* виконати тестування розробленої програми;
* оформити документацію з курсової роботи.

Розроблене програмне забезпечення “Файловий менеджер” складається з 8-ми модулів: модуль головного процесу програми, модуль рендерер-процесу програми, модуля менеджменту вікон, модуля безпосередньої роботи з файловою системою, модуля зв’язку з модулем файлової системи, модуля інтерфейсу програми, модуля компонентів інтерфейсу програми та модуля спільних ресурсів для головного та рендерер-процесів.

Реалізовані шаблони проектування: Адаптер, Віртуальний заступник, Легковаговик, Спостерігач, Стратегія, Компонувальник, Посередник.

До функціональних можливостей програми належать: перегляд вмісту директорій файлової системи, перегляд інформації про них, пересування між ними, відкриття/виконання файлів через програми, які встановлені на конкретному комп’ютері та призначені для роботи з файлами даного типу.

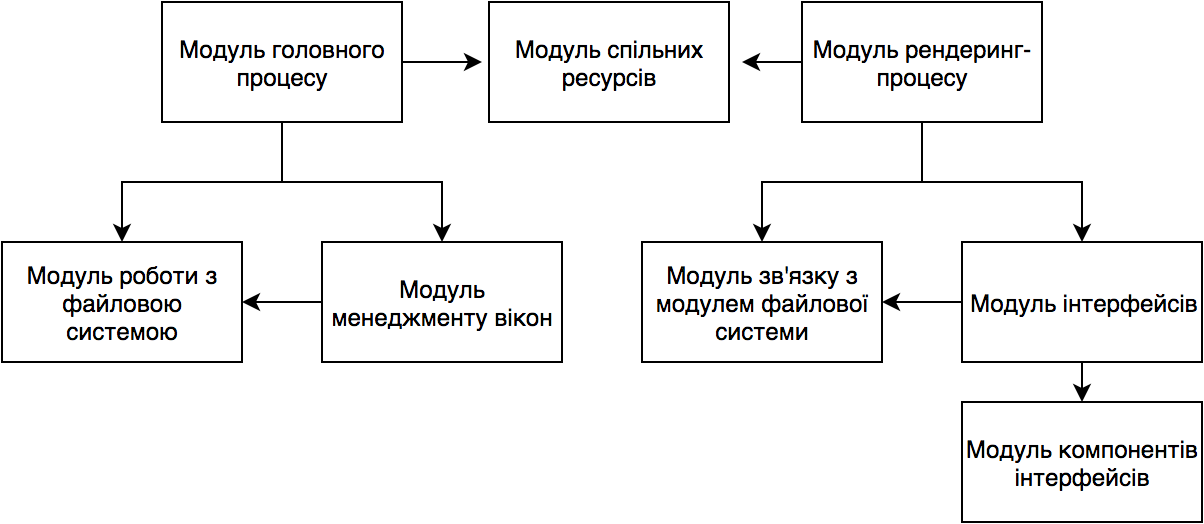
Для функціонування розробленої програми необхідно забезпечити наявність на комп’ютері встановленого оточення Node.js.

Розроблене програмне забезпечення може використовуватися як у повсякденному використанні компʼютера, так і в більш професійних сферах, оскільки представляє собою універсальний інструмент менеджменту файлової системи.

**1. СТРУКТУРНА ОРГАНІЗАЦІЯ РОЗРОБЛЕНОГО**

**ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

**1.1 Модульна організація програми**

[****](https://www.draw.io/?scale=2#G1mWeB5GpqiNFhxFIxKA3TnSZykwu8GvuZ)

**Модуль головного процесу** відповідає за ініціанування програми та налаштування її роботи. Містить в собі модуль роботи з файловою системою та модуль менеджменту вікон.

**Модуль роботи з файловою системою** надає можливість маніпулювати з файловою системою комп’ютера. Містить функціонал для створення, перегляду та змінення об’єктів файлової системи.

**Модуль менеджменту вікон** відповідає за створення вікон та налаштування і підтримку їх життєвого циклу. Ініціалізує вікна та зв’язує їх з відповідними рендерер-процесами.

**Модуль рендерер-процесу** містить в собі функціонал для роботи інтерфейсів програми. Налаштовує зв’язок між рендерер- та головним процесом зі сторони першого.

**Модуль зв’язку з модулем файлової системи** надає можливість маніпулювати з файловою системою безпосередньо з рендерер-процесу, відповідаючи за налаштування та підтримку зв’язку з об’єктами роботи з файловою системою, які знаходяться в головному процесі виконяння програми.

**Модуль інтерфейсів** програмного забезпечення відповідає за компонування різних елементів в єдині інтерфейси користувача.

**Модуль компонентів інтерфейсів** програмного забезпечення відповідає за показ користувачеві результатів різних операцій з файловою системою та надання команд програмі.

**1.2 Функціональні характеристики**

Розроблений програмний продукт надає кінцевому користувачеві зручний інтерфейс для маніпулювання з файловою системою.

При запуску програми користувач одразу потрапляє на головний екран програми, який розділено на декілька частин.

Більшу частину займає компонент показу вмісту поточної директорії. При запуски буде відображатися вміст домашньої директорії поточного користувача системи. Користувач може побачити там всі файли поточної директорії. За допомогою подвійного кліку користувач матиме можливість переключитися в нову директорію та побачити її вміст. Якщо ж користувач клікне два рази по файлу, то файловий менеджер спробує знайти на комп’ютері програму, призначену для роботу з файлами даного типу. Якщо така програма буде знайдена, то вибраний файл буде запущений вже в цій програмі для подальших змін або подальшого перегляду. Також розроблений продукт буде забезпечувати користувача завжди актуальною інформацією, оскільки всі сторонні зміни в директоріях одразу відображаються в інтерфейсі розробленого програмного забезпечення. За допомогою меню, яке розміщене в верхній частині інтерфейсу користувач матиме можливість змінювати відображення вмісту директорії за своїм бажанням: або у вигляді іконок та назв, або у вигляді таблиці з більш детальною інформацією про той чи інший об’єкт, та включити відображення прихованих файлів та директорій.

В верхній частині програми також є кнопки навігації між раніше відкритими директоріями.

Також в інтерфейсі програми є компонент для відображення ієрархії файлової системи: дерево директорій. За допомогою цього компонента користувач матиме більш гнучких механізм переходу між директоріями та більш наглядну інформацію про стан файлової системи.

**1.3 Опис реалізованих класів**

1. FileSystemWatcher - даний клас надає потрібний функціонал для нагляду за змінами в конкретних директоріях та оповіщення про ці зміни всіх зацікавлених об’єктів. Належить до модулю файлової системи.

Основні методи:

* setListener(id: number, onChange: (id: number) => void) - підписує об’єкт на отримання інформації про зміни в конкретній директовії (шаблон Спостерігач)
* removeListener(id: number) : boolean - відписує об’єкт від отримання вищє зазначеної інформації

|  |
| --- |
| FileSystemWatcher.ts |
| import fs from "fs"  type CallbackDictionary = {  [id: number]: (id: number) => void;  }  export default class FileSystemWatcher {  private watcher: fs.FSWatcher;  private callbackDictionary: CallbackDictionary;  private timeOfLastEvent: number  constructor(path: string) {  this.callbackDictionary = {};  this.timeOfLastEvent = Date.now() - 100  this.watcher = fs.watch(path, (eventType, filename) => {  this.handleEvent(eventType, filename);  });  }  setListener(id: number, onChange: (id: number) => void){  this.callbackDictionary[id] = onChange;  }  removeListener(id: number) : boolean{  if(id in this.callbackDictionary){  delete this.callbackDictionary[id];  return true;  }  return false;  }  close(): void{  this.watcher.close();  this.watcher = null;  }  removable() : boolean{  for(let id in this.callbackDictionary){  return false;  }  return true;  }  private handleEvent(eventType: string, filename: string): void {  if (eventType == "change") return;  let timeOfNewEvent = Date.now();  if (timeOfNewEvent - this.timeOfLastEvent < 100) return;  this.timeOfLastEvent = timeOfNewEvent;  this.emitChanges();  }  private emitChanges(): void {  let promises: Promise<{}>[] = [];  for (let callbackId in this.callbackDictionary) {  promises.push(new Promise<{}>((resolve, reject) => {  this.callbackDictionary[callbackId](parseInt(callbackId));  resolve();  }))  }  Promise.all(promises);  }  } |

2. FileSystemWatcherManager - клас, який відповідає за підписування об’єктів на об’єкти класу FileSystemWatcher, запобігаючи створенню різних спостерігачів за однією і тією-ж директорією. Належить до модулю файлової системи.

Основні методи:

* getWatcher(path: string): FileSystemWatcher - отримання спостерігача над конкретною директорією (якщо немає такого спостерігача, то буде створений новий) (шаблон Легковаговик)

|  |
| --- |
| FileSystemWatcherManager.ts |
| import FileSystemWatcher from "./FileSystemWatcher"  type FileSystemWatchersDictionary = {  [path: string]: FileSystemWatcher  }  export default class FileSystemWatcherManager {  dictionary: FileSystemWatchersDictionary;  id: number  constructor() {  this.dictionary = {};  this.id = 0;  }  setListener(path: string, onChange: (id: number) => void): number {  const watcher = this.getWatcher(path);  const id = this.id++;  watcher.setListener(id, onChange);  return id;  }  removeListener(id: number): void {  for(let path in this.dictionary){  let removed = this.dictionary[path].removeListener(id)  if(removed && this.dictionary[path].removable) {  this.dictionary[path].close();  delete this.dictionary[path]  return  }  }  }  private getWatcher(path: string): FileSystemWatcher {  if(!(path in this.dictionary)) this.dictionary[path] = new FileSystemWatcher(path);  return this.dictionary[path];  }  } |

3. FileSystem - клас для маніпулювання над файловою системою. Реалізує інтерфейс IFileSystem з модулю спільних ресурсів, використовуючи інтерфейси середовища Node.js (шаблон Адаптер)

|  |
| --- |
| FileSystem.ts |
| import fs from "fs"  import os from "os"  import pth from "path"  import { ipcMain, app } from "electron"  import IFileSystem from "../../common/IFileSystem"  import { FileDescription, FolderDescription, Description } from "../../common/Descriptions"  import {  FileSystemSignals, GetContentArgv,  CreateFolderArgv, RemoveFolderArgv,  CreateFileArgv, RemoveFileArgv,  RenameArgv, WrappedArgv,  Argv, Answer, SetListenerArgv, RemoveListenerArgv, Listener  } from "../../common/FileSystemConnection"  import AppManager from "../Management/AppManager"  import FileSystemWatcherManager from "./FileSystemWatcherManager"  class FileSystem implements IFileSystem {  private static instance: FileSystem;  private watcherManager: FileSystemWatcherManager;  private constructor() {  this.watcherManager = new FileSystemWatcherManager();  const handleReq = async (event, id: number, signal: FileSystemSignals, func) => {  let answer: Answer  try {  let value = await func();  answer = {  id,  error: null,  value  }  } catch (error) {  answer = {  id,  error: error.message,  value: null  }  } finally {  event.sender.send(signal, answer);  }  }  const broadcast = (signal: FileSystemSignals, argv: Listener) => {  AppManager.sendToAllMainWindows(signal, argv);  }  ipcMain.on(FileSystemSignals.GET\_CONTENT, async (event, req: WrappedArgv) => {  const { id } = req;  const argv = req.argv as GetContentArgv  handleReq(event, id, FileSystemSignals.GET\_CONTENT, async () => {  return this.getContent(argv.path)  })  })  ipcMain.on(FileSystemSignals.CURRENT\_USER\_HOME\_FOLDER, (event, req: WrappedArgv) => {  const { id } = req;  handleReq(event, id, FileSystemSignals.CURRENT\_USER\_HOME\_FOLDER, async () => {  return this.getCurrentUserHomeFolder()  })  })  ipcMain.on(FileSystemSignals.CREATE\_FOLDER, async (event, req: WrappedArgv) => {  const { id } = req;  const argv = req.argv as CreateFolderArgv  handleReq(event, id, FileSystemSignals.CREATE\_FOLDER, async () => {  return this.createFolder(argv.path, argv.name)  })  })  ipcMain.on(FileSystemSignals.REMOVE\_FOLDER, async (event, req: WrappedArgv) => {  const { id } = req;  const argv = req.argv as RemoveFolderArgv  handleReq(event, id, FileSystemSignals.REMOVE\_FOLDER, async () => {  return this.removeFolder(argv.dir)  })  })  ipcMain.on(FileSystemSignals.CREATE\_FILE, async (event, req: WrappedArgv) => {  const { id } = req;  const argv = req.argv as CreateFileArgv;  handleReq(event, id, FileSystemSignals.CREATE\_FILE, async () => {  return this.createFile(argv.path, argv.name, argv.ext);  })  })  ipcMain.on(FileSystemSignals.REMOVE\_FILE, async (event, req: WrappedArgv) => {  const { id } = req;  const argv = req.argv as RemoveFileArgv  handleReq(event, id, FileSystemSignals.REMOVE\_FILE, async () => {  return this.removeFile(argv.file)  })  })  ipcMain.on(FileSystemSignals.RENAME, async (event, req: WrappedArgv) => {  const { id } = req;  const argv = req.argv as RenameArgv  handleReq(event, id, FileSystemSignals.RENAME, async () => {  return this.rename(argv.desc, argv.newName)  })  })  ipcMain.on(FileSystemSignals.SET\_LISTENER, async (event, req: WrappedArgv) => {  const { id } = req;  const argv = req.argv as SetListenerArgv;  handleReq(event, id, FileSystemSignals.SET\_LISTENER, async () => {  return this.setListenerForChanges(argv.path, (idOfListener: number) => {  broadcast(FileSystemSignals.LISTENER, { id: idOfListener })  });  })  })  ipcMain.on(FileSystemSignals.REMOVE\_LISTENER, async (event, req: WrappedArgv) => {  const { id } = req;  const argv = req.argv as RemoveListenerArgv;  handleReq(event, id, FileSystemSignals.REMOVE\_LISTENER, async () => {  return this.removeListenerForChanges(argv.id);  })  })  }  public static getInstance(): FileSystem {  if (!this.instance) this.instance = new FileSystem();  return this.instance;  }  async getCurrentUserHomeFolder(): Promise<string> {  return os.homedir();  }  private generateDescription(path: string, fullName: string, stats: fs.Stats): Description {  if (stats.isDirectory()) {  return {  name: fullName,  path: pth.join(path, fullName),  created: stats.ctimeMs,  modified: stats.ctimeMs,  size: stats.size  }  }  let dotExt = pth.extname(fullName);  return {  name: dotExt.length > 0 ? fullName.slice(0, -dotExt.length) : fullName,  ext: dotExt.slice(1),  path: pth.join(path, fullName),  created: stats.ctimeMs,  modified: stats.ctimeMs,  size: stats.size  }  }  async getStats(path: string, name: string): Promise<fs.Stats> {  return new Promise<fs.Stats>((resolve, reject) => {  fs.stat(pth.join(path, name), (error, stats) => {  if (error) reject(error);  resolve(stats);  })  })  }  async getContent(path: string): Promise<Description[]> {  return new Promise<Description[]>((resolve, reject) => {  fs.readdir(path, async (error, files) => {  if (error) reject(error);  let statsPromises: Promise<fs.Stats>[] = files.map(file => {  return this.getStats(path, file);  });  let stats = await Promise.all(statsPromises);  let descriptions = stats.map((stats, i) => {  return this.generateDescription(path, files[i], stats);  })  resolve(descriptions);  });  })  }  async createFolder(path: string, name: string): Promise<{}> {  return new Promise((resolve, reject) => {  fs.readdir(path, (error, existNames) => {  let newName: string;  main: for (let i = 0; ; i++) {  newName = i == 0 ? name : `${name}(${i})`;  for (let existName of existNames) {  if (existName == newName) continue main;  }  break main;  }  fs.mkdir(pth.join(path, newName), error => {  if (error) reject(error);  resolve();  })  })  });  }  async removeFolder(dir: FolderDescription): Promise<{}> {  return new Promise((resolve, reject) => {  fs.rmdir(dir.path, error => {  if (error) reject(error);  resolve();  });  });  }  async createFile(path: string, name: string, ext: string): Promise<{}> {  return new Promise((resolve, reject) => {  fs.readdir(path, (error, existNames) => {  if (error) reject(error);  let filename;  main: for (let i = 0; ; i++) {  filename = i == 0 ? name : `${name}(${i}).${ext}`  for (let existName of existNames) {  if (filename == existName) continue main;  }  break main;  }  fs.writeFile(pth.join(path, filename), "", error => {  if (error) reject(error);  resolve();  })  });  });  }  async removeFile(file: FileDescription): Promise<{}> {  return new Promise((resolve, reject) => {  fs.unlink(file.path, error => {  if (error) reject(error);  resolve()  })  })  }  async rename(desc: Description, newName: string): Promise<{}> {  return new Promise((resolve, reject) => {  let path = desc.path.slice(0, desc.path.lastIndexOf('/'));  let newPath = pth.join(path, newName);  fs.readdir(path, (error, existNames) => {  if (error) reject(error);  for (let existName of existNames) {  if (existName == newName) reject(new Error(`${newName} already exists`))  }  fs.rename(desc.path, newPath, error => {  if (error) reject(error);  resolve();  })  })  });  }  async setListenerForChanges(path: string, onChange: (id: number) => void): Promise<number> {  return this.watcherManager.setListener(path, onChange);  }  async removeListenerForChanges(id: number): Promise<{}> {  this.watcherManager.removeListener(id);  return;  }  }  export default FileSystem.getInstance() |

4. Window - абстрактний клас, описуючий базові налаштування та поведінку будь-якого класу. Належить до модулю менеджменту вікон.

|  |
| --- |
| Window.ts |
| import { BrowserWindow, BrowserWindowConstructorOptions } from "electron"  export default abstract class Window {  protected window: BrowserWindow  protected id: number  constructor(options : BrowserWindowConstructorOptions | undefined) {  this.window = new BrowserWindow(options)  this.id = this.window.webContents.id  this.window.once('ready-to-show', () => {  this.window.show()  })  }  getId() : number {  return this.id  }  abstract close() : void;  } |

5. MainWindow - клас підтримки життєвого циклу головного вікна зі сторони головного процесу. Наслідується від класу Window. Відповідає за операцію над конкретним вікном програми, такими як мінімізування, перехід в режим fullscreen та закривання. Також надає механізм для зв’язку з рендеринг-процесом конкретного вікна. Належить до модулю менеджменту вікон. Імплементує інтерфейс IMainWindow з модулю спільних ресурсів.

|  |
| --- |
| MainWindow.ts |
| import url from "url"  import path from "path"  import { BrowserWindowConstructorOptions } from "electron"  import Window from "./Window"  import IMainWindow from "../../common/IMainWindow"  import AppManager from "./AppManager"  export default class MainWindow extends Window implements IMainWindow {  private pathToLoad : string;  constructor(pathToLoad: string) {  const options : BrowserWindowConstructorOptions = {  width: 800,  height: 450,  minWidth: 350,  minHeight: 250,  frame: false  }  super(options)  this.pathToLoad = pathToLoad  this.window.loadURL(url.format({  pathname: path.resolve(\_\_dirname, "views/MainWindowView/index.html"),  protocol: 'file:',  slashes: true  }))  this.window.on("closed", () => {  AppManager.removeReferenceToMainWindowById(this.id)  this.window = null  })  }  close() : void {  this.window.close();  }  minimize() : void {  this.window.minimize();  }  changeFullscreenMode() : void {  let enabled = this.window.isFullScreen();  this.window.setFullScreen(!enabled);  }  send(signal, argv): void {  this.window.webContents.send(signal, argv);  }  } |

6. AppManager - клас менеджменту вікон програми. Відповідає за створення нових вікон. Належить до модулю менеджменту вікон.

Основні методи:

* openNewMainWindow(pathToLoad: string = undefined): void - відкриття нового головного вікна
* sendToAllMainWindows(signal, argv): void - розповсюдження певної інформації всім вікнам

|  |
| --- |
| AppManager.ts |
| import { app, ipcMain } from "electron"  import MainWindow from "./MainWindow"  import FileSystem from "../FileSystem/FileSystem"  import { MainWindowSignals } from "../../common/MainWindowConnection"  interface MainWindowsDictionary {  [id: number]: MainWindow  }  class App {  private static instance: App;  private mainWindowDictionary: MainWindowsDictionary = {} as MainWindowsDictionary;  private constructor() {  app.on('window-all-closed', () => {  app.quit()  })  ipcMain.on(MainWindowSignals.CLOSE, (event) => {  let window : MainWindow = this.mainWindowDictionary[event.sender.id];  window.close()  });  ipcMain.on(MainWindowSignals.CHANGE\_FULLSCREEN\_MODE, (event) => {  let window : MainWindow = this.mainWindowDictionary[event.sender.id];  window.changeFullscreenMode()  });  ipcMain.on(MainWindowSignals.MINIMIZE, (event) => {  let window : MainWindow = this.mainWindowDictionary[event.sender.id];  window.minimize()  });  }  static getInstance(): App {  if (!this.instance) this.instance = new App()  return this.instance  }  openNewMainWindow(pathToLoad: string = undefined): void {  const f = async () => {  if(!pathToLoad) pathToLoad = await FileSystem.getCurrentUserHomeFolder();  if (!app.isReady()) {  setTimeout(() => { this.openNewMainWindow(pathToLoad) }, 100);  return;  }    let window: MainWindow = new MainWindow(pathToLoad);  this.mainWindowDictionary[window.getId()] = window;  }  f();  }  getMainWindowById(id: number): MainWindow {  return this.mainWindowDictionary[id]  }  removeReferenceToMainWindowById(id: number): void {  delete this.mainWindowDictionary[id]  }  sendToAllMainWindows(signal, argv): void{  for(let id in this.mainWindowDictionary){  this.mainWindowDictionary[id].send(signal, argv);  }  }  }  export default App.getInstance(); |

7. RemoteFileSystem - клас, який використовується для роботи з файловою системою зі сторони рендеринг-процесу. Реалізує інтерфейс IFileSystem, але, на відміну від класу FileSystem, не використовує системні модулі, а лише передає запити до об’єкту вище зазначеного класу за допомогою певного механізму зв’язку (шаблон Віртуальний заступник). Належить до модулю зв’язку з модулем файлової системи.

|  |
| --- |
| RemoteFileSystem.ts |
| import { ipcRenderer } from "electron"  import {  FileSystemSignals, GetContentArgv,  CreateFolderArgv, RemoveFolderArgv,  CreateFileArgv, RemoveFileArgv,  RenameArgv, WrappedArgv,  Argv, Answer, SetListenerArgv, RemoveListenerArgv  } from "../../common/FileSystemConnection"  import { FileDescription, FolderDescription, Description } from "../../common/Descriptions"  import IFileSystem from "../../common/IFileSystem"  import RemoteFileSystemWatcherManager from "./RemoteFileSystemWatcherManager";  class RemoteFileSystem implements IFileSystem {  private static instance: RemoteFileSystem;  private watcherManager: RemoteFileSystemWatcherManager;  private id: number = 0;  private constructor() {  this.watcherManager = new RemoteFileSystemWatcherManager()  }  static getInstance(): RemoteFileSystem {  if (this.instance == null) this.instance = new RemoteFileSystem();  return this.instance;  }  async wrapSignal(signal: FileSystemSignals, argv: Argv): Promise<any> {  return new Promise<any>((resolve, reject) => {  const id = this.id++;  const handler = (event, answer: Answer) => {  if (answer.id != id) {  ipcRenderer.once(signal, handler)  return;  }  if (answer.error) reject(answer.error);  resolve(answer.value);  }  ipcRenderer.once(signal, handler)  const req: WrappedArgv = {  id,  argv  }  ipcRenderer.send(signal, req);  })  }  async getContent(path: string): Promise<Description[]> {  try {  let argv: GetContentArgv = {  path  }  return await this.wrapSignal(FileSystemSignals.GET\_CONTENT, argv) as Description[]  } catch (error) {  throw error  }  }  async getCurrentUserHomeFolder(): Promise<string> {  try {  let argv;  return await this.wrapSignal(FileSystemSignals.CURRENT\_USER\_HOME\_FOLDER, argv) as string  } catch (error) {  throw error  }  }  async createFolder(path: string, name: string): Promise<{}> {  try {  let argv: CreateFolderArgv = {  path,  name  }  return await this.wrapSignal(FileSystemSignals.CREATE\_FOLDER, argv)  } catch (error) {  throw error  }  }  async removeFolder(dir: FolderDescription): Promise<{}> {  try {  let argv: RemoveFolderArgv = {  dir  }  return await this.wrapSignal(FileSystemSignals.REMOVE\_FOLDER, argv)  } catch (error) {  throw error  }  }  async createFile(path: string, name: string, ext: string): Promise<{}> {  try {  let argv: CreateFileArgv = {  path,  name,  ext  }  return await this.wrapSignal(FileSystemSignals.CREATE\_FILE, argv)  } catch (error) {  throw error  }  }  async removeFile(file: FileDescription): Promise<{}> {  try {  let argv: RemoveFileArgv = {  file  }  return await this.wrapSignal(FileSystemSignals.REMOVE\_FILE, argv)  } catch (error) {  throw error  }  }  async rename(desc: Description, newName: string): Promise<{}> {  try {  let argv: RenameArgv = {  desc,  newName  }  return await this.wrapSignal(FileSystemSignals.RENAME, argv)  } catch (error) {  throw error  }  }  async setListenerForChanges(path: string, onChange: (id: number) => void): Promise<number> {  if(!this.watcherManager.isContainWatcher(path)){  try {  let argv: SetListenerArgv = {  path  }  let id = await this.wrapSignal(FileSystemSignals.SET\_LISTENER, argv) as number  this.watcherManager.createWatcher(path, id);  } catch (error) {  throw error  }  }  return this.watcherManager.setListener(path, onChange);  }  async removeListenerForChanges(id: number): Promise<{}>{  let removedWatcherRealId = this.watcherManager.removeListener(id);  if(removedWatcherRealId){  try {  let argv: RemoveListenerArgv = {  id: removedWatcherRealId  }  return await this.wrapSignal(FileSystemSignals.REMOVE\_LISTENER, argv)  } catch (error) {  throw error  }  }  }  }  export default RemoteFileSystem.getInstance(); |

8. RemoteFileSystemWatcher - допоміжний до попереднього класу клас, який зберігає зв’язок між об’єктом рендеринг-процесу та спостерігачем з головного процесу. У разі приходу сигналу відповідного сигналу від спостерігача, сповіщує всі підписані об’єкти про зміни у директорії (поєднання шаблонів Віртуальний заступник та Спостерігач).

Основні методи:

* setListener(id: number, onChange: (id: number) => void) - підписує об’єкт на отримання інформації про зміни в конкретній директовії (шаблон Спостерігач)
* removeListener(id: number) : boolean - відписує об’єкт від отримання вищє зазначеної інформації

|  |
| --- |
| RemoteFileSystemWatcher.ts |
| type CallbackDictionary = {  [id: number]: (id: number) => void  }  export default class RemoteFileSystemWatcher {  private dictionary: CallbackDictionary;  readonly realId: number  constructor(realId: number) {  this.realId = realId;  this.dictionary = {}  }  setListener(id: number, onChange: (id: number) => void) {  this.dictionary[id] = onChange;  }  removeListener(id: number): boolean {  if (!(id in this.dictionary)) return false;  delete this.dictionary[id];  return true;  }  removable(): boolean {  for (let id in this.dictionary) return false;  return true;  }  emit(): void{  let promises: Promise<{}>[] = [];  for (let id in this.dictionary) {  promises.push(new Promise<{}>((resolve, reject) => {  this.dictionary[id](parseInt(id));  resolve();  }))  }  Promise.all(promises);  }  } |

9. RemoteFileSystemWatcherManager - клас, аналогічний за призначенням до класу FileSystemWatcherManager. Але, також зменшує навантаження на об’єкт класу FileSystemWatcherManager, оскільки для підключення об’єкта до директорії, за якою спостерігає хоча-б один інший об’єкт, використовуються вже закешовани об’єкти зв’язку без створення нових (шаблон Заступник).

Належить до модулю зв’язку з модулем файлової системи.

|  |
| --- |
| RemoteFileSystemWatcherManager.ts |
| import { ipcRenderer } from "electron"  import RemoteFileSystemWatcher from "./RemoteFileSystemWatcher"  import { Listener, FileSystemSignals } from "../../common/FileSystemConnection"  type WatcherDictionary = {  [path: string]: RemoteFileSystemWatcher  }  export default class RemoteFileSystemWatcherManager {  private watcherDictionary: WatcherDictionary;  private id: number;  constructor() {  this.watcherDictionary = {}  this.id = 0;  ipcRenderer.on(FileSystemSignals.LISTENER, (event, listener: Listener) => {  for(let path in this.watcherDictionary){  if(this.watcherDictionary[path].realId == listener.id){  this.watcherDictionary[path].emit();  }  return;  }  })  }  isContainWatcher(path: string): boolean {  return path in this.watcherDictionary;  }  createWatcher(path: string, realId: number): RemoteFileSystemWatcher {  this.watcherDictionary[path] = new RemoteFileSystemWatcher(realId);  return this.watcherDictionary[path];  }  setListener(path: string, onChange: (id: number) => void): number {  const id = this.id++;  this.watcherDictionary[path].setListener(id, onChange);  return id;  }  removeListener(id: number): number {  for (let path in this.watcherDictionary) {  let removed = this.watcherDictionary[path].removeListener(id);  if (removed && this.watcherDictionary[path].removable()) {  let id = this.watcherDictionary[path].realId;  delete this.watcherDictionary[path];  return id;  }  }  }  } |

10. FileRunner - дозволяє виконувати/переглядати файли певних типів в відповідних програмах, встановлених на поточному компʼютері. Для реалізації використовуються системні виклики та створення нових процесів за допомогою виклику fork(). Належить до модулю інтерфейсів програми.

|  |
| --- |
| FileRunner.ts |
| import child\_process from "child\_process"  export default class FileRunner {  static run(path: string){  let modifiedPath = path.split(' ').join('\\ ')  child\_process.exec(`xdg-open ${modifiedPath}`)  }  } |

11. RemoteMainWindow - реалізує інтерфейс IMainWindow з модулю спільних ресурсів. Надає механізм для керування станом вікна безпосередньо з рендерер-процесу шляхом відправки запитів до відповідного об’єкту класу MainWindow. (шаблон Віртуальний заступник)

|  |
| --- |
| RemoteMainWindow.ts |
| import { ipcRenderer } from "electron"  import { MainWindowSignals } from "../common/MainWindowConnection"  import IMainWindow from "../common/IMainWindow"  class RemoteMainWindow implements IMainWindow {  private static instance: RemoteMainWindow  private constructor() { }  public static getInstance(): RemoteMainWindow {  if (!this.instance) this.instance = new RemoteMainWindow();  return this.instance;  }  public close(): void {  ipcRenderer.send(MainWindowSignals.CLOSE);  }  public minimize(): void {  ipcRenderer.send(MainWindowSignals.MINIMIZE);  }  public changeFullscreenMode(): void{  ipcRenderer.send(MainWindowSignals.CHANGE\_FULLSCREEN\_MODE);  }  }  export default RemoteMainWindow.getInstance() |

*Примітка:* подальші класи за своєю специфікою не мають публічних методів, оскільки представляють собою механізми побудови інтерфейсів. Вся логіка та шаблони проектування реалізуються шляхом передачі певних об’єктів від компонента до компонента.

12. WindowButtons - клас, відповідний за відображення та функціонування кнопок управління вікном. Також реалізує меню вибору типу відображення вмісту директорії в класі DirectoryBrowser.

|  |
| --- |
| WindowButtons.tsx |
| import \* as React from "react";  import RemoteMainWindow from "../../RemoteMainWindow"  import IconButton from "material-ui/IconButton"  import MoreIcon from "@material-ui/icons/MoreHoriz"  import MinimizeIcon from "@material-ui/icons/Remove"  import FullscreenIcon from "@material-ui/icons/Fullscreen"  import CloseIcon from "@material-ui/icons/Cancel"  import ExpandMoreIcon from '@material-ui/icons/ExpandMore';  import withStyles, { WithStyles } from "material-ui/styles/withStyles"  import createMuiTheme from "material-ui/styles/createMuiTheme"  import MuiThemeProvider from "material-ui/styles/MuiThemeProvider"  import red from "material-ui/colors/red"  import Menu from 'material-ui/Menu'  import Radio, { RadioGroup } from 'material-ui/Radio'  import { FormLabel, FormControl, FormControlLabel, FormHelperText, FormGroup } from 'material-ui/Form'  import DrawStrategy from "./DrawStrategy"  import Checkbox from 'material-ui/Checkbox'  const style = () => ({  container: {  marginRight: "-15px"  },  button: {  "-webkit-app-region": "no-drag"  }  })  const theme = createMuiTheme({  palette: {  primary: {  main: red[600]  },  secondary: {  main: "#ffffff"  }  }  })  type ComponentState = {  menuAnchor: any,  drawStrategy: any,  showHidden: boolean,  }  type ComponentProps = {  drawStrategy: any,  showHidden: boolean,  onDrawStrategyChanged: (drawStrategy: any, showHidden: boolean) => void  }  type ComponentPropsWithStyle = ComponentProps & WithStyles<'button' | 'container'>  class WindowButtons extends React.Component<ComponentPropsWithStyle, ComponentState>{  state = {  menuAnchor: null,  drawStrategy: this.props.drawStrategy,  showHidden: this.props.showHidden  }  static getDerivedStateFromProps(nextProps: ComponentProps, prevState: ComponentState) {  return {  drawStrategy: nextProps.drawStrategy,  showHidden: nextProps.showHidden  }  }  handleDrawStrategyChanged(drawStrategy: any, showHidden: boolean) {  this.props.onDrawStrategyChanged(drawStrategy, showHidden)  }  render() {  const { classes } = this.props  return (  <MuiThemeProvider theme={theme}>  <div className={classes.container}>  <IconButton  aria-owns={this.state.menuAnchor ? 'menu' : null}  aria-haspopup="true"  color="secondary"  className={classes.button}  onClick={event => {  this.setState({  menuAnchor: event.currentTarget  })  }}  >  <MoreIcon />  </IconButton>  <Menu  id="menu"  anchorEl={this.state.menuAnchor}  open={this.state.menuAnchor != null}  onClose={() => { this.setState({ menuAnchor: null }) }}  >  <FormGroup>  <FormControl component="fieldset">  <FormLabel component="legend">View type:</FormLabel>  <RadioGroup  aria-label="drawStrategy"  name="drawStrategy"  value={this.props.drawStrategy == DrawStrategy.Icon ? "Icon" : "List"}  onChange={event => {  this.handleDrawStrategyChanged(  DrawStrategy[(event.target as any).value],  this.state.showHidden  )  }}  >  <FormControlLabel value="Icon" control={<Radio color="primary" />} label="Icons" />  <FormControlLabel value="List" control={<Radio color="primary" />} label="List" />  </RadioGroup>  </FormControl>  <FormControlLabel  control={  <Checkbox  checked={this.state.showHidden}  onChange={event => {  this.handleDrawStrategyChanged(  this.state.drawStrategy,  !this.state.showHidden  )  }}  color="primary"  />  }  label="Show hidden"  />  </FormGroup>  </Menu>  <IconButton color="secondary" className={classes.button} onClick={() => { RemoteMainWindow.minimize() }}>  <MinimizeIcon />  </IconButton>  <IconButton color="secondary" className={classes.button} onClick={() => { RemoteMainWindow.changeFullscreenMode() }}>  <FullscreenIcon />  </IconButton>  <IconButton color="primary" className={classes.button} onClick={() => { RemoteMainWindow.close() }}>  <CloseIcon />  </IconButton>  </div>  </MuiThemeProvider>  );  }  }  export default withStyles(style)<ComponentProps>(WindowButtons); |

13. NavigationButtons - відповідає за створення та функціонування кнопок “вперед” та “назад” і збереження історії переходів.

|  |
| --- |
| NavigationButtons.tsx |
| import \* as React from "react";  import IconButton from "material-ui/IconButton"  import ArrowBackIcon from "@material-ui/icons/ArrowBack"  import ArrowForwardIcon from "@material-ui/icons/ArrowForward"  import withStyles, { WithStyles } from "material-ui/styles/withStyles"  import createMuiTheme from "material-ui/styles/createMuiTheme"  import MuiThemeProvider from "material-ui/styles/MuiThemeProvider"  import red from "material-ui/colors/red"  const style = () => ({  container: {  marginLeft: "-15px"  },  button: {  "-webkit-app-region": "no-drag"  }  })  const theme = createMuiTheme({  palette: {  primary: {  main: "#ffffff"  }  }  })  type ComponentState = {  history: string[],  current: number  }  type ComponentProps = {  path: string,  onChangePath: (path: string) => void  }  type ComponentPropsWithStyle = ComponentProps & WithStyles<'button' | 'container'>  class NavigationButtons extends React.Component<ComponentPropsWithStyle, ComponentState>{  state = {  history: [this.props.path],  current: 0  }  private onNext() {  const { history, current } = this.state  this.props.onChangePath(history[current + 1])  }  private onPrev() {  const { history, current } = this.state  this.props.onChangePath(history[current - 1])  }  static getDerivedStateFromProps(nextProps: ComponentProps, prevState: ComponentState) {  if (nextProps.path == prevState.history[prevState.current]) return null  else if (nextProps.path == prevState.history[prevState.current + 1])  return {  current: prevState.current + 1  }  else if (nextProps.path == prevState.history[prevState.current - 1])  return {  current: prevState.current - 1  }  else {  let history = prevState.history.slice(0, prevState.current + 1)  history.push(nextProps.path)  return {  history,  current: prevState.current + 1  }  }  }  render() {  const { classes } = this.props  const { history, current } = this.state  return (  <MuiThemeProvider theme={theme}>  <div className={classes.container}>  <IconButton  color="primary"  className={classes.button}  onClick={() => { this.onPrev() }}  disabled={current == 0}  >  <ArrowBackIcon />  </IconButton>  <IconButton  color="primary"  className={classes.button}  onClick={() => { this.onNext() }}  disabled={current == history.length - 1}  >  <ArrowForwardIcon />  </IconButton>  </div>  </MuiThemeProvider>  );  }  }  export default withStyles(style)<ComponentProps>(NavigationButtons); |

14. TopBar - відповідає за компонування елементів NavigationButtons i WindowButtons на екрані.

|  |
| --- |
| TopBar.tsx |
| import \* as React from "react";  import AppBar from "material-ui/AppBar"  import Toolbar from "material-ui/Toolbar"  import withStyles, { WithStyles } from "material-ui/styles/withStyles"  import WindowButtons from "./WindowButtons"  import NavigationButtons from "./NavigationButtons"  const style = theme => ({  "appbar-container": {  ...theme.mixins.toolbar,  "z-index": 1000  },  appbar: {  "-webkit-app-region": "drag"  },  toolbar: {  display: 'flex',  'align-content': 'center'  },  empty: {  flex: 1  }  })  type ComponentProps = {  onChangePath: (path: string) => void,  path: string,  drawStrategy: any,  showHidden: boolean,  onDrawStrategyChanged: (drawStrategy: any, showHidden: boolean) => void  }  type ComponentPropsWithStyle = ComponentProps & WithStyles<'appbar' | 'toolbar' | 'empty' | 'appbar-container'>  class TopBar extends React.Component<ComponentPropsWithStyle>{  render() {  const { classes } = this.props;  return (  <div className={classes["appbar-container"]}>  <AppBar className={classes.appbar} position={"static"}>  <Toolbar className={classes.toolbar}>  <NavigationButtons onChangePath={this.props.onChangePath} path={this.props.path} />  <div className={classes.empty} />  <WindowButtons  drawStrategy={this.props.drawStrategy}  showHidden={this.props.showHidden}  onDrawStrategyChanged={this.props.onDrawStrategyChanged}  />  </Toolbar>  </AppBar>  </div>  );  }  }  export default withStyles(style)<ComponentProps>(TopBar); |

15. Класи ListDrawStrategy та IconDrawStrategy відповідають за різні типи відмальовки елементів файлів та папок в компоненті DirectoryBrowser (виступають в ролі класів-стратегій при реалізуванні патерну Стратегія)

16. DirectoryBrowser - клас, який відображає вміст поточної директорії та оновлює його за необхідністю. Приймає на вхід стратегію відмальовування (шаблон Стратегія) та поточну директорію. Стратегія відмальовування використовується при побудові елементів.

17. MainWindowView - основний клас рендерер-процесу. Представляє собою кореневий контейнер для всіх елементів та механізм передачі таких даних, як поточна директорія та стратегія відмальовування від одного об’єкта до всіх (шаблон Посередник).

**2. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ**

**2.1. Обґрунтування вибору та опис шаблонів проектування для реалізації програмного забезпечення автомату**

1. Адаптер.

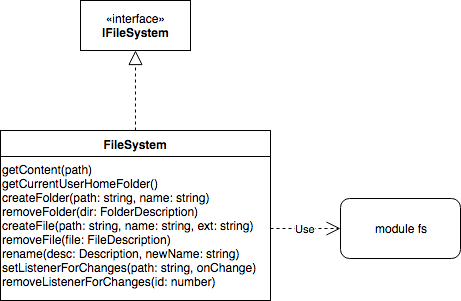
[](https://www.draw.io/#G1sayforDe7Sm9IL-6McEC_PYVtnifncS7)

Рис. 2.1.1: Структура використаного шаблону Адаптер

*Структурний шаблон проектування:* дозволяє об'єктам з несумісними інтерфейсами працювати разом.

*Структура:* Сервіс, який треба адаптувати: fs; інтерфейс, який потрібен для розробки даного ПО - IFileSystem; адаптер - клас FileSystem.

*Обгрунтування використання шаблону:* оскільки модуль fs розроблен за функціональною парадигмою програмування, його використання в даному проекті є незручним, оскільки складна асинхронна операція буде мати дуже велику кількість колбеків для виконання однієї операції. Адаптор в такому разі інкапсулює весь код взаємодії з цим модулем, даючи назовні дуже зручний інтерфейс, що заснований на використанні Promise-обʼєкта. Також, адаптер бере на себе перетворення даних з об’єкта класу fs.Stats в структуру Description, з якою зручно оперувати в контексті даного проекта.

2. Віртуальний проксі.

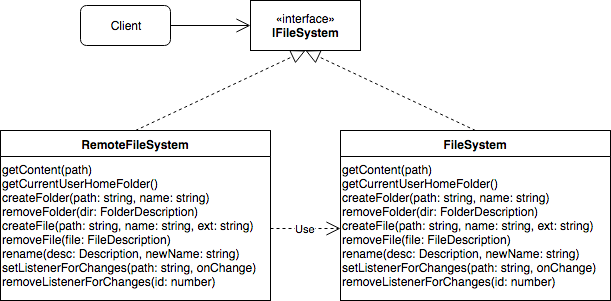
[](https://www.draw.io/#G1HAQ2mUUCkDFDuEHPhOp8EhG7NFmf1VGz)

Рис. 2.1.2: Структура використаного шаблона Віртуальний проксі

*Структурний шаблон проектування:* дозволяє підставляти замість реальних об'єктів спеціальні об'єкти-замінники. Ці об'єкти перехоплюють виклики до оригінального об'єкту для виконання певних операцій.

*Структура:* інтерфейс IFileSystem - спільний інтерфейс для заступника та заступаємого об’єкта; клас FileSystem - клас, до якого потрібно отримати доступ з іншого процесу виконяння; RemoteFileSystem - клас, який надає такий доступ.

*Обгрунтування використання шаблону:* деякі операції над файловою системою повинні робитися централізовано з одного об’єкта, а не з декількох окремих об’єктів. Для того, щоб досягти цієї цілі, і використовується цей шаблон. Будь-який виклик функцій об’єкта класу RemoteFileSystem перенаправляється в головний процес до об’єкта класу FileSystem, там обробляється, і відповідь буде направлена обратно.

Для зв’язку між цими класами використовуються об’єкти ipcRenderer в рендерер-процесі та ipcMain в головному процесі.

3. Спостерігач.

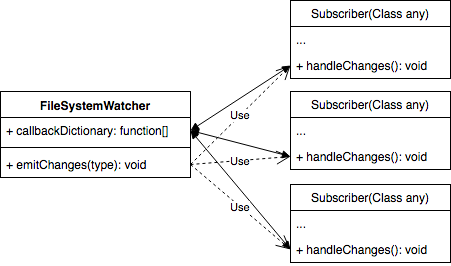
[](https://www.draw.io/#G1aBc52Pd88XHMIbQVjNKCoc3zBIYYjIBN)

Рис. 2.1.3: Структура використаного шаблона Спостерігач

*Поведінковий шаблон:* створює механізм підписки, що дозволяє одним об'єктам стежити і реагувати на події, що відбуваються в інших об'єктах.

*Структура шаблона:* підписник - обʼєкт будь-якого класу, якому цікаві зміни у конкретній директорії; при підписці передає функцію-обробника події; спостерігач - об’єкт класу FileSystemWatcher; при змінах у директорії викликає усі наявні функції-обробники.

*Обгрунтування використання шаблона:* однією із функціональних можливостей файлового менеждера повинне бути оновлення інформації при будь-які зміни в директорії, викликані як самим менеджером, так і будь-якими іншими процесами. Даний шаблон як-раз і надає необхідний функціонал для сповіщення про зміни у директорії.

4. Легковаговик

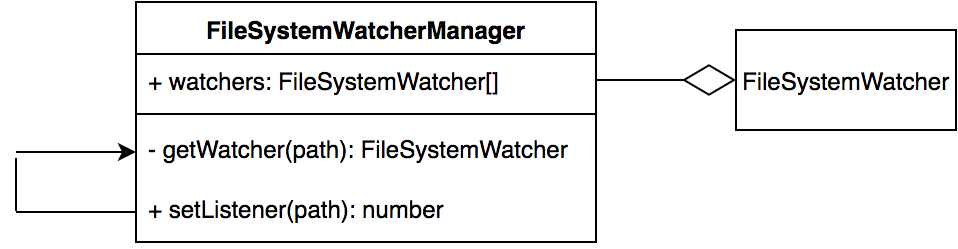
[](https://www.draw.io/?scale=2#G1OpHtwgSw5k8eIXppEDp74-95u-VVSgiZ)

Рис. 2.1.4: Структура використаного шаблона Легковаговик

*Структурний шаблон:* дозволяє вмістити більшу кількість об'єктів в відведену оперативну пам'ять.

*Структура шаблона:* об’єкт FileSystemWatcherManager містить в собі поле зі всіма існуючими об’єктами класу FileSystemWatcher. Коли приходить новий запрос на підпуски до директорії, цей клас дивиться, чи є створений для цієї директорії спостерігач. Якщо ні, то цей обʼєкт створюється. Далі підпистик добавляється до спостерігача.

*Обгрунтування використання шаблона:* даний шаблон використовується в даному контексті не з метою економії оперативної пам’яті, а з метою підвищення надійності програми. В поточній версії модуля fs, завдяки якому реалізовується робота класу FileSystemWatcher, є баг. Коли на одну директорію з одного процесу підписується декілька обробників подій, то ці події можуть не приходити жодному з них. Тому і було вирішено використовувати шаблон Легковик як шаблон, який гарантує, що однакових елементів не буде.

5. Стратегія

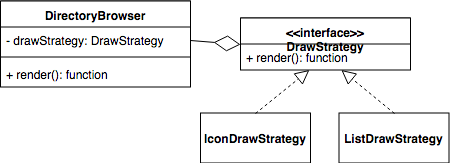
[](https://www.draw.io/#G1dFf-dIuiwQCsWfjuc_55tFidlweutx-z)

Рис. 2.1.5: Структура використаного шаблона Стратегія

*Поведінковий шаблон:* визначає сімейство схожих алгоритмів і поміщає кожен з них в власний клас, після чого алгоритми можна взаємозамінювати прямо під час виконання програми.

*Структура:* інтерфейс DrawStrategy визначає взаємний інтерфейс для алгоритмів відмальювання змісту директорій IconDrawStrategy i ListDrawStrategy.

*Обгрунтування використання шаблона:* користувач повинен мати спосіб вибирати, як будуть відображатися папки та файли. Даних шаблон дає нам можливість створювати взаємозамінні алгоритми та динамічно заміняти їх один на одного в процесі виконання програми.

6. Компонувальник

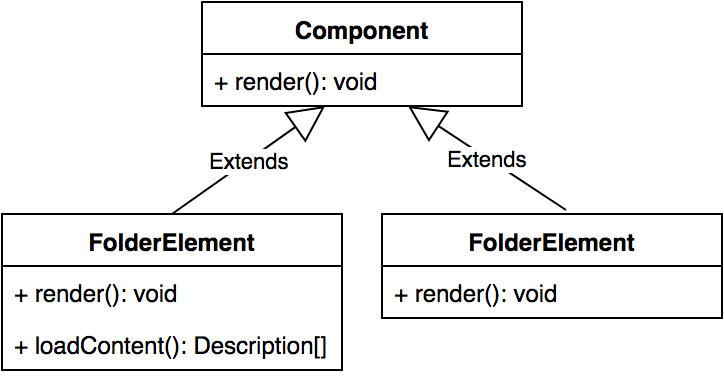
[](https://www.draw.io/?scale=2#G1wRxc2qKpkJbw720Nb7tOm3Le3ih6q1eH)

Рис. 2.1.5: Структура використаного шаблона Компонувальник

*Структурний шаблон:* дозволяє згрупувати безліч об'єктів в деревоподібну структуру, а потім працювати з нею так, як ніби це одиничний об'єкт.

*Структура шаблону:* абстрактний клас Component визначає загальний інтерфейс для простих і складових компонентів дерева. FileElement - це простий компонент дерева, що не має відгалужень. FolderElement - це складовий компонент дерева. Він містить набір дочірніх компонентів, але нічого не знає про їх типах. Це можуть бути як прості FileElement, так і інші FolderElement. Але це не є проблемою, якщо все дочірні компоненти слідують єдиному інтерфейсу.

*Обгрунтування використання шаблону:* згідно з функціональними вимогами, користувач мовинен мати можливість пересуватися по директоріях не тільки за допомогою відображеного вмісту директорії, а і за допомогою дерева папок файлової системи. В такому дереві при нажатті на папку, вона повинна розкритися і показати свій вміст. Якщо взяти FolderElement як контейнер, а FileElement як лист, то в нас якраз і виходить потрібна структура.

7. Посередник

*Поведінковий шаблон -* дозволяє зменшити зв'язаність множини класів між собою, завдяки переміщенню цих зв'язків в один клас-посередник.

*Структура шаблону -* в якості посередника виступає клас MainWindowView, оскільки він є батьківським елементом для всіх компонентів інтерфейсу. Компонентами, що зв’язані з посередником, є такі компоненти інтерфейсу, як NavigationButton, DirectoryBrowser, WindowButton, FolderElement.

*Oбгрунтування використання шаблону:* всі вище перечислені елементи залежать від таких даних, як поточний стиль відмальовки та поточна директорія, або від чогось з цього списку. При цьому, кожен з них повинен мати можливість змінювати ці дані. Щоб не робити багато заплутаних звʼязків між цими елементами, було вирішено використати інший об’єкт, який може їх зв’язати - MainWindowView.

**2.2 Результати роботи програми**

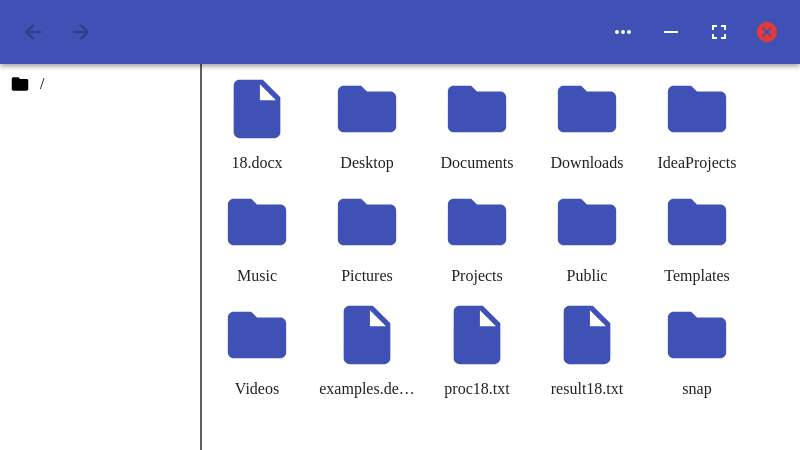


Рис. 2.2.1 Вікно програми при запуску

Для зміни стилю відображення елементів, треба нажати на три крапки в тулбарі та вибрати зі списка бажаний стиль

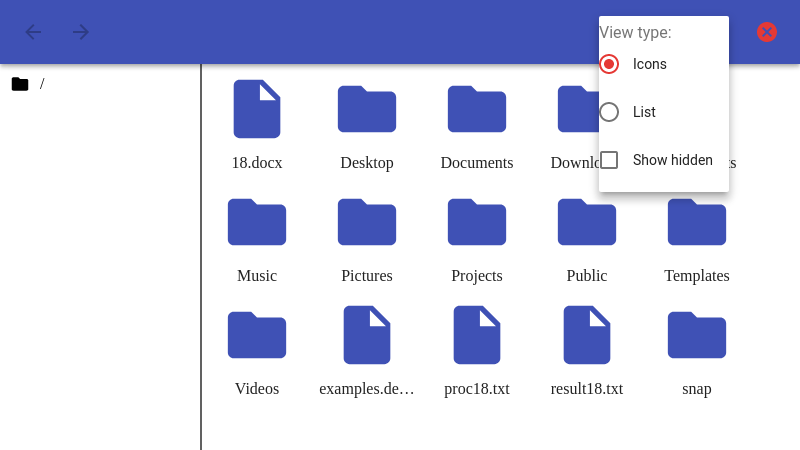


Рис. 2.2.2 Вибір стилю відображення

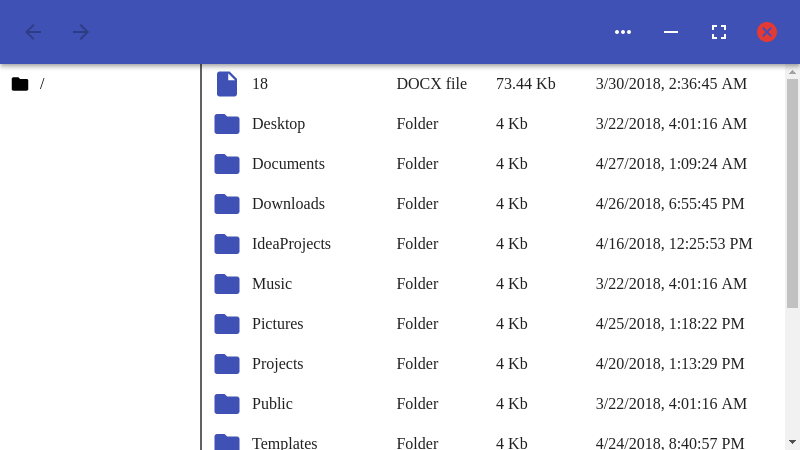


Рис. 2.2.3 Стиль відображення у вигляді списку

Для того, щоб бачити також і скриті файли і папки, треба в тому-ж меню поставити відповідну галочку.

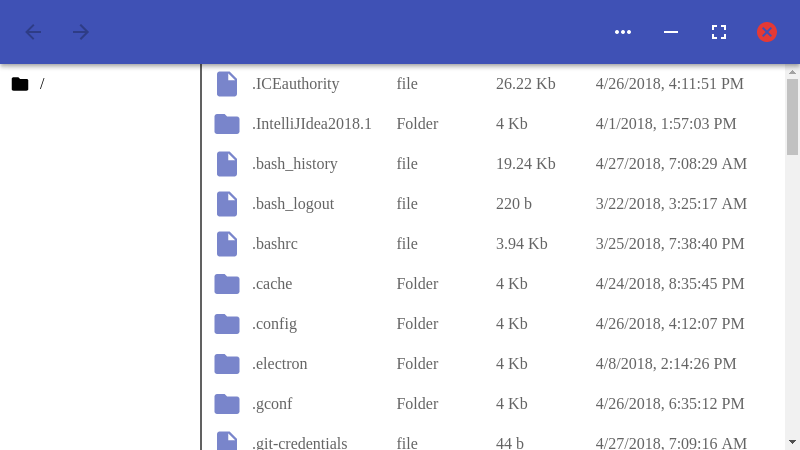


Рис. 2.2.4 Включений режим показу скритих файлів та папок

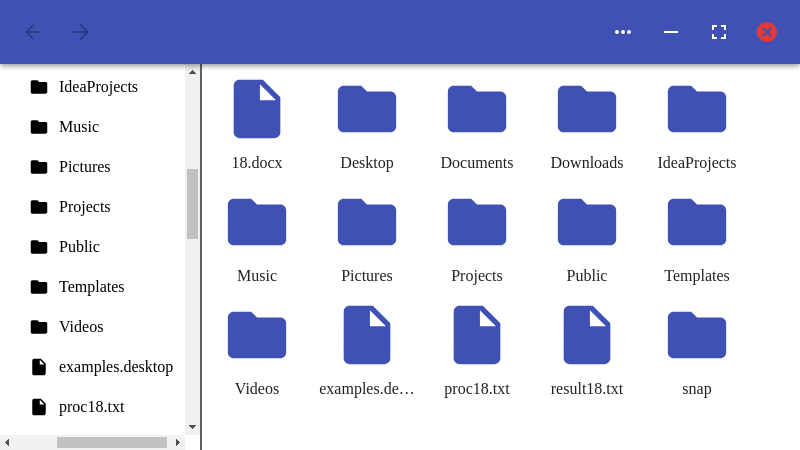


Рис. 2.2.5 Розкрите дерево папок

Для того, щоб відкрити файл, достатньо нажати на нього 2 рази.

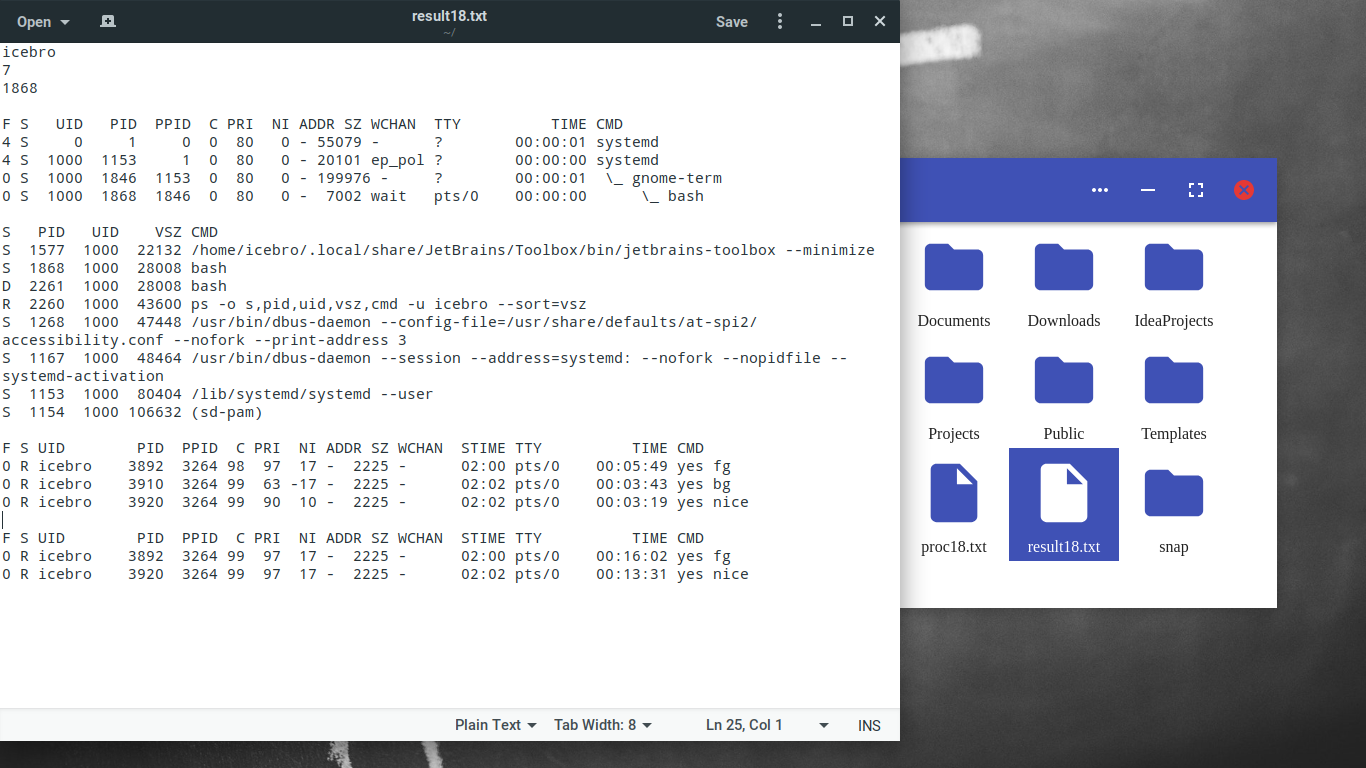


Рис. 2.2.6 Файл, відкритий за допомогою стандартної програми

**ВИСНОВКИ**

Метою даної курсової роботи було розроблення файлового менеджера з використанням шаблонів проектування. Підставою для розроблення стало завдання на виконання курсової роботи з дисципліни «Об’єктно-орієнтоване програмування» студентами ІІ курсу кафедри ПЗКС НТУУ «КПІ».

Для досягнення поставленої мети у повному обсязі виконано завдання, визначені у аркуші завдання на курсову роботу; розроблено графічні матеріали; реалізовано всі вимоги до програмного продукту, наведені у технічному завданні; створено відповідну документацію. Розроблене програмне забезпечення дозволяє користувачу проглядати файлову систему у графічному інтерфейсі, а також відкривати/запускати файли.

Програму створено на основі використання шаблонів проектування. Зокрема, до структури програмного забезпечення входить реалізація семи шаблонів, які належать до різних груп.

Для розроблення програмного забезпечення була використана мова програмування TypeScript, бібліотека Electron, середовище виконання Node.js. Також для побудови інтерфейсу програми мною використовувалися фреймворк React, технології HTML та CSS.

**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ**

1. Язык шаблонов. Города. Здания. Строительство. / Кристофер Вольфганг Александер. – 1977. – 1096 с.  
 2. Приёмы объектно-ориентированного проектирования. Паттерны проектирования / Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. – 1994. – 395 с.  
 3. Руководство Microsoft по проектированию архитектуры приложений. / С. Сомасегар, Скотт Гатри, Девид Хилл. – 2009. – 529 с.