МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ

ІМЕНІ ІГОРЯ СІКОРСЬКОГО”

Факультет прикладної математики

Кафедра програмного забезпечення комп’ютерних систем

Напрям підготовки 121 “Інженерія програмного забезпечення ”

**КУРСОВА РОБОТА**

з дисципліни «Об’єктно-орієнтоване програмування»

на тему

**Шаблони проектування в ООП. Електронний журнал відвідування**

Виконала студентка

ІІ курсу групи КП-81

Мозгова Катерина Олегівна

Керівник роботи

доцент, к. т. н. Заболотня Т.М.

Оцінка

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

(дата, підпис)

КИЇВ 2020

**ЗМІСТ**

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ ВСТУП 4

1. СТРУКТУРНО-АЛГОРИТМІЧНА ОРГАНІЗАЦІЯ ЕЛЕКТРОННОГО ЖУРНАЛУ ВІДВІДУВАННЯ 6

1.1. Модульна організація програми 6

1.2. Функціональні характеристики 8

1.3. Опис класів програми 10

2. ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ 22

2.1. Обгрунтування вибору та опис шаблонів проектування для   
 програмної реалізації системи обліку відвідувань студентів 22

2.2. Діаграма класів 34

2.3. Опис результатів роботи програми 35

ВИСНОВКИ 48

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 50

ДОДАТКИ 51

ДОДАТОК А.Текст коду програми 51

**ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ**

* **JavaScript (JS)** – динамічна, не типізована, об'єктно-орієнтована прототипна мова програмування. Реалізація стандарту ECMAScript.
* **Node JS** – інтерпретатор мови **JavaScript**.
* **Class** – у мові **JavaScript** різновид функції,шаблон для створення об’єктів та заповнення їх початковими даними, що підтримує об’єктно-орієнтовану парадигму програмування.
* **ПЗ** – програмне забезпечення
* **JSON** – текстовий формат даних для збереження JavaScript обʼєктів

ВСТУП

Дана курсова робота присвячена розробці електронного журналу відвідування за допомогою використання шаблонів проектування. Суть полягає у тому, що є потреба реалізації даного застосунку для полегшення роботи з паперовим журналом.

*Об’єктом* дослідження є процес автоматизації обліку відвідувань студента.

*Метою роботи* є розроблення електронного журналу відвідування 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.

Розроблена програма складається з 3-х модулей: модуля з функціональними класами створюваних об’єктів та користувачей, модуля з класами для взаємодій між об’єктами та їх коллекціями та модуль взаємодії з консольним інтерфейсом. Реалізовані шаблони проектування: Будівельник, Фабрика, Легковаговик, Прототип, Замісник, Посередник, Фасад, Репозиторій.

До функціональних можливостей програми належать: реєстрація та авторизація, збереження даних про користувачів, створення дисциплін, груп та відміток відвідування, запис даних у форматі JSON, наявність різноманітних прав доступу.

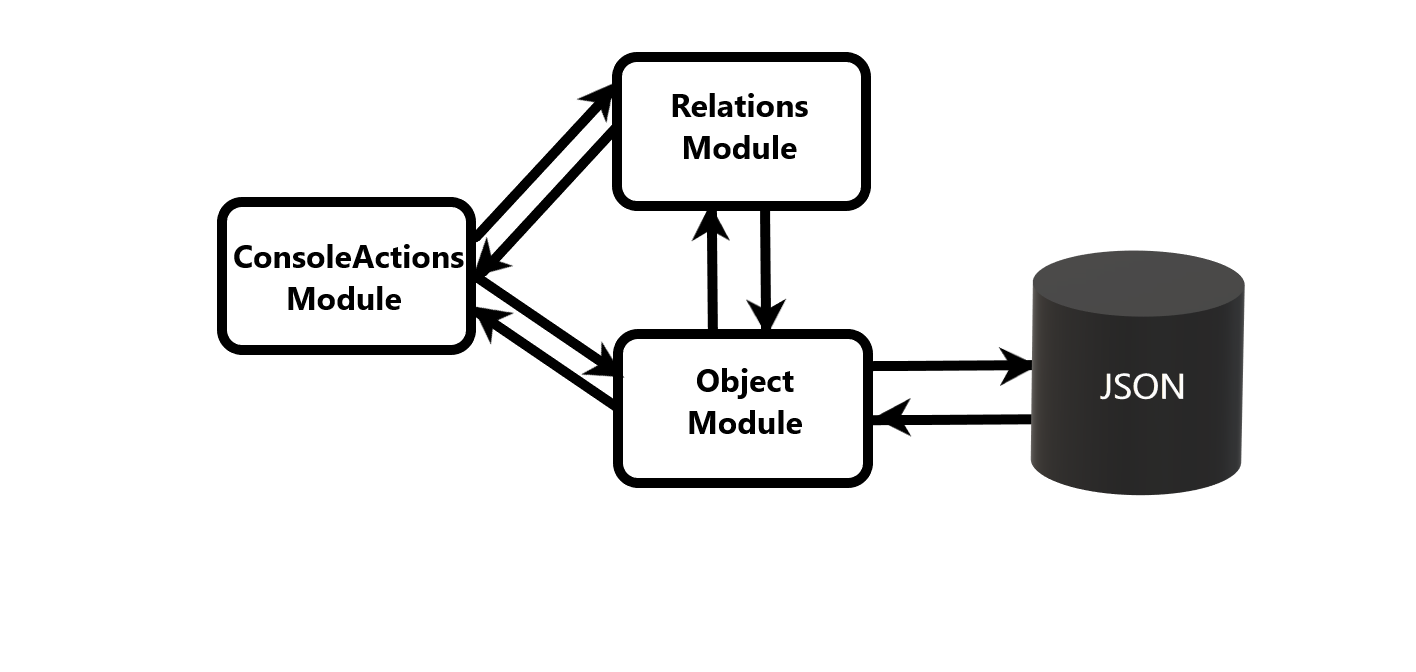
Для функціонування розробленої програми необхідно забезпечити наявність на комп’ютері щонайменше 21 МБ вільного дискового простору та наявність консолі.

**Пояснювальна записка складається зі вступу, двох розділів, загальних висновків, списку використаних джерел (4 з них електронний ресурс), одного додатку. Робота містить 60 рисунків. Загальний обсяг роботи – 64 друкованих сторінок, з них 50 сторінки тексту та 14 сторінки коду програми (тільки основні аспекти).**

**СТРУКТУРНО-АЛГОРИТМІЧНА ОРГАНІЗАЦІЯ**

**ЕЛЕКТРОННОГО ЖУРНАЛУ ВІДВІДУВАНЬ**

**1.1 Модульна організація програми**



*Рис. 1.1.1. Модульна організація програми*

У програмі можна виділити декілька основних логічних модулів, що мають певну самостійність і обмінюються один з одним даними. Модулям необов’язково відповідають конкретні класи чи ієрархії класів. У програмі є 3 основні модулі: модуль інтерфейсу та його керуванням (**Console Actions**), модуль створення та збереження об’єктів (**Object Module**), , що створює запити до JSON файлів (**JSON**) та модуль взаємодії обʼєктів між собою (**Relations Module**). Деякі з них містять використання додаткових бібліотек для запису та читання файлів. Взаємодія між модулями відбувається двома способами: за допомогою виклику методів одного із модулей.

Основним модулем є **Сonsole Actions.** Цей модуль визначає роботу програми в залежності від дій користувача. Коли користувач запускає програму, йому надається можливість до входу у систему(авторизація чи реєстрація). Після чого в залежності від прав доступу, що надаються за статусом користувача, йому надається вхід до головного меню, в якому можна вибрати одну із запропонованих дій з обʼєктами.. Завдяки ланцюжку взаємодії з двома іншими модулями та базою даних (**Console Actions** > **Relations Module** > **Object Module** > **JSON**), модуль **Console Actions** координує роботу усієї програми, і дає можливість користувачам працювати з іншими модулями, посилаючи відповідні команди.

**Relations Module** відповідає за створення викликів для взаємодії між такими класами як: User/Users, Subject/Subjects, Group/Groups та Check/Checks. Саме цей модуль забезпечує зв’язок між User та Subject, User та Group, User та Check.

**Object Module** приймає виклики із модуля **Relations.** Даний модуль забезпечує зв’язок між модулем **Console Actions** i файлами JSON. Такождає можливість створювати користувачів за трьома ролями, групи студентів, дисципліни та відмітки відвідування та редагувати їх.

**1.2. Функціональні характеристики**

Дана програма забезпечує зручну роботу з даними користувачів в залежності від їх ролі, що зберігаються у JSON форматі.

При запуску програми, користувач повинен авторизуватись або зареєструватись, інакше він не матиме доступ до ніякого функціоналу програми. Щоб його отримати він повинен вибрати варіант входу за допомогою вводу у консоль та створити аккаунт з унікальним нікнеймом, якого його ще немає у базі програми. Після реєстрації користувач зможе авторизуватися, використовуючи пароль та свою пошту як логін, які він вказав при реєстрації. Якщо користувач авторизується він потрапить в основне меню взаємодій з обʼєктами, де може вибрати одну з запропонованих дії.

**Далі детальний опис взаємодії з елементами програми**

Після запуску програму користувачу надається вибір з трьох команд: “Авторизуватись”, “Зареєструватись” та “Вихід”. Вибравши одну з наведених команд, користувачу надається можливість ввести свої дані та перейти до головного меню або вийти, завершивши програму. При реєстрації відбувається вибір ролі, а саме: Адміністратор, Вчитель чи Студент. При виборі ролі адміністратора, користувачу надається доступ до створення груп та дисциплін, при виборі ролі вчителя - до створення відміток, та вибору своєї дисципліни при створенні запису користувача, а при виборі ролі студента - до вибору своєї групи при створенні запису користувача та перегляду відміток і своєї групи.  
 У меню надається вибір команд, а саме: “Створити дисципліну”, “Створити групу”, “Запис відміток”, “Перегляд своєї групи” та “Перегляд своїх відміток”. Вибір відбувається за допомогою вибору номеру операції через консольний ввід. Відповідно до назви команди відбувається певний виклик функцій і через консоль відбувається вивід чи ввід інформації, щодо об’єкту, що фігурує у назві операції.

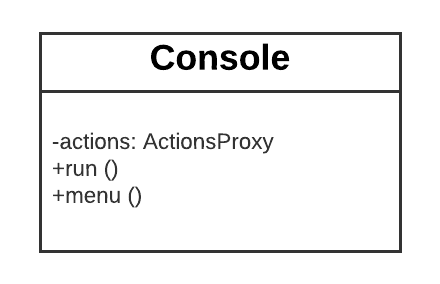
При виборі команди “Створити дисципліну”, якщо користувач має права адміністратора, надається можливість ввести назву дисципліни, та якщо її ще немає у базі, то відбудеться запис обʼєкту у JSON.

При виборі команди “Створити групу”, якщо користувач має права адміністратора, надається можливість ввести назву групи, та якщо її ще немає у базі, то відбудеться запис обʼєкту у JSON.

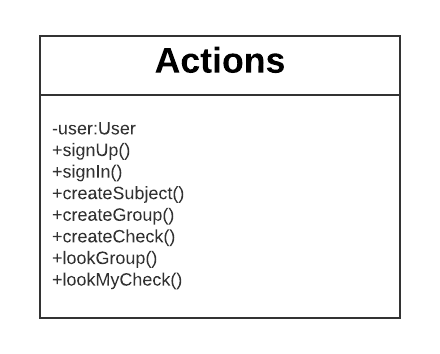
При виборі команди “Запис відміток”, якщо користувач має права вчителя, надається можливість ввести логін студента, та значення відміток, після чого відбувається запис обʼєкту у JSON.

При виборі команди “Перегляд своєї групи” або “Перегляд своїх відміток”, якщо користувач має права студента, за допомогою консольного виводу надається можливість перегляду своєї групи чи відміток відповідно, дані про які містяться у JSON.файлах.

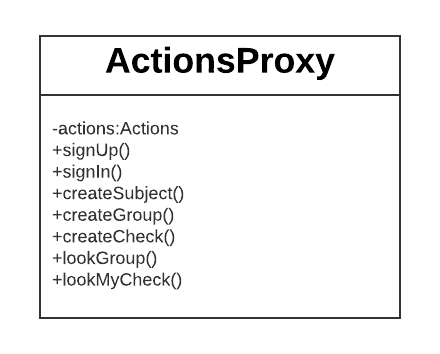
**1.3 Опис класів програми**

**1) Клас Console**

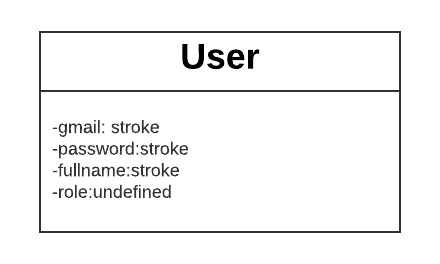
Клас, котрий є учасником шаблону “Фасад”. Містить метод run що виконується при запуску програми і дозволяє користувачу авторизуватись чи зареєструватись. Також містить метод menu, що викликається у попередньому методі і надає доступ до головного меню і дій в ньому.



**2) Клас Actions**

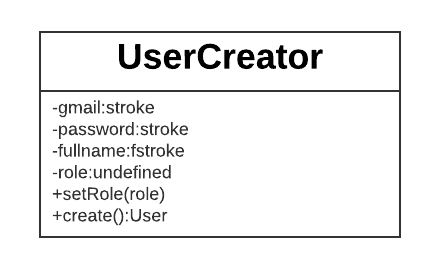
Клас що реалізує всі команди, що викликаються через методи run та menu класу Console за допомогою викликів з інших класів із модулів Relations та Object. Є учасником шаблону “Фасад” та “Замісник”.

**3) Клас Actions Proxy**

Клас - замісник класу Actions. Містить набір перевизначених методів класу Actions з надбудовою над ними перевірок ролі користувача.

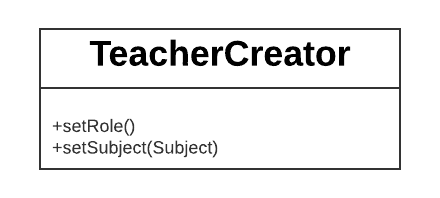
**4) Клас User**

З назви класу очевидно що це клас - конструктор користувача.

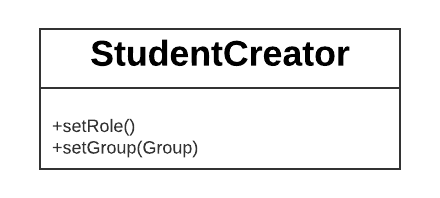


**5) Клас UserCreator**

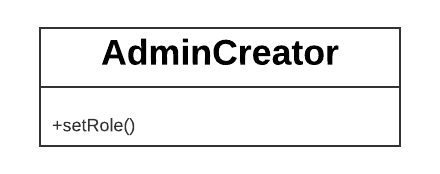
Абстрактний клас з шаблону “Будівельник” для створення користувача.

**6)Клас TeacherCreator**

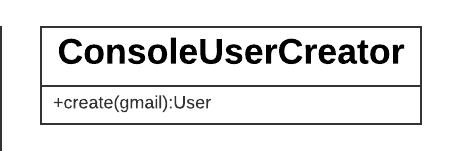
Дочірній клас класу UserCreator. Один із конкретних будівельників з шаблону “Будівельник”. Визначає конкретну роль користувача як вчитель та приймає назву дисципліни на додавання до властивостей користувача.

**7)Клас StudentCreator**

Дочірній клас класу UserCreator. Один із конкретних будівельників з шаблону “Будівельник”. Визначає конкретну роль користувача як студент та приймає назву групи на додавання до властивостей користувача.

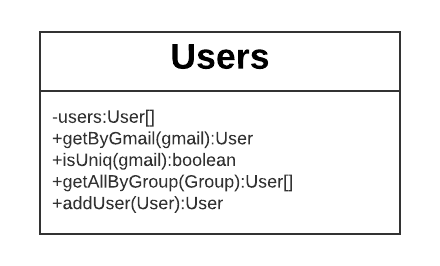
**8)Клас AdminCreator**

Дочірній клас класу UserCreator. Один із конкретних будівельників з шаблону “Будівельник”. Визначає конкретну роль користувача як адміністратор.

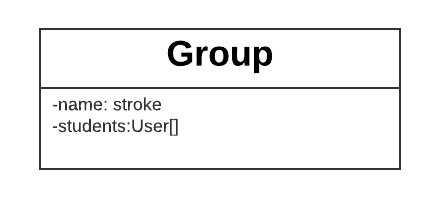


**9) Клас ConsoleUserCreator**

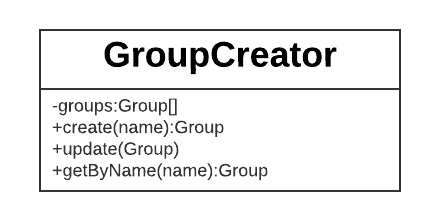
Клас - директор конкретних будівельників, що корегує за допомогою вводу користувача який з будівельників потрібно створити.

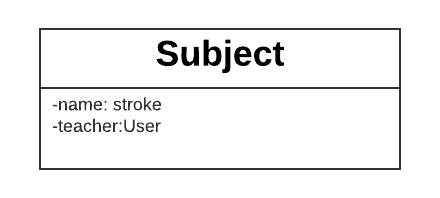


**10)Клас Users**Клас - репозиторій, що зберігає колекцію користувачів, взаємодіє з JSON для збереження і відтворення даних про користувачів. Містить методи для взаємодії з цією колекцією.

**11) Клас Group**

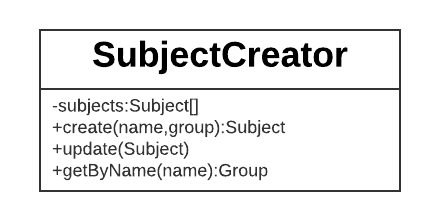
З назви класу очевидно що це клас - конструктор групи.

**12) Клас GroupCreator**

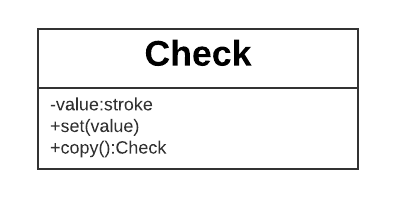
Клас, що э фабрикою легковаговиків(груп), що взаємодіє з JSON і не дає створюватись копіям вже створених груп з ціллю економії пам’яті. Також містить методи для взаємодії з колекцією груп.

**13) Клас Subject**

З назви класу очевидно що це клас - конструктор дисципліни.

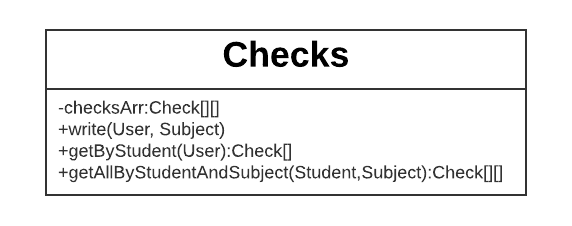
**14) Клас SubjectCreator**

Клас, що э фабрикою легковаговиків(дисциплін), що взаємодіє з JSON і не дає створюватись копіям вже створених дисциплін з ціллю економії пам’яті. Також містить методи для взаємодії з колекцією груп.



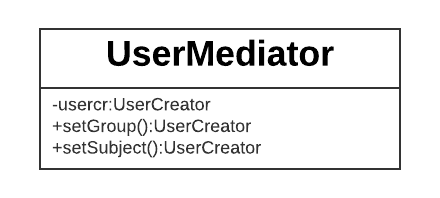
**15) Сheck**

Клас - прототип, для копіювання однотипних відміток про відвідування студентів.



**15) Сhecks**

Клас для взаємодії з колекціями відміток, зчитування та запису у JSON відповідних даних.



**17) User Mediator**

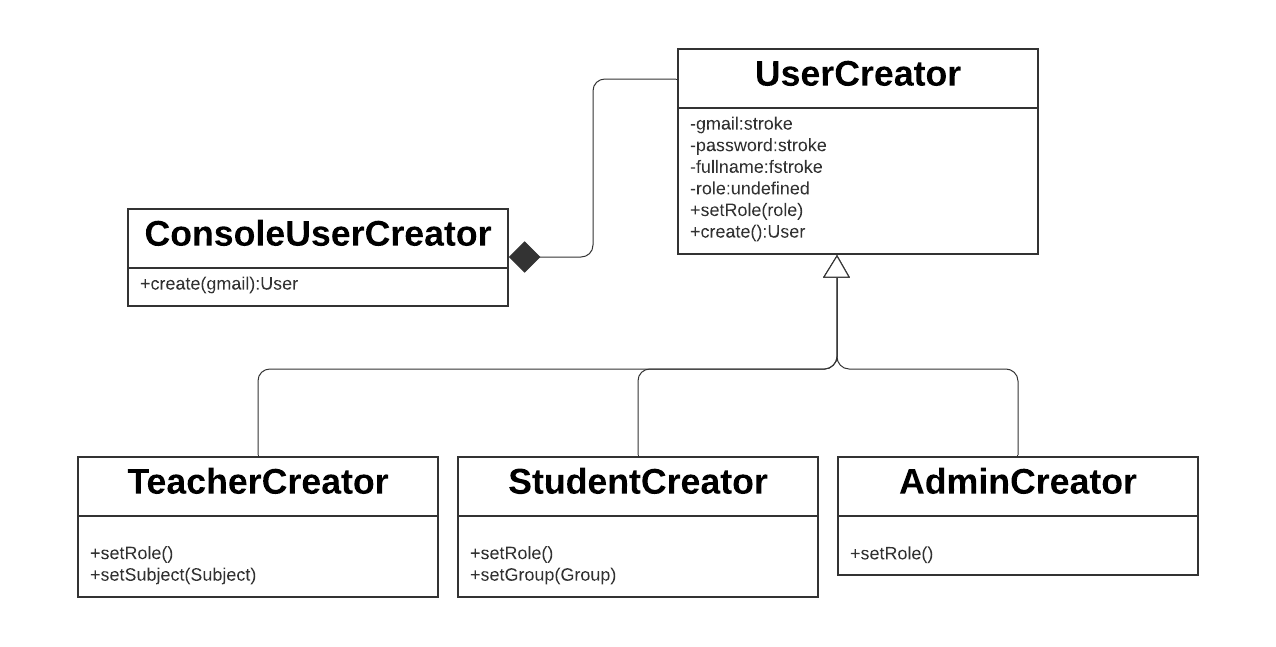
Kлас для встановлення зв’язків між такими класами як: User/Users, Group/Groups та Subject/Subjects. Є медіатором для цих класів. Викликається при створенні користувача з такими ролями як студент та вчитель для створення зв’язку з групою або дисципліною відповідно.

**ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ ЗА ДОПОМОГОЮ ШАБЛОНІВ ПРОЕКТУВАННЯ**

**2.1 Обгрунтування вибору та опис шаблонів проектування для   
 програмної реалізації системи електронного журналу відвідувань**

**1) Будівельник**

Шаблон проектування, відноситься до класу твірних шаблонів. На відміну від шаблону абстрактної фабрики і фабричного методу, ціль яких є застосування поліморфізму, задачею шаблону будівельника є забезпечення реалізації анти-шаблону телескопічного (багатоступеневого) конструювання. Оскільки клієнт повинен створювати складений об’єкт User, а при цьому процес створення об’єкта можна розділити на етапи то ми можемо використати шаблон будівельник.

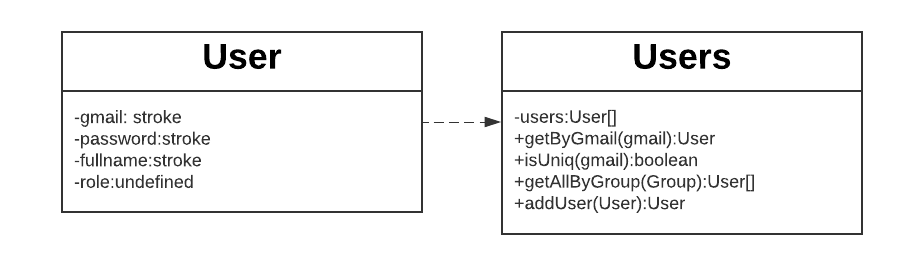


**UserCreator - Builder**

**TeacherCreator, StudentCreator, AdminCreator - ConcreteBuilder**

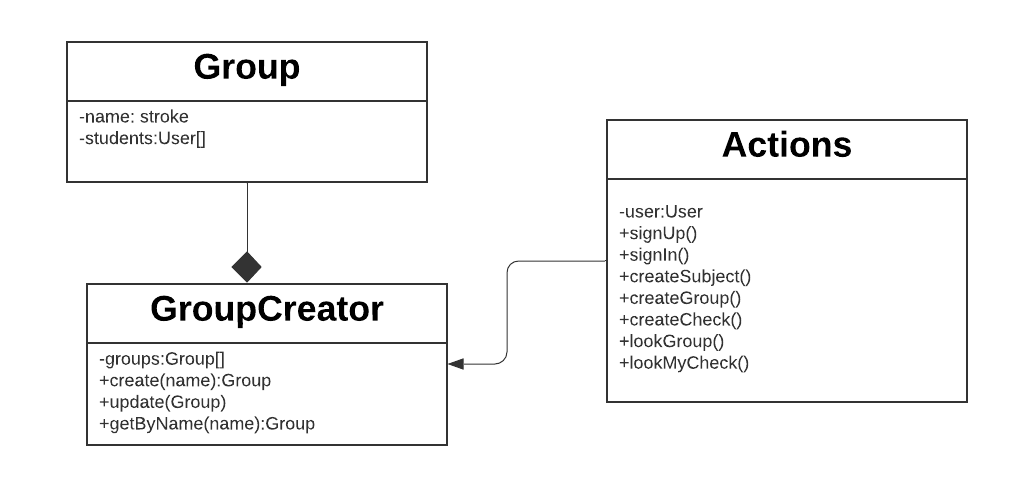
**ConsoleUserCreator - Director**

**2)Репозиторій**  
Repository є посередником між шаром області визначення і шаром розподілу даних, працюючи, як звичайна колекція об'єктів області визначення. Об'єкти-клієнти створюють опис запиту декларативно і направляють їх до об'єкта-сховища (Repository) для обробки. Так як ми маємо клас User, що потребує створення колекції ми можемо використати даний метод.

**Users - Repository  
User - Query Object**

**3) Легковаговик + Фабрика + Репозиторій**

Шаблон проектування, за яким створюється об'єкт, що мінімізує використання пам'яті розподіляючи стільки даних скільки це можливо між іншими подібними об'єктами; це спосіб використання великої кількості об'єктів, коли просте повторювання їх представлення зайняло б неприпустиму кількість пам'яті. Часто деякі частини, що зберігають стан об'єктів можуть бути однаковими, тобто як бачимо на прикладі ми маємо запис у JSON файлі колекції груп, доречно буде зеконимити пам’ять і не давати користувачеві створювати подібних груп, що вже були створені і записані у JSON . Для цього і використаємо шаблон Легковаговик. Так як є потреба у створенні однотипних обʼєктів, то легковик був зкомбінований з фабрикою та репозиторієм для збереження колекцій груп.



**Group - Flyweight**

**GroupCreator - Flyweight Fabric + Repository**

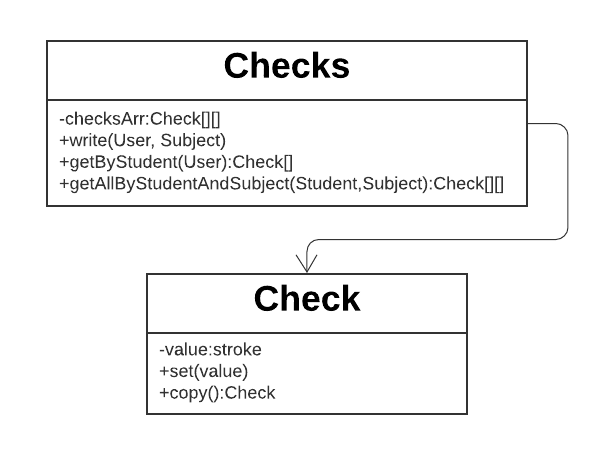
**Actions - Client**

**4)Прототип**

Шаблон, що задає види створюваних об’єктів

за допомогою екземпляра-прототипу і створює нові об’єкти шляхом копіювання цього прототипу.

Це єдиний шаблон з серії Породжуючих, котрий для створення нових об’єктів використовує не явне інстанціювання (ключове слово new), а клонування. Так як клас Check створює однотипні об’єкти, то для збереження памʼяті можемо використати даний шаблон і створювати обʼєкти шляхом копіюванням, так як створення об’єкта заново (new + встановлення значень полів) можна вважати досить “дорогим”.

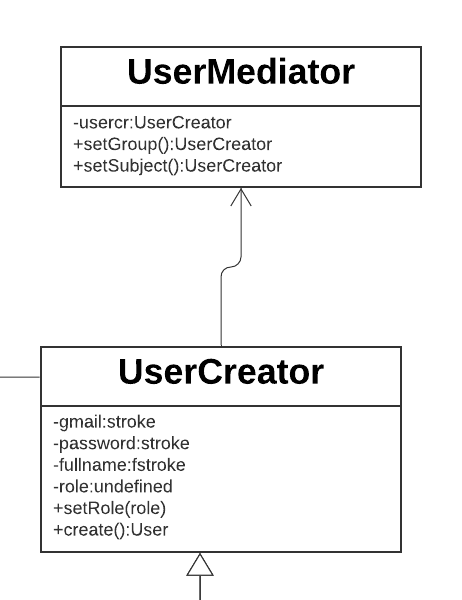


**Checks - Client**

**Check - Prototype + Concrete Prototype**

**5)Посередник**

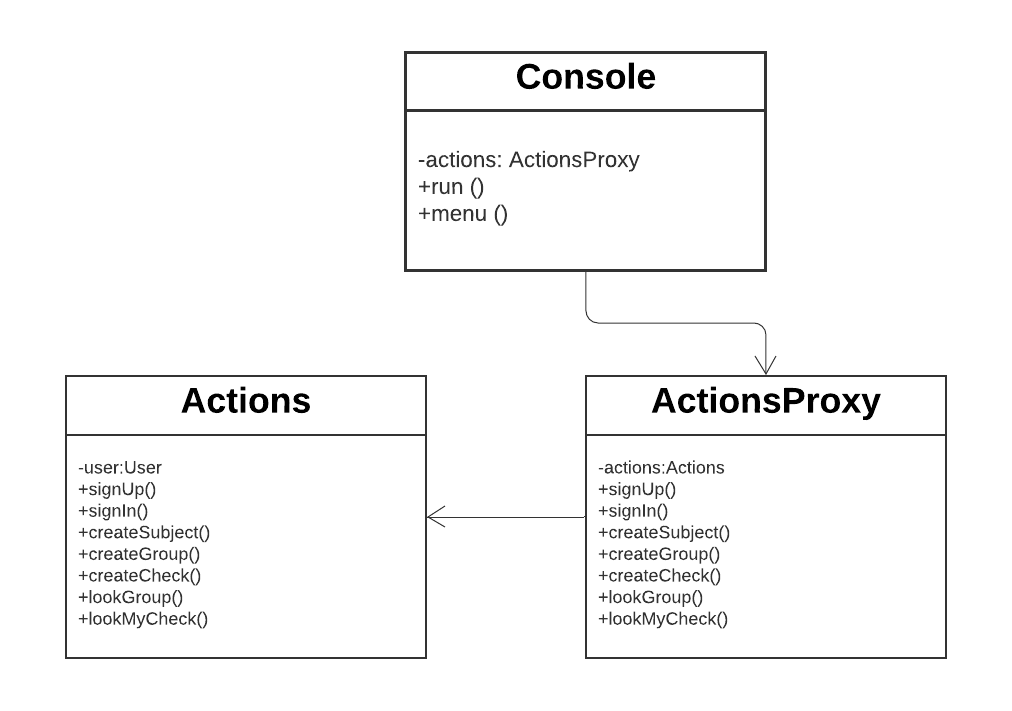
Поведінковий шаблон, який надає єдиний центр взаємодії певної групи об’єктів, які повинні бути взаємопов’язаними між собою. Визначає об'єкт, що інкапсулює спосіб взаємодії множини об'єктів. Так як існують об'єкти, зв'язки між котрими досить складні та чітко задані, а саме User/Users, Subject/Subjects та Group/Groups, то можемо використати даний шаблон



**UserMediator - Mediator**

**UserCreator - Colleague**

**6)Заступник**  
Надає об’єкт-заступник або об’єкт-замінник, обгортаючи доступ до реального компонента, «Заступник» зменшує складність роботи з ним. Так як є потреба у надані прав доступу до команд головного меню в залежності від прав доступу, то можемо використати даний шаблон.



**Console - Client**

**Actions -Real Subject**

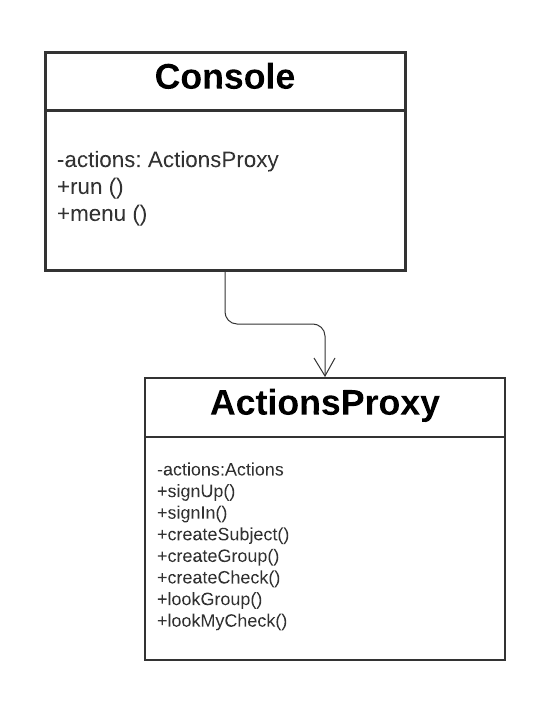
**ActionsProxy - Proxy**

**7)Фасад**

Це структурний шаблон проектування, який структурує об’єкти,

надаючи до них всіх доступ через єдиний шлюз. Надає єдиний,

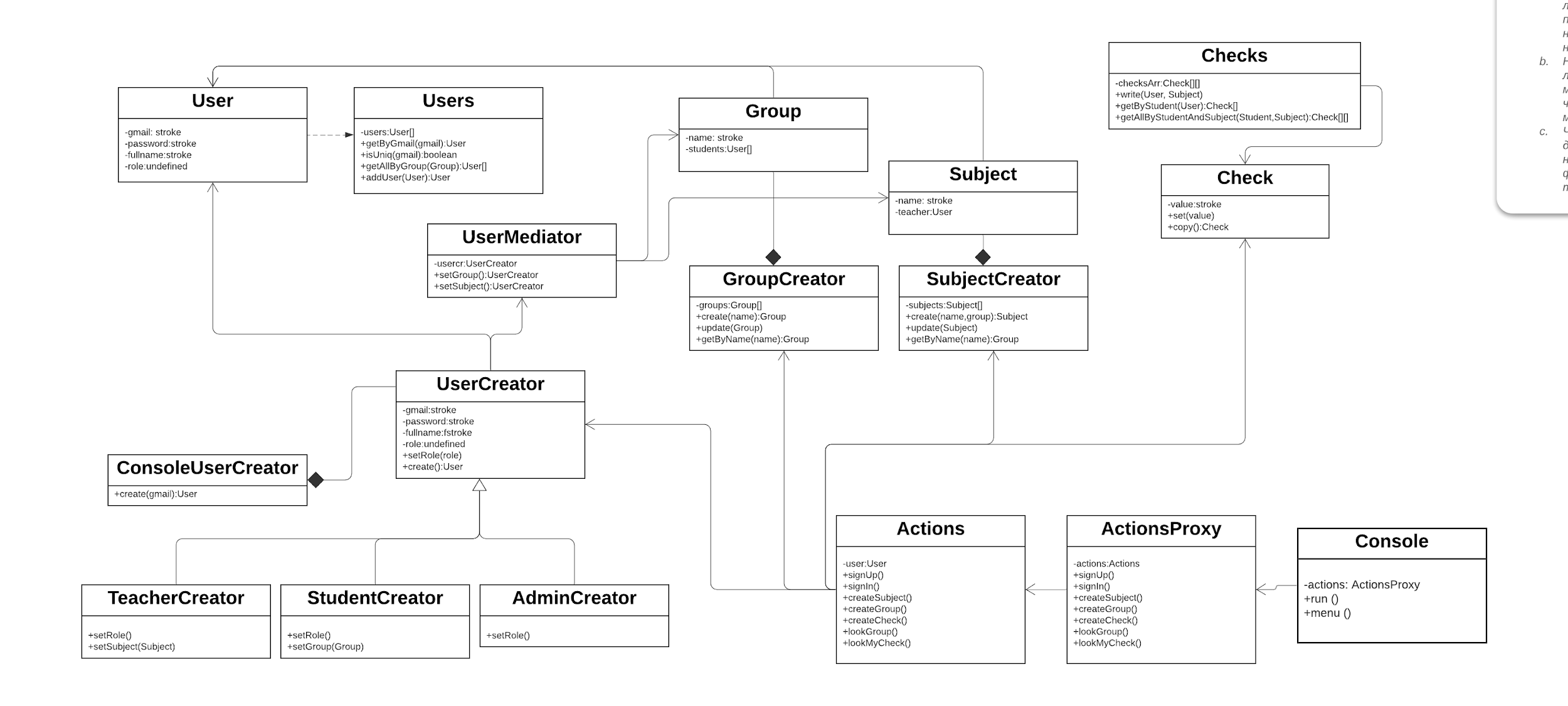
уніфікований інтерфейс до всієї підсистеми замість набору окремих та багаточисельних інтерфейсів. Фактично, “Фасад” визначає інтерфейс більш високого рівня, який спрощує використання системи. Оскільки ми маємо клас Actions з багатьма методами для взаємодії з обʼєктами, які треба використовувати за певних умов, що задає юзер через консольний інтерфейс, було використано шаблон “Фасад” у вигляді класу Console для надання єдиної функції для полегшення користувацького інтерфейсу програми.



**Console - Facade**

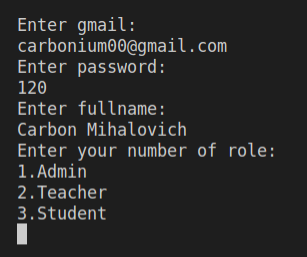
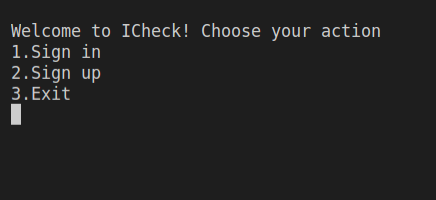
**ActionsProxy - object of Facade**

**2.2. Діаграма класів**



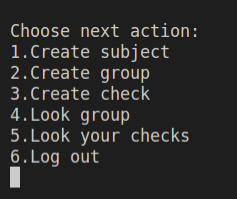
**2.3. Результат роботи програмного продукту**

Після запуску програми за допомогою команди node app.js користувач потрапляє у меню авторизациї:



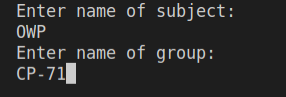
Так виглядає підменю реєстрації, що доступне після вибору команди Sign Up. Користувачу потрібно ввести свою пошту, пароль та ім’я, а також обрати свою роль у системі.

У випадку, якщо користувач вже був зареєстрований потрібно перейти у режим Sign In та ввести лише пошту та пароль.



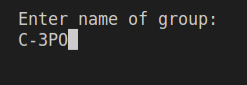
Після входу у систему користувачу доступно 6 операцій:

1)Створити предмет  
Команда доступна лише користувачам з роллю адміністратора. Для створення потрібно ввести назву дисципліни та існуючу групу, у якої вона викладається



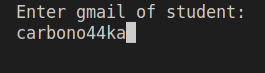
2)Створити групу

Команда доступна лише користувачам з роллю адміністратора. Для створення потрібно ввести назву групи.

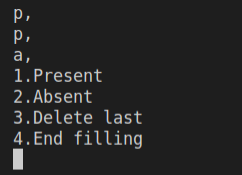


3)Створити відмітку

Команда доступна лише користувачам з роллю вчителя. Для створення відмітки потрібно ввести пошту студента, якому потрібно виставити відмітку

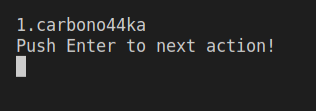


та вибрати один з варіантів заповнення: 1. Присутній 2. Відсутній 3. Видалити останню відмітку 4.Завершити ввід.



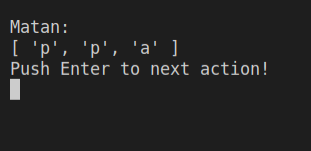
4)Подивитися групу

Команда доступна лише користувачам з роллю студента. При виконанні команди користувач може переглянути список своєї групи. Після натискання Enter користувач повернеться до головного меню.

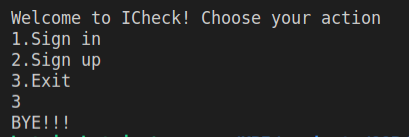


5)Подивитися свої відмітки

Команда доступна лише користувачам з роллю студента. При виконанні команди користувач може переглянути список своїх відміток. Після натискання Enter користувач повернеться до головного меню.



6)Завершити сеанс після чого користувач потрапить знову у меню реєстрації, вийти з якого можна вибравши опцію Exit, завершивши цим програму.



**ВИСНОВКИ**

Метоюданої курсової роботи було розроблення електронного журналу відвідування 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 ім. І.Сікорського».

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, програмного та апаратного забезпечення, наведені у технічному завданні; створено відповідну документацію.

Розроблення електронного журналу відвідування має важливе значення, оскільки на даний момент нема діджиталізованої версії журналу, а паперовий варіант досить вразливий до фізичних пошкоджень.. Саме тому зростає необхідність у розроблення програми, що може замінити паперовий носій та реалізувати усі його функції в своєму інтерфейсі. Розроблена програма реалізує базові операції. В процесі удосконалення даного програмного забезпечення можна перейти до онлайн версії журналу, та реалізувати програмне забезпечення на основі вже готової електронної версії.

Розроблене програмне забезпечення дозволяє створювати об’єкти користувачів, груп, дисциплін та відміток відвідування, забезпечує запис і відтворення інформації з JSON файлів.

Програму створено на основі використання шаблонів проектування: зокрема до структури розробленого програмного забезпечення входить реалізація семи шаблонів, які належать до різних груп шаблонів проектування (структурних, породжувальних та поведінкових). Особливу увагу слід приділити комбінуванню шаблонів, що покращує функціональність програми.

Для розроблення програмного забезпечення використані мова програмування JavaScript та Node.js.

Перспективним напрямком подальшого дослідження даної тематики є розроблення онлайн-версії спроектованої системи.

**СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ**

1. «Введение в паттерны» [Електронний ресурс]. <https://refactoring.guru/ru/design-patterns>
2. Документація TypeScript [Електронний ресурс] <https://www.typescriptlang.org/docs/home.html>
3. Документація Node.js [Електронний ресурс] <https://nodejs.org/uk/docs/>
4. Рефакторинг гуру, паттерны [Електронний ресурс] <https://refactoring.guru/ru/design-patterns>

**ДОДАТКИ**

**Додаток А. Текст коду програми**

|  |
| --- |
| **App.js** |
| **////////////////////ADD LIBS**  const readlinesync = require("readline-sync");  const fs = require("./addlibs/jsonparser.js");  ////////////////////MODELS  //////////////USERS  class User {  constructor(params) {  this.gmail = params.gmail;  this.password = params.password;  this.fullname = params.fullname;  this.role = undefined;  }  getInfo() {  console.log(``);  }  }  ////////BUILDER  class UserCreator {  constructor(gmail, password, fullname) {  let params = {};  params.gmail = gmail;  params.password = password;  params.fullname = fullname;  this.user = new User(params);  }  setRole(role) {  this.user.role = role;  }  create() {  return this.user;  }  }  class TeacherCreator extends UserCreator {  setRole() {  this.user.role = "teacher";  }  setSubject(subject) {  this.user.subject = subject;  }  }  class StudentCreator extends UserCreator {  setRole() {  this.user.role = "student";  }  setGroup(group) {  this.user.group = group;  }  }  class AdminCreator extends UserCreator {  setRole() {  this.user.role = "admin";  }  }  //Medi? Builder director  class ConsoleUserCreator {  // constructor(gmail){  // this.gmail = gmail;  // }  create(gmail) {  let password = readlinesync.question("Enter password:\n");  let fullname = readlinesync.question("Enter fullname:\n");  //let user = new UserCreator(gmail, password, fullname);  let r = readlinesync.question(  "Enter your number of role:\n1.Admin\n2.Teacher\n3.Student\n"  );  let user;  let usercreator;  switch (r) {  case "1": {  usercreator = new AdminCreator(gmail, password, fullname);  usercreator.setRole();  user = usercreator.create();  break;  }  case "2": {  usercreator = new TeacherCreator(gmail, password, fullname);  usercreator.setRole();  let medi = new UserMediator(usercreator);  usercreator = medi.setSubject();  user = usercreator.create();  break;  }  case "3": {  usercreator = new StudentCreator(gmail, password, fullname);  usercreator.setRole();  let medi = new UserMediator(usercreator);  usercreator = medi.setGroup();  user = usercreator.create();  break;  }  }  return user;  }  }  ////////REPOSITORY  class Users {  constructor() {  this.users = fs.readJsonFile("./data/users.json");  }  getAll() {  return this.users;  }  getByGmail(gmail) {  let users = this.getAll();  for (let user of users) {  if (user.gmail == gmail) {  return user;  }  }  return false;  }  isUniq(gmail) {  if (this.getByGmail(gmail)) return false;  else return true;  }  getAllByGroup(group) {  let users = this.getAll();  let usersByGroup = [];  for (let user of users) {  if ((user.group = group)) {  usersByGroup.push(user);  }  }  return usersByGroup;  }  deleteUser(gmail) {  let users = this.getAll();  for (let index in users) {  if (users[index].gmail === gmail) {  users.splice(index, 1);  }  }  this.users = users;  fs.writeJsonFile("./data/users.json", users);  }  addUser(user) {  if (this.isUniq(user.gmail)) {  let users = this.getAll();  users.push(user);  this.users = users;  fs.writeJsonFile("./data/users.json", users);  return user;  } else {  console.log("This user has been already added");  return user;  }  }  }  //////////////GROUPS  class Group {  constructor(name) {  this.name = name;  this.students = [];  }  }  //////////FACTORY + FLYWEIGHT + REPOSITORY  class GroupCreator {  constructor() {  this.groups = fs.readJsonFile("./data/groups.json");  }  create(name) {  let groups = this.groups;  let group = this.getByName(name);  if (group) {  console.log("This group has been already created");  return group;  }  group = new Group(name);  groups.push(group);  this.groups = groups;  fs.writeJsonFile("./data/groups.json", groups);  return group;  }  update(group) {  let groups = this.groups;  for (let index in groups) {  if (groups[index].name === group.name) {  groups[index] = group;  }  }  this.groups = groups;  fs.writeJsonFile("./data/groups.json", groups);  }  delete(name) {  let groups = this.groups;  for (let index in groups) {  if (groups[index].name === name) {  groups.splice(index, 1);  }  }  this.groups = groups;  fs.writeJsonFile("./data/groups.json", groups);  }  getByName(name) {  let groups = this.getAll();  for (let group of groups) {  if (group.name == name) {  return group;  }  }  return false;  }  getAll() {  return this.groups;  }  }  //////////////SUBJECTS  class Subject {  constructor(name, group) {  this.name = name;  this.teacher;  this.group = group;  }  }  //////////FACTORY + FLYWEIGHT + REPOSITORY  class SubjectCreator {  constructor() {  this.subjects = fs.readJsonFile("./data/subjects.json");  }  create(name, group) {  let subjects = this.subjects;  let subject = this.getByName(name);  if (subject) {  console.log("This subject has been already created");  return false;  }  subject = new Subject(name, group);  subjects.push(subject);  this.subjects = subjects;  fs.writeJsonFile("./data/subjects.json", subjects);  return subject;  }  update(subject) {  let subjects = this.subjects;  for (let index in subjects) {  if (subjects[index].name === subject.name) {  subjects[index] = subject;  }  }  this.subjects = subjects;  fs.writeJsonFile("./data/subjects.json", subjects);  }  delete(name) {  let subjects = this.subjects;  for (let index in subjects) {  if (subjects[index].name === name) {  subjects.splice(index, 1);  }  }  this.subjects = subjects;  fs.writeJsonFile("./data/subjects.json", subjects);  }  getByName(name) {  let subjects = this.getAll();  for (let subject of subjects) {  if (subject.name == name) {  return subject;  }  }  return false;  }  getAll() {  return this.subjects;  }  }  //////////////CHECKS  //////////PROTOTYPE  class Check {  constructor(value) {  this.value = value;  }  set(value) {  this.value = value;  }  copy() {  return new Check(this.value);  }  }  //////////FLYWEIGHT  class Checks {  constructor() {  this.checksArr = fs.readJsonFile("./data/allChecks.json");  }  write(student, subject) {  //let checksArr = this.checksArr;  let checks = this.getByStudentAndSubject(student, subject);  /////////////////////////////////////////////////////////  if (!checks) {  checks = {  student: student,  subject: subject,  values: []  };  }  let value = new Check("empty");  let t = true;  while (t) {  console.clear();  for (let value of checks.values) {  console.log(value.value + ", ");  }  let c = readlinesync.question(  "1.Present\n2.Absent\n3.Delete last\n4.End filling\n"  );  let val;  switch (c) {  case "1": {  value.set("p");  val = value.copy();  checks.values.push(val);  break;  }  case "2": {  value.set("a");  val = value.copy();  checks.values.push(val);  break;  }  case "3": {  checks.values.pop();  break;  }  case "4": {  t = false;  }  }  }  let temp = false;  let checksArr = this.checksArr;  for (let index in checksArr) {  if (  checksArr[index].name == checks.name &&  checksArr[index].student == checks.student  ) {  checksArr[index] = checks;  temp = true;  }  }  if (temp == false) {  checksArr.push(checks);  console.log(temp);  }  this.checksArr = checksArr;  fs.writeJsonFile("./data/allChecks.json", checksArr);  /////////////////////////////////////////////////////////  return checksArr;  }  delete(name) {  let allChecks = this.allChecks;  for (let index in allChecks) {  if (allChecks[index].name === name) {  allChecks.splice(index, 1);  }  }  this.allChecks = allChecks;  fs.writeJsonFile("./data/allChecks.json", allChecks);  }  getByStudentAndSubject(student, subject) {  let allChecks = this.getAll();  for (let checks of allChecks) {  if (checks.student == student && checks.subject == subject) {  return checks;  }  }  return false;  }  getByStudent(student) {  let allChecks = this.getAll();  let arr = [];  for (let checks of allChecks) {  if (checks.student == student) {  arr.push(checks);  }  }  //if ((arr[0] = undefined)) return false;  return arr;  }  getAll() {  return this.checksArr;  }  }  ////////////////////RELATIONS  //////////MEDIATOR  class UserMediator {  constructor(usercreator) {  this.usercr = usercreator;  }  setGroup() {  let groupcr = new GroupCreator();  let name = readlinesync.question("Enter your group name:");  let group = groupcr.getByName(name);  while (!group) {  console.log("This group does not exist!");  name = readlinesync.question("Enter your group name:\n");  group = groupcr.getByName(name);  }  //  this.usercr.setGroup(group.name);  //  group.students.push(this.usercr.user.gmail);  groupcr.update(group);  return this.usercr;  }  setSubject() {  let subjectcr = new SubjectCreator();  let sname = readlinesync.question("Enter your subject name:");  let subject = subjectcr.getByName(sname);  //  while (!subject) {  console.log("This subject does not exist!");  sname = readlinesync.question("Enter your subject name:\n");  subject = subjectcr.getByName(sname);  }  //  this.usercr.setSubject(subject.name);  //  subject.teacher = this.usercr.user.gmail;  subjectcr.update(subject);  return this.usercr;  }  }  //////////PROXY  class ActionsProxy {  constructor(actions) {  this.actions = actions;  }  signIn() {  this.actions.signIn();  }  signUp() {  this.actions.signUp();  }  createSubject() {  if (this.actions.user.role == "admin") {  this.actions.createSubject();  } else {  console.log("Not enough rights");  readlinesync.question("Push Enter to next action!\n");  }  } //admin  createGroup() {  if (this.actions.user.role == "admin") {  this.actions.createGroup();  } else {  console.log("Not enough rights");  readlinesync.question("Push Enter to next action!\n");  }  } //admin  createCheck() {  if (this.actions.user.role == "teacher") {  this.actions.createCheck();  } else {  console.log("Not enough rights");  readlinesync.question("Push Enter to next action!\n");  }  } //teacher  lookGroup() {  if (this.actions.user.role == "student") {  this.actions.lookGroup();  } else {  console.log("Not enough rights");  readlinesync.question("Push Enter to next action!\n");  }  } //student  lookMyCheck() {  if (this.actions.user.role == "student") {  this.actions.lookMyCheck();  } else {  console.log("Not enough rights");  readlinesync.question("Push Enter to next action!\n");  }  } //student  }  class Actions {  constructor() {  this.user = undefined;  }  signUp() {  console.clear();  let gmail = readlinesync.question("Enter gmail:\n");  //check gmail  let users = new Users();  while (!users.isUniq(gmail)) {  console.log("This gmail has been already registered");  gmail = readlinesync.question("Enter gmail:\n");  }  let usercr = new ConsoleUserCreator();  let user = usercr.create(gmail);  users.addUser(user);  console.log("You are registered!");  this.user = user;  return user;  }  signIn() {  console.clear();  let gmail = readlinesync.question("Enter gmail:\n");  //find by gmail  let users = new Users();  while (!users.getByGmail(gmail)) {  console.log(  "This gmail is not found in the system!\nChoose next action:"  );  let c = readlinesync.question("1.Enter gmail again\n2.Get back\n");  switch (c) {  case "1": {  gmail = readlinesync.question("Enter gmail:\n");  break;  }  case "2": {  return 0;  }  }  }  let user = users.getByGmail(gmail);  //check password  let password = readlinesync.question("Enter password:\n");  while (user.password != password) {  console.log("This password is not right!\nChoose next action:");  let c = readlinesync.question("1.Enter password again\n2.Get back\n");  switch (c) {  case "1": {  password = readlinesync.question("Enter password:\n");  break;  }  case "2": {  return 0;  }  }  }  this.user = user;  return user;  }  createSubject() {  console.clear();  let name = readlinesync.question("Enter name of subject:\n");  let subjectcr = new SubjectCreator();  let subject = subjectcr.getByName(name);  while (subject) {  console.log("Subject has been already created\nChoose next action");  let c = readlinesync.question("1.Enter name again\n2.Get back\n");  switch (c) {  case "1": {  name = readlinesync.question("Enter name of subject:\n");  subject = subjectcr.getByName(name);  break;  }  case "2": {  return 0;  }  }  }  ////  let gname = readlinesync.question("Enter name of group:\n");  let groupcr = new GroupCreator();  let group = groupcr.getByName(gname);  while (!group) {  console.log("Group not found\nChoose next action");  let c = readlinesync.question("1.Enter name again\n2.Get back\n");  switch (c) {  case "1": {  gname = readlinesync.question("Enter name of group:\n");  group = groupcr.getByName(group);  break;  }  case "2": {  return 0;  }  }  }  subjectcr.create(name, gname);  } //admin  createGroup() {  console.clear();  let name = readlinesync.question("Enter name of group:\n");  let groupcr = new GroupCreator();  let group = groupcr.getByName(name);  while (group) {  console.log("Group has been already created\nChoose next action");  let c = readlinesync.question("1.Enter name again\n2.Get back\n");  switch (c) {  case "1": {  name = readlinesync.question("Enter name of group:\n");  group = groupcr.getByName(name);  break;  }  case "2": {  return 0;  }  }  }  groupcr.create(name);  } //admin  createCheck() {  console.clear();  let checksArr = new Checks();  //  let gmail = readlinesync.question("Enter gmail of student:\n");  let users = new Users();  let user = users.getByGmail(gmail);  while (!user) {  console.log("This gmail is not found\nChoose next action");  let c = readlinesync.question("1.Enter gmail again\n2.Get back\n");  switch (c) {  case "1": {  gmail = readlinesync.question("Enter gmail of student:\n");  user = users.getByGmail(gmail);  break;  }  case "2": {  return 0;  }  }  }  ////  let name = this.user.subject;  let checks = checksArr.write(gmail, name);  } //teacher  lookGroup() {  console.clear();  let name = this.user.group;  let groupcr = new GroupCreator();  let group = groupcr.getByName(name);  let i = 1;  for (let user of group.students) {  console.log(`${i}.${user}`);  i++;  }  readlinesync.question("Push Enter to next action!\n");  } //student  lookMyCheck() {  console.clear();  let allcheck = new Checks();  let checks = allcheck.getByStudent(this.user.gmail);  //checks = checks.values;  let arr = [];  for (let check of checks) {  for (let value of check.values) {  arr.push(value.value);  }  console.log(check.subject+":")  console.log(arr);  }  readlinesync.question("Push Enter to next action!\n");  } //student  }  ////////////////////CONSOLE  ///////FACADE  class Console {  constructor(actions) {  this.actions = actions;  }  run() {  while (true) {  console.clear();  console.log("Welcome to ICheck! Choose your action");  let c = readlinesync.question("1.Sign in\n2.Sign up\n3.Exit\n");  let user;  switch (c) {  case "1": {  user = this.actions.signIn();  break;  }  case "2": {  user = this.actions.signUp();  break;  }  case "3": {  console.log("BYE!!!");  return 0;  }  }  this.menu();  }  }  menu() {  while (true) {  console.clear();  console.log("Choose next action:");  let c = readlinesync.question(  "1.Create subject\n2.Create group\n3.Create check\n4.Look group\n5.Look your checks\n6.Log out\n"  );  let user;  switch (c) {  case "1": {  this.actions.createSubject();  break;  }  case "2": {  this.actions.createGroup();  break;  }  case "3": {  this.actions.createCheck();  break;  }  case "4": {  this.actions.lookGroup();  break;  }  case "5": {  this.actions.lookMyCheck();  break;  }  case "6": {  console.log("LOG OUT");  return 0;  }  }  }  }  }  ////////////////////PROGRAM  let cons = new Console(new ActionsProxy(new Actions()));  cons.run(); |