

**Звіт**  
**Лабораторна робота №1а з ООП**  
**“Моделювання з використанням UML”**

виконав студент групи ІПС-21  
Тесленко Назар

# Design/Implementation Modeling

1. Обраним проєктом став невеличкий сайт, що дозволяв користувачу видаляти фон зображень і завантажувати їх на пристрої. Стек використаний у роботі: MERN(MongoDB, Express, React, Node.js), а також були використані Clerk та Clipdrop API. Сайт хоститься на Vercel ([link](#))
2. У початковій версії проєкту не було реалізації unit-tests, тому були написані unit tests для серверної частини (**server**), **і додатково реалізовані тести для БД** з майже ідеальним покриттям . Для реалізації тестування були використані: Jest, Babel

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered
All files	100	90.38	100	100	
configs	100	100	100	100	
appConfig.js	100	100	100	100	
controllers	100	86.66	100	100	
ImageController.js	100	100	100	100	
UserController.js	100	81.81	100	100	39-52
middlewares	100	100	100	100	
auth.js	100	100	100	100	
multer.js	100	100	100	100	
models	100	100	100	100	
userModel.js	100	100	100	100	
services	100	89.65	100	100	
imageService.js	100	100	100	100	
userService.js	100	88	100	100	32, 56, 91

3. Для проєкту були реалізовані наступні UML діаграми ([github](#)): для сценаріїв використання (**Use Case**), структури коду( **Class, Component, Deployment, Object**), логіки та поведінки програми (**Sequence, Activity**). Крім того, був реалізований [glossary](#) та [документація](#) (**хоститься на GitHub Pages**).
4. Були реалізовані зміни, як клієнтській частині, так і у серверній:

**Додатковий функціонал:** був створений новий функціонал - додана система кредитів для користувачів та спеціальний таймер для

коректного функціоналу. При реєстрації, кожен користувач отримує 5 кредитів, за які він може прибирати фон з зображень (1 фон - 1 кредит). Для нарахування нових кредитів був створений таймер з інтервалом (2 хвилини для тестового варіанту). Для коректного впровадження таймеру, була змінена модель користувача для БД, і тепер додатково воно зберігає час наступного кредиту, чи активований таймер та останнє оновлення кредитів.

### **client:**

- Був модифікований інтерфейс користувача для підтримки адаптивності на пристроях малих, середніх та великих розмірів

### **server:**

- Було прийняте рішення відокремити від контролерів (*ImageController.js*, *UserController.js*) всю бізнес-логіку та винести їх до *userService.js* та *imageService.js*. Це дозволило зробити логіку контролерів більш чіткішою, та повиносити окремі функції та компоненти для легких інтеграцій та повторних використань
  - Відповідно переписані **controllers** з використанням **services**
  - Створено новий файл конфігурації застосунку **appConfig.js**, який зберігає всі необхідні параметри та налаштування, для зручного централізованого доступу в проєкті (містить налаштування для системи кредитів, серверної частини, бази даних та API ключів), що робить вимушені зміни у коді максимально простими
  - Змінені unit tests для контролерів, та написані для сервісів
5. Для наглядного вигляду внесених змін, була додатково створена папка з [рефакторною версією проєкту](#).
  6. Загальна логіка програми не була видозміненою, але оскільки логіка контролерів була сепарована у окремі файли, були відповідно видозмінені unit tests як для контролерів так і для сервісів
  7. Оскільки проєкт не зазнав критичних змін логіки, загалом була більше реорганізація структури проєкту, то час виоконання залишився майже незмінним
  8. Загалом проєкт має досить гарну структуру, як з архітектурної точки зору, так і з загальних ООП принципів

### **Основи ООП:**

**Інкапсуляція:** неодноразово використана у різних частинах проєктної структури

- реалізована у services (userService, imageService). Сервісний шар приховує логіку реалізації від контролерів
- Моделі даних (userModel) інкапсулюють структуру бази даних
- Конфігураційні файли (appConfig) приховують налаштування системи
- Контролери інкапсулюють логіку обробки HTTP-запитів

**Наслідування:** у проєкті пряме класове наслідування майже не використовується, оскільки він побудований переважно на функціональній парадигмі. Замість цього застосовано підхід композиції, що дозволяє створювати більш гнучкі та повторно використовувані конструкції. У деяких частинах все ж простежується непряме наслідування, таке як через механізм middleware у Express.js, де параметри req, res, next передаються через ланцюжок викликів. Та й загалом, Express внутрішньо використовує прототипне наслідування, що хоч і неявно, але є частиною логіки фреймворку.

**Поліморфізм:** реалізований в обмеженому вигляді оскільки стилістика є більше функціональною, аніж класовою, але як неявним поліморфізмом, можна виділити обробку різних вебхук подій у UserController

### Більш загальні принципи:

- KISS (Keep It Simple, Stupid) — принцип реалізовано через розбиття коду на прості й зрозумілі частини: controllers, services, models. Логіку проєкту чітко розділено на клієнтську та серверну частини. Маршрути (routes) також структуровано відповідно до їхньої специфіки, що спрощує підтримку та масштабування коду.
- DRY (Don't Repeat Yourself) — принцип став основою рефакторингу: спільні дані винесено до централізованого конфігураційного файлу, а бізнес-логіку — з контролерів у відповідні сервіси. Також для уникнення дублювання коду використовуються middleware (наприклад, для авторизації та

обробки файлів), що забезпечує повторне використання функціоналу.

- YAGNI (You Aren't Gonna Need It) - реалізовано лише необхідний функціонал без додавання зайвого
- SOLID (загалом дотримано)
  - Single Responsibility Principle (SRP)  
Чіткий поділ на контролери та сервіси, кожен клас/модуль відповідає за конкретну функціональність. Приклади: UserController.js відповідає лише за обробку запитів користувачів, а userService.js - за бізнес-логіку користувача
  - Open/Closed Principle (OCP)  
Використання модульної структури, що дозволяє розширювати функціональність без зміни існуючого коду. Приклад: винесений конфігураційний файл, або ж сервіси, які легко підлягають розширенню
  - Liskov Substitution Principle (LSP)  
Як вже було зазначено раніше, що класичного наслідування не відбудеться
  - Interface Segregation Principle (ISP)  
Відсутнє через особливості JS
  - Dependency Inversion Principle (DIP)  
Залежності вводяться через імпорти, модулі високого рівня не залежать безпосередньо від модулів низького рівня

### Більш конкретні об'єктно-орієнтовані принципи:

- Coupling/Cohesion  
Функції та класи згруповані за функціональністю (userService, imageService) і мають мінімізовані залежності між модулями
- Law of Demeter (Принцип найменшого знання)  
Функції взаємодіють в основному з об'єктами, які безпосередньо пов'язані з ними, та загалом система API побудована так, що клієнт не потребує знання внутрішньої структури сервера

- Principle of Least Astonishment  
Загалом функції не мають непередбачуваного та перевантаженого функціоналу. Наявне послідовне найменування файлів і функцій, передбачувана поведінка API, послідовна обробка помилок

## — Патерни —

### **Архітектурні патерни (Architectural pattern)**

- **MVC (Model-View-Controller)**

Використаний даний патерн для розбиття програми на окремі взаємопов'язані компоненти: модель (дані), представлення (інтерфейс користувача) та контролер (бізнес-логіка)

Приклад:

- Model: файли в каталозі models (наприклад, userModel.js)
- View: клієнтська частина у каталозі client
- Controller: файли в каталозі controllers (наприклад, ImageController.js, UserController.js)

- **Middleware (в GoF (Behavioral)=> Chain of Responsibility)**

Функція або набір функцій, які перехоплюють, змінюють або контролюють потік обробки запиту між клієнтом і сервером у веб-додатках

Приклад:

Express middleware функції у проєкті

Також були реалізовані деякі GoF патерни, які можна окремо виділити у проєкті:

### **Створювальні патерни (Creational)**

- **Singleton (одинак)**

Забезпечує існування лише одного екземпляра класу і надає глобальну точку доступу до нього.

Приклад:

Підключення до бази даних MongoDB у configs/[mongodb.js](#)(connectDB())

- **Factory Method**

Дозволяє підкласам вирішувати, який об'єкт створити

Приклад:

Абстрагований процес створення користувачів у сервісі і може бути легко розширене для створення різних типів користувачів у майбутньому

```
// services/userService.js
export const createUser = async (userData) => {
  return await userModel.create(userData);
};
```

## ***Структурні патерни (Structural)***

- **Facade**

Надання простого інтерфейсу до складної системи

Приклад:

userService та imageService приховують усю логіку видалення фону та поміщають це все у виклик однієї функції removeImageBackground()

- **Adapter**

Дозволяє об'єктам з несумісними інтерфейсами працювати разом

Приклад:

У модулі imageService використовується патерн для адаптації зовнішнього ClipDrop API до внутрішнього інтерфейсу програми. Функція removeImageBackground адаптує отримані дані в потрібний формат (конвертує бінарні дані в base64), щоб їх можна було легко використовувати в інших частинах програми

## ***Поведінкові патерни (Behavioral)***

- **Strategy**

Дозволяє визначити збірку алгоритмів, та пов'язати їх навколо певної об'єктної частини. Цей патерн дозволяє змінювати алгоритм незалежно від клієнтів, які його використовують

**Приклад:**

Система управління кредитами користувача у userService.js. Залежно від стану користувача вибирається подальший алгоритм до оновлення кредитів

- **Observer**

Визначає залежність "один-до-багатьох" між об'єктами, так що коли один об'єкт змінює стан, всі залежні об'єкти отримують повідомлення та оновлюються автоматично

**Приклад:**

Вебхуки від Clerk. Система підписується на події Clerk (створення, оновлення, видалення користувача) і реагує відповідно до цих подій (система є "observer", який реагує на зміни стану в Clerk)

Загалом проєкт добре дотримується як архітектурних патернів та принципів, так і основних, і більш конкретних принципів ООП. Структура проєкту чітко поділена на client та server частини, що забезпечує зрозумілу і зручну організацію коду, а також полегшує подальшу масштабованість і підтримку. Використання принципів SOLID, DRY, KISS та інших дозволяє досягнути високої якості коду, зручності в його рефакторингу та тестуванні. Архітектурні патерни, такі як MVC, Observer, Middleware, забезпечують ефективну взаємодію компонентів і зручність у розробці. Всі частини проєкту взаємодіють згідно з мінімальними залежностями, що сприяє гнучкості й адаптивності системи в майбутньому.