Звіт
Лабораторна робота №2а з ООП

“Проєктування та розробка програм з використанням патернів проєктування”

виконав студент групи ІПС-21
Тесленко Назар

У даній лабораторній роботі було реалізовано патерни проєктування, які були використані в першій та третій лабораторних роботах. Ця робота є узагальненням і поєднанням раніше реалізованих підходів до проєктування програмного забезпечення з використанням патернів проєктування за GoF.

Посилання на лабораторні з реалізованими патернами:
- Лабораторна №2 (unit-tests, документація)
- Лабораторна №3 (unit-tests, документація)

Породжувальні

- **Builder Pattern(Будівельник)**

1. Лабораторна №3

Файли реалзіації GraphBuilder.h, GraphBuilder.cpp

```
class GraphBuilder {
private:
    int vertices = 0;
    int targetEdges = 0;
    double minWeight = 1.0;
    double maxWeight = 100.0;
    bool useRandomEdges = false;

    // For manual edges
    struct EdgeInfo {
        int from, to;
        double weight;
    };
    std::vector<EdgeInfo> manualEdges;

public:
    GraphBuilder& setVertices(int v);
    GraphBuilder& setTargetEdges(int edges);
    GraphBuilder& setWeightRange(double min, double max);
    GraphBuilder& useRandomGeneration();
    GraphBuilder& addManualEdge(int from, int to, double weight);

    std::unique_ptr<Graph> build();

private:
    void addRandomEdgesToGraph(Graph& graph);
    void addManualEdgesToGraph(Graph& graph);
};
```

Надалі використовуємо у Menu.cpp

```
void Menu::createGraph() {
    int vertices = getIntInput("Enter number of vertices: ");
    int edges = getIntInput("Enter number of edges: ", 0, vertices * (vertices - 1));

    GraphBuilder builder;
    builder.setVertices(vertices).setTargetEdges(edges);
```

```
        builder.setWeightRange(minWeight, maxWeight).useRandomGeneration();
    }
```

Використано для створення об’єкта графа з гнучко налаштованими параметрами. Побудова графа може вимагати значної кількості параметрів — кількість вершин, цільова кількість ребер, діапазон ваг ребер, спосіб генерації (випадковий чи ручний), тощо. Замість створення кількох перевантажених конструкторів або складної логіки в одному методі, Builder дозволяє поетапно і гнучко формувати об’єкт.

- **Factory Pattern (Фабрика)**

1. Лабораторна №3

Файли реалізації: AlgorithmFactory.h, AlgorithmFactory.cpp

```
// usage of Factory Pattern
class AlgorithmFactory {
public:
    enum class AlgorithmType {
        BELLMAN_FORD,
        DIJKSTRA
    };

    static std::unique_ptr<ShortestPathAlgorithm> createAlgorithm(AlgorithmType type);
    static std::unique_ptr<ShortestPathAlgorithm> createAlgorithm(const std::string& name);
    static std::unique_ptr<DijkstraAlgorithm> createDijkstra() {
        return std::make_unique<DijkstraAlgorithm>();
    }
};
```

```
std::unique_ptr<ShortestPathAlgorithm> AlgorithmFactory::createAlgorithm(AlgorithmType type) {
    switch (type) {
    case AlgorithmType::BELLMAN_FORD:
        return std::make_unique<BellmanFordAlgorithm>();
    case AlgorithmType::DIJKSTRA:
        return std::make_unique<DijkstraAlgorithm>();
    default:
        throw std::invalid_argument("Unknown algorithm type");
    }
}
```

Використовується в JohnosonAlgorithm.cpp

```
JohnsonAlgorithm::JohnsonAlgorithm()
    :bellmanFord(AlgorithmFactory::createAlgorithm(AlgorithmFactory::AlgorithmType::BELLMAN_FORD)),
    dijkstra( AlgorithmFactory::createAlgorithm(AlgorithmFactory::AlgorithmType::DIJKSTRA)){}
```

Патерн **Factory** реалізовано для створення об’єктів різних алгоритмів пошуку найкоротших шляхів — зокрема, Bellman-Ford та Dijkstra. Factory Pattern дозволяє інкапсулювати процес створення об’єктів і забезпечити єдину точку доступу до логіки створення, зберігаючи принцип Open/Closed.

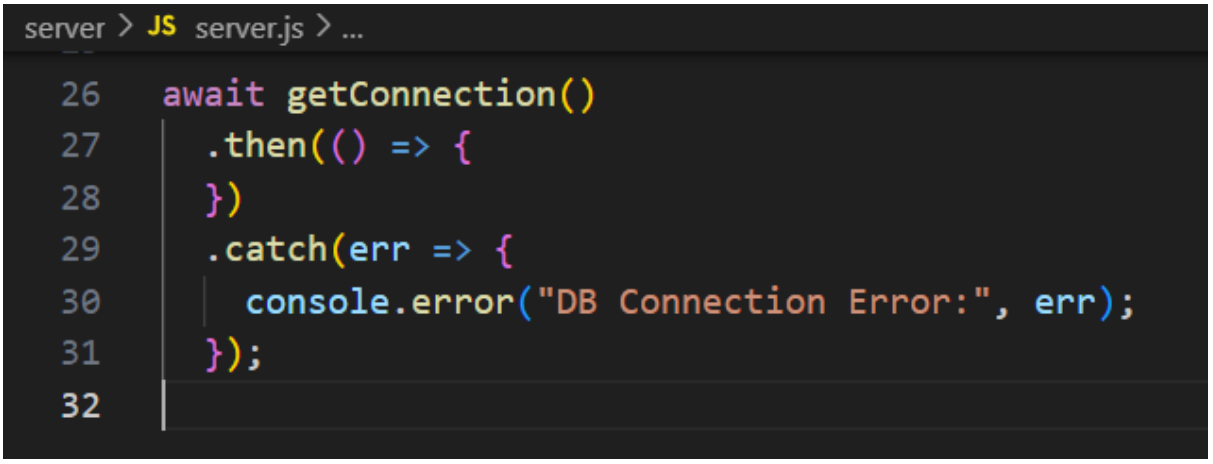
- **Singleton (Одинак)**

1. Лабораторна №2

Файли реалізації: server/configs/mongodb.js

server > configs > JS dbConnection.js > getConnection

```
1   import mongoose from "mongoose";
2   import config from "./appConfig.js";
3
4   let connection = null;
5   let isConnecting = false;
6
7   export const getConnection = async () => {
8       if (connection && mongoose.connection.readyState === 1) {
9           return connection;
10      }
11
12      if (isConnecting) {
13          // If a connection attempt is in progress, wait until it completes
14          while (isConnecting) {
15              await new Promise(resolve => setTimeout(resolve, 100));
16          }
17          return connection;
18      }
19
20      isConnecting = true;
21
22      try {
23          await mongoose.connect(`${config.db.uri}/${config.db.name}`, config.db.options);
24          connection = mongoose.connection;
25          console.log("Database connected");
26          return connection;
27      } finally {
28          isConnecting = false;
29      }
30  };
```



Патерн Singleton призначений для забезпечення існування лише одного екземпляра певного об’єкта протягом життєвого циклу застосунку. У цьому випадку цей об’єкт — з’єднання з базою даних MongoDB. Для коректної реалізації цього патерну файл підключення до БД був адаптований відповідно до ідеології Singleton. При кожному виклику функції відбувається перевірка наявності існуючого підключення: якщо воно вже встановлене і активно, функція повертає цей інстанс, інакше відбувається ініціалізація нового з’єднання.

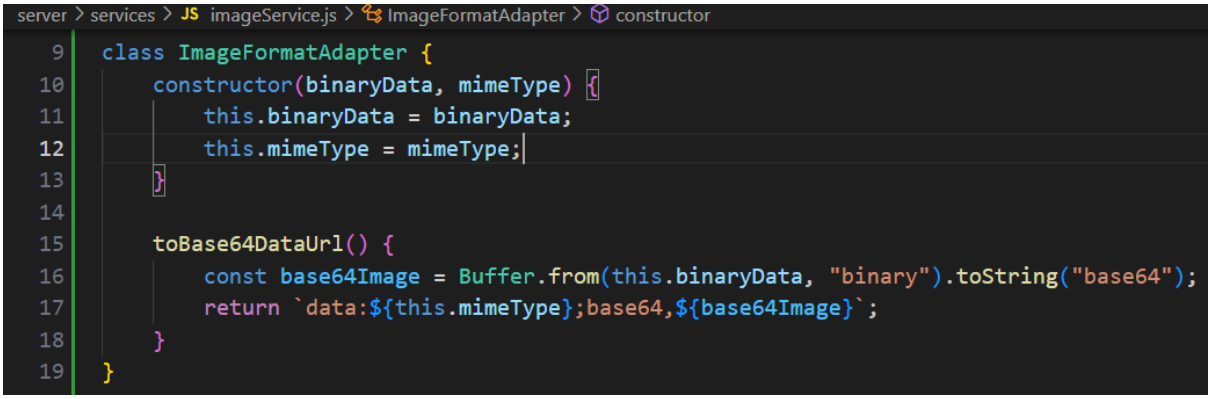
Хоча реалізація і не використовує класичний класовий стиль, вона повністю відповідає основним принципам патерну Singleton, а саме: створенню єдиного глобального екземпляра і наданню контролю за його ініціалізацією.

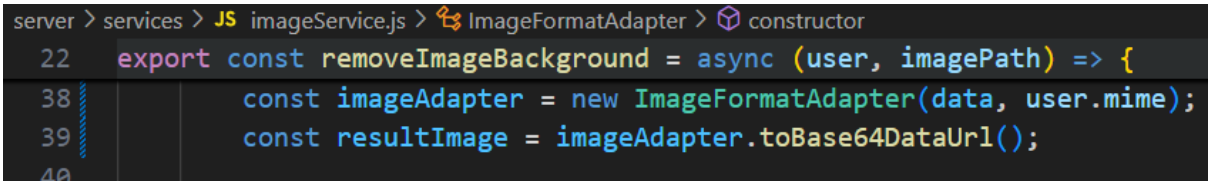
Структурні

- **Adapter Pattern (Адаптер)**

1. Лабораторна №2

Файли реалізації:server/services/imageService.js



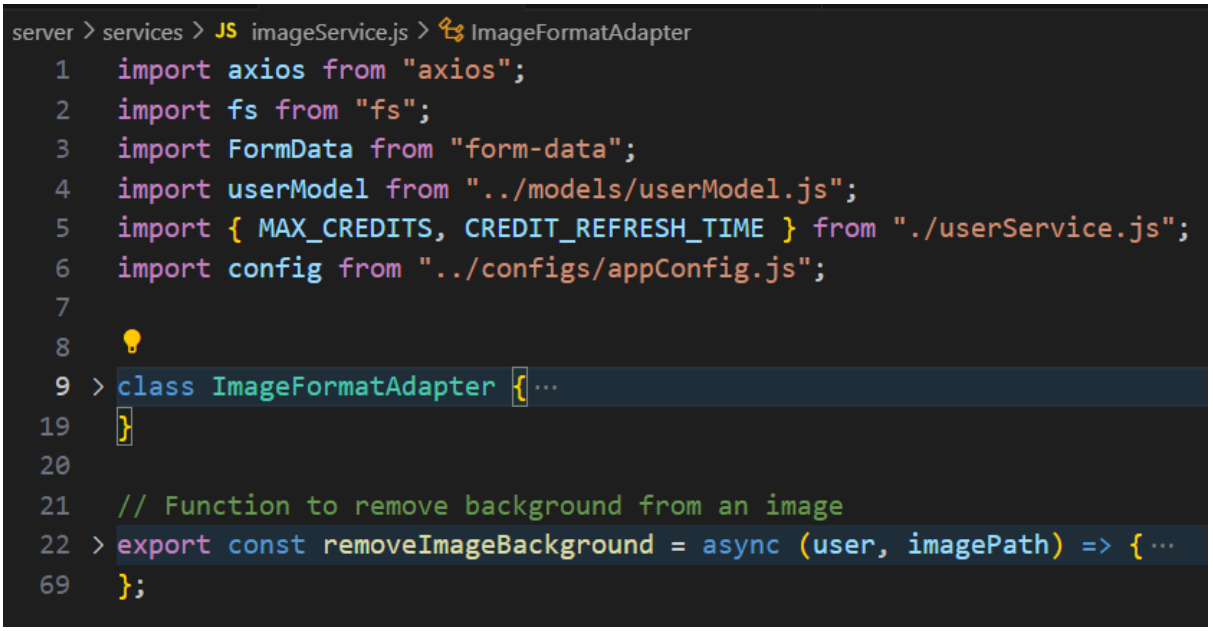


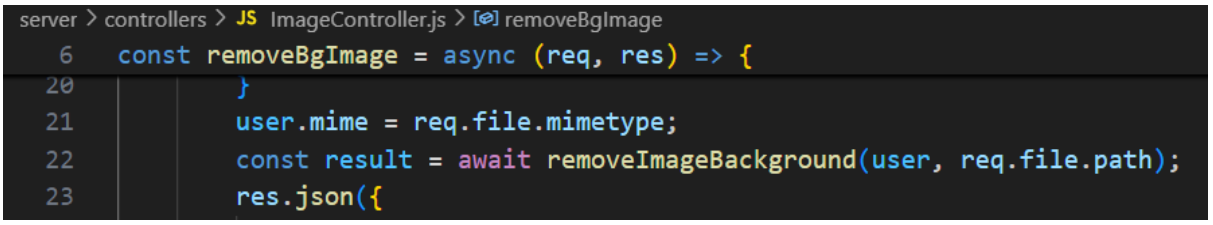
У проєкті реалізовано патерн Adapter через клас ImageFormatAdapter, який інкапсулює логіку перетворення низькорівневих двійкових даних (отриманих із зовнішнього API) у формат Base64 Data URL.
Адаптація дозволяє узгодити «несумісний» формат (arraybuffer/binary) з форматом, очікуваним для подальшого відображення в інтерфейсі або передачі, що й відповідає класичній ідеї патерну Adapter — перетворити інтерфейс одного об'єкта до вигляду, який очікує інший клієнт.

- **Facade Pattern (Фасад)**

1. Лабортаорна №2

Файли реалізації: server/controllers, server/services





У проєкті використано патерн Facade для спрощення взаємодії з підсистемою обробки зображення. Функція removeImageBackground виступає в ролі фасаду: вона інкапсулює всі кроки, пов’язані з відправленням зображення до стороннього API, перетворенням отриманих даних та оновленням інформації про користувача. Завдяки цьому контролер removeBgImage взаємодіє лише з одним методом, не знаючи про внутрішню реалізацію. Це відповідає принципам патерну Facade — надати єдиний інтерфейс до складної системи.

Поведінкові

- **Strategy Pattern (Стратегія)**

1. Лабораторна №3

Файли реалізації:

- ShortestPathAlgorithm.h
- ShortestPathAlgorithm.cpp

```cpp
// usage of Strategy Pattern
class ShortestPathAlgorithm {
public:
    virtual ~ShortestPathAlgorithm() = default;
    virtual std::vector<double> findShortestPaths(const Graph& graph, int source) = 0;
    virtual std::string getName() const = 0;
};

class BellmanFordAlgorithm : public ShortestPathAlgorithm {
public:
    std::vector<double> findShortestPaths(const Graph& graph, int source) override;
    std::string getName() const override { return "Bellman-Ford"; }
    bool hasNegativeCycle(const Graph& graph, int source);
};

class DijkstraAlgorithm : public ShortestPathAlgorithm {
private:
    struct Compare {
        bool operator()(const std::pair<double, int>& a, const std::pair<double, int>& b) {
            return a.first > b.first; // min-heap
        }
    };

public:
    std::vector<double> findShortestPaths(const Graph& graph, int source) override;
    std::string getName() const override { return "Dijkstra"; }
};
```

```cpp
class JohnsonAlgorithm : public AlgorithmSubject {  // Inherit from AlgorithmSubject
private:
    std::unique_ptr<ShortestPathAlgorithm> bellmanFord;
    std::unique_ptr<ShortestPathAlgorithm> dijkstra;
```

Реалізовано для інкапсуляції різних алгоритмів пошуку найкоротших шляхів (Bellman-Ford та Dijkstra) за єдиним інтерфейсом ShortestPathAlgorithm. Це дозволяє використовувати конкретні стратегії незалежно від контексту, у якому вони виконуються — наприклад, усередині алгоритму Джонсона.

- **Observer Pattern (Спостерігач)**

1. Лабораторна робота №3

Файл реалізації: Observer.h, ConcreteObservers.h

```cpp
// Observer interface
class AlgorithmObserver {
public:
    virtual ~AlgorithmObserver() = default;

    // Events for algorithm execution
    virtual void onAlgorithmStart(const std::string& algorithmName) = 0;
    virtual void onAlgorithmFinish(const std::string& algorithmName, double executionTime) = 0;
    virtual void onProgressUpdate(const std::string& algorithmName, int current, int total) = 0;
    virtual void onNegativeCycleDetected() = 0;
    virtual void onStepCompleted(const std::string& stepName) = 0;
};
```

```cpp
// Subject interface
class AlgorithmSubject {
private:
    std::vector<std::shared_ptr<AlgorithmObserver>> observers;

public:
    virtual ~AlgorithmSubject() = default;

    void addObserver(std::shared_ptr<AlgorithmObserver> observer) {
        observers.push_back(observer);
    }

    void removeObserver(std::shared_ptr<AlgorithmObserver> observer) {
        observers.erase(
            std::remove_if(observers.begin(), observers.end(),
                [&observer](const std::weak_ptr<AlgorithmObserver>& obs) {
                    return obs.lock() == observer;
                }),
            observers.end()
        );
    }
```

Відповідно створені спостерігачі:

```cpp
// Console logger observer
class ConsoleObserver : public AlgorithmObserver {
public:
    void onAlgorithmStart(const std::string& algorithmName) override {
        std::cout << "[INFO] Starting " << algorithmName << " algorithm..." << std::endl;
    }

    void onAlgorithmFinish(const std::string& algorithmName, double executionTime) override {
        std::cout << "[INFO] " << algorithmName << " completed in "
            << std::fixed << std::setprecision(2) << executionTime << " ms" << std::endl;
    }
```

```cpp
// Progress bar observer
class ProgressBarObserver : public AlgorithmObserver {
private:
    int lastPercentage = -1;

public:
    void onAlgorithmStart(const std::string& algorithmName) override {
        lastPercentage = -1;
```

Реалізовано для реагування на події під час виконання алгоритмів без зміни їхньої логіки. Алгоритм Джонсона надсилає сповіщення про свій стан (початок, завершення, прогрес, кроки тощо), а спостерігачі (ConsoleObserver, ProgressBarObserver) реагують на ці події незалежно. Можна легко додати нові типи спостерігачів без зміни коду алгоритму.

2. Лабораторна №2

Файли реалізації: server/controllers/userController.js

server > controllers > JS UserController.js > ...

```
18  const clerkWebhooks = async (req, res) => {
32          switch (type) {
33              case "user.created": {
34                  const userData = {
35                      clerkId: data.id,
36                      email: data.email_addresses[0].email_address,
37                      firstName: data.first_name,
38                      lastName: data.last_name,
39                      photo: data.profile_image_url || data.image_url
40
41                  }
42                  console.log("Creating user with data:", userData);
43                  await createUser(userData)
44
45                  break;
46              }
47              case "user.updated": {
48                  const userData = {
49                      email: data.email_addresses[0].email_address,
50                      firstName: data.first_name,
51                      lastName: data.last_name,
52                      photo: data.profile_image_url || data.image_url
53                  }
54                  await updateUser(data.id, userData);
55                  res.json({})
56                  break;
57              }
58              case "user.deleted": {
59                  await deleteUser(data.id);
60                  res.json({})
61                  break;
62              }
63
64          }
65          res.json({});
```

Хоча у проєкті не реалізовано повноцінний патерн Observer у класичному вигляді, взаємодія з сервісом Clerk через webhook-и частково відповідає його ідеї. Контролер clerkWebhooks виступає в ролі спостерігача, який реагує на зовнішні події (створення, оновлення або видалення користувача) та виконує відповідні дії. Таким чином, система обробляє події в режимі підписки, що є характерною рисою патерну Observer.

- **Chain of Responsibility(Ланцюжок відповідальності)**

1. Лбаораторна №2

Файл реалзіації: server/routes/imageRoutes.js

Хоча Express не реалізує патерн Chain of Responsibility буквально, сама структура imageRouter.post(...), де запит проходить через послідовність middleware-функцій (upload.single, authUser, removeBgImage), відповідає основним принципам цього патерну. Кожна функція може або обробити запит, або передати його далі. Таким чином, можна сказати, що у проєкті застосовано адаптовану форму патерну Chain of Responsibility.

Висновок:

У рамках даної лабораторної роботи було реалізовано та застосувано низки патернів проєктування з категорій породжувальних, структурних і поведінкових. Зокрема:

- Породжувальні патерни (**Builder, Factory, Singleton**) було використано для створення об'єктів із гнучкою конфігурацією, централізованого керування створенням екземплярів та забезпечення єдиного доступу до ресурсів (наприклад, підключення до БД).
- Структурні патерни (**Adapter, Facade**) забезпечили узгодження несумісних інтерфейсів та спростили взаємодію з підсистемами, дозволяючи приховати складну логіку за єдиним інтерфейсом.
- Поведінкові патерни (**Strategy, Observer, Chain of Responsibility**) надали гнучкість у виборі алгоритмів, забезпечили реакцію на події без тісного зв’язку між об’єктами, а також дозволили реалізувати обробку запитів як ланцюг послідовних відповідальних обробників.