Звіт

Лабораторна робота №2

“Проєктування та розробка програм

з використанням патернів проєктування”

Виконала студентка групи ІПС-21

Сенечко Дана Володимирівна

За основу для виконання цієї роботи я взяла код своєї першої лабораторної роботи (тому юніт-тести да документація вже наявні). Було внесено зміни в архітектуру проєкту без зміни функціональності сайту: відредаговано/доповнено код із використанням патернів проєктування.

Тепер в моєму коді можна виділити наступні класичні GoF патерни:

Creational Patterns

1. Factory Method

“Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.”

Реалізовано через створення моделей Mongoose. Наприклад:

```
const Product = mongoose.model("Product", productSchema);

const User = mongoose.model("User", userSchema);
```

`Mongoose.model()` функціонує як фабричний метод для створення моделей.

Переваги: єдиний інтерфейс для створення моделей, легке додавання нових типів моделей.

Недоліки: залежність від бібліотеки mongoose, тому можливі ускладнення при потребі кастомної ініціалізації.

2. Singleton

“Ensure a class only has one instance, and provide a global point of access to it.”

Реалізовано через підключення до бази даних. Наприклад:

```
export const connectDB = async () => {
    const conn = await mongoose.connect(process.env.MONGO_URI);
};
```

Також реалізовано в файлі productStore.js:

```
class SimpleProductStore {
    static instance;
    constructor() {
        if (SimpleProductStore.instance) {
            return SimpleProductStore.instance;
        }
        // ...
        SimpleProductStore.instance = this;
    }
}
```

Патерн забезпечує створення лише одного екземпляра `SimpleProductStore` в усьому додатку.

Переваги: економія ресурсів (єдине з’єднання з БД), централізоване управління станом продуктів.

Недоліки: можливе ускладнення тестування.

3. Builder

“Separate the construction of a complex object from its representation so that the same construction process can create different representations.”

Реалізовано у файлі responseBuilder.js:

- `ResponseBuilder` з fluent interface;
- Методи ланцюжка: `success(), message(), data(), meta(), error();`
- Фінальний метод `build()` для отримання результату.

Переваги: гнучкість в створенні різних типів відповідей та запобігання помилкам при їх створенні.

Structural Patterns

4. Adapter

“Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.”

Реалізовано у файлі DatabaseAdapter.js:

- Абстрактний клас `DatabaseAdapter` визначає єдиний інтерфейс;
- `MongooseAdapter` адаптує Mongoose до цього інтерфейсу;
- Всі CRUD операції інкапсульовані.

Переваги: легка заміна реалізації БД за потреби, єдиний інтерфейс для CRUD операцій.

5. Facade

“Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.”

Реалізовано через контролери. Контролери представляють собою фасад, який спрощує доступ до складної підсистеми (взаємодія з базою даних, бізнес-логіка, обробка помилок). Наприклад, файл product.controller.js:

```
export const getProducts = async (req, res) => {
    try {
        const products = await Product.find({});
        res.status(200).json({ success: true, data: products });
    } catch (error) {
        console.log("Error in fetching products:", error.message);
        res.status(500).json({ success: false, message: "Server Error" });
    }
};
```

Переваги: спрощення складної системи (БД, бізнес-логіка та обробка помилок), легке тестування та підтримка.

6. Decorator

“Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.”

Реалізовано у файлі decorators.js

- Базовий `MiddlewareDecorator`;

- Конкретні декоратори: `LoggingDecorator`, `RateLimitDecorator`, `ValidationDecorator`, `ErrorHandlingDecorator`;

- Фабричні функції для зручності використання.

Переваги: динамічне додавання функціональності до middleware, комбінування різних типів обробки запитів.

Недоліки: збільшення складності архітектури.

7. Proxy

“Provide a surrogate or placeholder for another object to control access to it.”

Реалізовано через ProtectedRoute.jsx:

```
if (!isAuthenticated) {
  return <Navigate to="/login" state={{ from: location }} replace />;
}

if (adminOnly && !isAdmin) {
  return <Navigate to="/unauthorized" replace />;
}
```

Також реалізовано через middleware для контролю доступу. Middleware protect та authorize діють як проксі, які контролюють доступ до захищених маршрутів, тобто middleware контролює доступ до оригінального об'єкта (обробника маршруту). Наприклад:

```
export const authorize = (...roles) => {

    return (req, res, next) => {

        if (!roles.includes(req.user.role)) {

            return res.status(401).json({

                success: false,
```

```
                message: `Role ${req.user.role} is not authorized to access
this route`
            });
        }
        next();
    };
};
```

Переваги: контроль доступу, централізована авторизація.

Behavioral Patterns

8. Chain of Responsibility

“Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.”

Кожен HTTP запит проходить через ланцюжок middleware функцій:

- `protect` - перевіряє JWT токен;

- `authorize("admin")` - перевіряє права доступу;

- `createProduct` - основний обробник.

Це класична реалізація даного патерну, де кожен обробник вирішує, чи обробляти запит або передати його далі по ланцюжку.

Використовується цей патерн в Express middleware системі. Наприклад, у файлі product.route.js:

```
router.post("/", protect, authorize("admin"), createProduct);

router.put("/:id", protect, authorize("admin"), updateProduct);

...
```

Переваги: гнучкість в порядку обробки, легке додавання або видалення ланок.

9. Observer

“Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”

Реалізовано в productStore.js + useProductStore.js:

```
subscribe(listener) {
    this.listeners.push(listener);
    return () => {
        this.listeners = this.listeners.filter(l => l !==
listener);
    };
}

notify() {
    this.listeners.forEach(listener => listener(this.products));
}
```

Переваги: автоматичне оновлення залежних компонентів, легке додавання нових спостерігачів.

10. Strategy

“Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.”

Реалізовано у файлі ValidationStrategy.js:

- Базовий клас `ValidationStrategy` з абстрактним методом `validate`;
- Конкретні стратегії: `ProductValidationStrategy`, `UserValidationStrategy`;
- Повертають уніфікований об'єкт із результатом валідації.

Переваги: різні правила валідації для різних типів даних, легке додавання або зміна алгоритмів валідації.

Недоліки: збільшення коду, цілком можливо в даному випадку недоцільне.

11. Command

“Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.”

Реалізовано у файлі productCommands.js:

- Базовий клас `Command` з методами `execute()` та `undo()`;
- Конкретні команди: `CreateProductCommand`, `UpdateProductCommand`, `DeleteProductCommand`, `BatchUpdateProductsCommand`;
- `ProductCommandInvoker` з історією команд та підтримкою undo/redo.

Переваги: підтримка undo/redo операцій, легке додавання нових команд.

Недоліки: досить складна реалізація для простих операцій.

Additional

Додатково було реалізовано Dependency Injection Container.

Висновки

Отже, в коді продемонстровано не лише базові патерни проєктування, а й дотримання принципів SOLID. Перевагами є гнучкість, модульність, масштабованість та підтримуваність програми. Серед недоліків можна виділити те, що, можливо, деякі з патернів є надмірними для відносно простого додатку. На мою думку, використання патернів підвищило якість архітектури та забезпечило дотримання принципів проєктування, що є основою для створення якісного програмного забезпечення.