УДК 62-97; 519.876.5
DOI https://doi.org/10.32782/2663-5941/2025.1.2/09

Григорчук Г.В.
Івано-Франківський національний технічний університет нафти і газу

Григорчук Л.І.
Івано-Франківський національний технічний університет нафти і газу

ЗАСТОСУВАННЯ ПАТЕРНІВ ПРОЕКТУВАННЯ ДЛЯ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ПРОГРАМНОГО КОДУ

У статті розглядається застосування патернів проектування для підвищення ефективності програмного коду. Патерни проектування, або шаблони проектування, є перевіреними рішеннями типових проблем, що виникають у процесі розробки програмного забезпечення. Вони сприяють підвищенню якості коду, зменшенню його складності та полегшенню його підтримки й розвитку. У роботі аналізуються основні типи патернів проектування, такі як породжуючі, структурні та поведінкові патерни, з акцентом на їх практичне застосування в різних аспектах програмування. Патерн проектування іменує, абстрагує і ідентифікує ключові аспекти структури, які дозволяють застосувати його для створення проектних рішень, що передбачають повторне використання.

Досліджено особливості застосування патернів проектування. Доведено, що їх використання допомагає структурувати код так, щоб пізніше в ньому можна було застосувати новий функціонал або здійснити зміни існуючого коду. Описано категорії патернів – породжуючі, структурні та поведінкові. Описано шляхи та особливості використання кожного з них. Детально розглянуто три основні види патернів, їх практичну реалізацію засобами об'єктно-орієнтованого програмування. Зокрема, показано, як впровадження цих патернів сприяє підвищенню гнучкості та масштабованості програмних систем. Дослідження включає приклади використання конкретних патернів у реальних проектах, демонструючи, як їхнє впровадження може знизити витрати на розробку, покращити продуктивність коду і сприяти більшій гнучкості програмних рішень. У висновках відзначено, що використання патернів проектування є важливим аспектом професійного програмування, що сприяє створенню надійних та масштабованих програмних продуктів.

Результати дослідження показують, що інтеграція патернів проектування в процес розробки значно підвищує ефективність програмного коду, роблячи його більш структурованим, читабельним та легким у підтримці.

Ключові слова: *патерни проектування, програмна архітектура, обєктно-орієнтоване програмування, шаблони проектування, якість коду.*

Постановка проблеми. Розробка програмного забезпечення є складним процесом, що вимагає високої якості коду, який повинен бути ефективним, легким у підтримці та масштабованим. Зі зростанням складності програмних систем виникає потреба у використанні перевірених рішень для типових проблем, що виникають під час розробки. Патерни проектування є такими рішеннями, однак їх правильне застосування потребує глибокого розуміння та досвіду. Постає питання, як ефективно інтегрувати патерни проектування у процес розробки, щоб досягти оптимальної продуктивності та якості коду.

Аналіз останніх досліджень і публікацій. Концепцію патернів вперше описав Крістофер Александер у книзі «Мова шаблонів. Міста. Будівлі. Будівництво». У книзі описано «мову» для проектування навколишнього середовища, одиниці якого – шаблони (або *патерни*, що ближче до оригінального терміна *patterns*) – відповідають на архітектурні запитання: якої висоти потрібно зробити вікна, скільки поверхів має бути в будівлі, яку площу в мікрорайоні відвести для дерев та газонів. Ідея видалася привабливою четвірці авторів: Еріху Гаммі, Річарду Хелму, Ральфу Джонсону, Джону Вліссідесу. У 1994 році вони написали книгу «Патерни проектування: повторно використовувані елементи архітектури об'єктно-орієнтованого програмного забезпечення», до якої увійшли 23 патерни, що вирішують різні проблеми об'єктно-орієнтованого дизайну». З того часу було знайдено десятки інших об'єктних патернів. «Патерновий» підхід став популярним і в інших галузях програмування, тому зараз

можна зустріти різноманітні патерни також за межами об'єктного проектування [1].

Постановка завдання. Метою дослідження є вивчення впливу патернів проектування на ефективність програмного коду та їх практичне застосування у розробці програмних систем засобами об'єктно-орієнтованого програмування. Дослідження спрямоване на аналіз переваг використання різних патернів, визначення кращих практик їх впровадження, а також оцінку їхнього впливу на продуктивність, читабельність та підтримку коду. Шаблони програмування, це свого роду, методики, які можуть допомогти розробнику розв'язати певні проблеми у певних ситуаціях. Іншими словами, вони можуть значно полегшити щоденну рутину то зробити проектування більш оптимальним процесом.

Виклад основного матеріалу. Патерн проектування представляє собою типовий метод розв'язання конкретної проблеми, яка часто виникає під час розробки програмної архітектури. Він не є частиною мови програмування, як бібліотеки чи функції, а тому не може бути прямо використаний в коді програми. Він скоріше є набором порад щодо того, як структурувати код таким чином, щоб потім у нього було легко додавати новий функціонал чи змінювати існуючий.

Патерни проектування розділяють на три категорії: породжуючі, структурні та поведінкові.

Породжуючі патерни описують гнучкий механізм створення об'єктів. До цієї категорії відносяться такі патерни як: Фабричний метод, Абстрактна фабрика, Будівельник, Прототип та Одинак. Структурні патирни вказують на те, як варто будувати зв'язки між об'єктами. До них відносяться: Адаптер, Міст, Компонувальник, Декоратор, Фасад, Легковаговик, Замісник.

Поведінкові патерни показують, якою повинна бути комунікація між об'єктами. До цієї категорії відносяться такі патерни як: Ланцюжок обов'язків, Команда, Ітератор, Посередник, Знімок, Спостерігач, Стан, Стратегія, Шаблонний метод та Відвідувач.

Щоб показати ефективність програмного коду, роблячи його більш структурованим, читабельним та легким у підтримці розглянемо наступні патерни [2].

Будівельник – це породжуючий патерн що полегшує створення складних об'єктів. Під складними об'єктами розуміють об'єкти, які мають багато полів.

Уявімо, що ми проектуємо робота. Робот, що схожий на людину, може мати руки, ноги, голову і тіло. Але в деяких випадках нам може бути не потрібна голова або, якщо потрібно, щоб робот літав, то ми можем додати крила і забрати ноги. Щоб створити такий об'єкт робота, можна використати конструктор, який приймає всі можливі параметри робота, але тоді такий конструктор буде дуже довгим і не завжди його доцільно використовувати, адже не завжди нам потрібно вказувати всі параметри.

```
new Robot(head, body, hands, legs, wings)
new Robot(null, body, hands, legs, null) //
робот без голови і крил
new Robot(null, body, hands, null, null) //
робот, який має тільки тіло
```

Іншим варіантом може бути використання конструктору без параметрів і встановлення потрібних полів пізніше [3]. Але такий підхід теж має свої недоліки, тому що дозволяє створити об'єкт, в якого всі поля null, що немає жодного сенсу.

```
Robot robot = new Robot()
robot.setHead(head)
robot.setBody(body)
```

Патерн Будівельник доручає створення об'єкту робота іншим об'єктам, які називаються будівельниками.

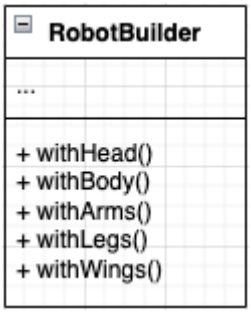


Рис. 1. Застосування патерну Будівельник

Приклад на мові Java

```
public class Robot {
  private String head;
  private String body;
  private String[] hands;
  private String[] legs;
  private String[] wings;
  private Robot(String head, String body,
String[] hands, String[] legs, String[] wings)
{
    this.head = head;
```

```
    this.body = body;
    this.hands = hands;
    this.legs = legs;
    this.wings = wings;}
  public String getHead() {
    return head;}
  public String getBody() {
    return body;}
  public String[] getHands() {
    return hands;}
  public String[] getLegs() {
    return legs;}
  public String[] getWings() {
    return wings;}
  public static RobotBuilder builder(String
body) {
    return new RobotBuilder(body);}
  public static class RobotBuilder {
    private String head;
    private String body;
    private String[] hands;
    private String[] legs;
    private String[] wings;
    public RobotBuilder(String body) {
      this.body = body;}
    public RobotBuilder withHead(String head)
{
      this.head = head;
      return this;}
     public RobotBuilder withHands(String[]
hands) {
      this.hands = hands;
      return this;}
    public RobotBuilder withLegs(String[] legs)
{
      this.legs = legs;
      return this;}
     public RobotBuilder withWings(String[]
wings) {
      this.wings = wings;
      return this;}
    public Robot build() {
      return new Robot(head, body, hands,
legs, wings);}}}
```

Отже, у нас є клас Robot з приватним конструктором, що унеможливлює створення об'єкту через конструктор. Також клас містить метод builder, який повертає об'єкт будівельника. Метод builder має вхідний параметр body, що робить його необхідним параметром, і гарантує нам те, що робот завжди буде мати тіло. Своєю чергою, будівельник RobotBuilder має набір методів, що дозволять вказати частини робота та метод build що повертає об'єкт класу Robot.

Приклад застосування

```
Robot robot = Robot.builder("green body").
build();
Robot robot = Robot.builder("green body").
withHead("big head").build();
```

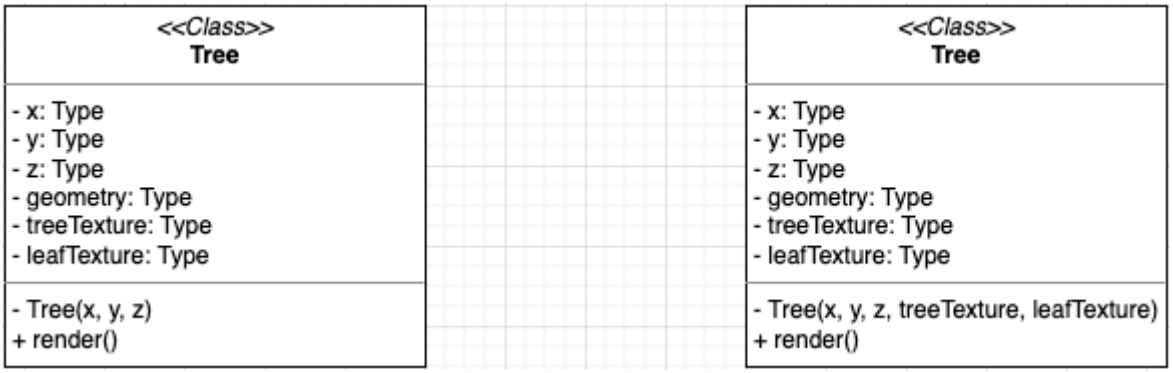
У першому випадку ми створюємо робота що має тільки тіло, у другому – тіло і голову. Використання патерну Будівельник дозволяє нам позбутись використання громістких конструкторів з великою кількістю параметрів з яких, у більшості випадків, нам потрібно тільки декілька. Також надає зручний спосіб конструювання складних об'єктів який може містити в собі додаткову логіку що забезпечує нам створення коректних екземплярів об'єктів [3; 4].

Легковаговик – це структурний патерн, що дозволяє зменшити обсяг потрібної пам'яті для зберігання об'єктів за рахунок винесення спільних ресурсів за межі об'єктів.

До прикладу, розробнику комп'ютерних ігор потрібно зробити рівень у грі, на якому буде ліс з великою кількістю дерев. Кожне дерево – це об'єкт, що містить геометрію дерева, його координати у віртуальному світі та матеріали (текстури дерева та листя). Для простоти розрахунків приймемо, що текстура дерева і листя займають по 2 мегабайти кожна і геометрія дерева – 1 мегабайт. В цілому, один об'єкт дерева буде займати близько 5 мегабайтів. Отже, якщо ми додаємо в гру хоча б 20 таких дерев, то це вже буде 100 мегабайт, а для нормального лісу будуть потрібні сотні мегабайт.

У такому випадку розробник може обмежитись декількома деревами або використати текстури, що мають гіршу якість і займають менше пам'яті, але все це вплине на якість нашої гри, що для розробника неприпустимо. В такому випадку доцільно застосувати патерн Легковаговик до створення дерева (Рис. 2).

Зліва (Рис. 2) показано початковий варіант класу, де текстури дерева і листя визначалися всередині класу і, як результат, кожен об'єкт дерева мав у собі об'єкти текстур. У варіанті справа (Рис. 2) текстури є параметрами конструктору і більше не створюються всередині об'єкта дерево, що дозволяє на використовувати ті ж текстури дерева і листя для всіх дерев. Це дозволить суттєво зекономити пам'ять, потрібну для зберігання створених об'єктів, і тепер для 20 дерев розробнику потрібно лише 24 мегабайти.



а) б)

Рис. 2. структура програмного коду до (а) та після (б) застосування патерну

Шаблонний метод – це поведінковий патерн, що визначає загальний алгоритм виконання, перекладаючи відповідальність за реалізацію його кроків на підкласи.

Як приклад використаємо випадок, коли потрібно написати програму для апарату, що варить каву. Алгоритм варіння кави досить простий: потрібно підігріти воду, додати каву, додати цукор, додати вершки або молоко і вилити отриманий результат у ємність. Але, наприклад, для експресо потрібно менше води, для капучино – додати молоко і т.д. Можна створити для кожного напою окремий клас, який буде описувати, як готувати той чи інший напій. Але це призведе до дублювання коду. Тому в цьому випадку добре буде скористатися патерном шаблонний метод.

```
public abstract class CoffeeMaker {
  //шаблонний метод що визнащає загальний
алгоритм варіння
  public void make() {
    addWater();
    heatWater();
    addCoffee();
    addSugar();
    addMilk();
    pourInCup();}
  protected abstract void addWater();
  protected void heatWater() {
    System.out.println("Heat water to 90
degree");}
  protected void addCoffee() {
    System.out.println("Add 20 grams of
coffee");}
  protected abstract void addSugar();
  protected abstract void addMilk();
  protected void pourInCup() {
    System.out.println("Pouring into cup");}}
```

CoffeMaker – це клас, який визначає загальний алгоритм варіння кави, розбиваючи його на окремі кроки, які можуть бути змінені підкласами. Далі розглянемо конкретні класи, відповідальні за варіння різних напоїв.

```
public class AmericanoCoffeeMaker extends
CoffeeMaker {
  @Override
  protected void addWater() {
    System.out.println("Add 200 grams of
water");}
  @Override
  protected void addSugar() {
    System.out.println("Add 20 grams of
sugar");}
  @Override
  protected void addMilk() {
    System.out.println("Add 30 grams of
milk");}}
```

AmericanoCoffeeMaker – відповідальний за варіння американо. Він наслідує CoffeMaker і реалізує три методи, щоб додати певну кількість води(addWater), цукру(addSugar) та молока(addMilk). Зверніть увагу, що в класі немає методу make(), оскільки він наслідується з батьківського класу.

```
public class EspressoCoffeeMaker extends
Cpublic class EspressoCoffeeMaker extends
CoffeeMaker {
  @Override
  protected void addWater() {
    System.out.println("Add 100 grams of
water");}
  @Override
  protected void addSugar() {
```

```
    System.out.println("Add 10 grams of
sugar");}
  @Override
  protected void addMilk() {
    //реалізація відсутня оскільки молока не
потрібно}}
```

EspressoCoffeeMaker – відповідальний за варіння експресо. Він також наслідує CoffeMaker, щоб використовувати заздалегідь визначений алгоритм варіння, і реалізує два методи addWater і addSugar.

Observer (**спостерігач)** – це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об'єктам стежити й реагувати на події, які відбуваються в інших об'єктах.

Розглянемо патерни в контексті сучасних парадигм. Асинхронне програмування стало важливою частиною сучасних систем, особливо в контексті веб-розробки та роботи з великими обсягами даних у реальному часі. Багато патернів проектування чудово адаптуються до цього підходу. Наприклад, у **JavaScript** та **Python** активно використовується асинхронність для побудови неблокуючого коду, що працює швидко і ефективно.

Observer – це один із патернів, який часто використовується для реалізації асинхронних систем. Його основна мета – реалізація реактивного підходу, коли один або кілька об'єктів «спостерігають» за станом іншого об'єкта та отримують повідомлення про зміни стану в режимі реального часу [4; 7; 8].

Асинхронність з використанням **Observer** дозволяє реагувати на події у системі негайно, без блокування головного потоку програми, що робить його ідеальним для реалізації таких систем, як інтерфейси реального часу, чати або обробка даних з датчиків.

Приклад використання патерну Observer в асинхронних системах. Ось приклад, як патерн **Observer** може бути реалізований в асинхронній системі на JavaScript з використанням **Promises** або **async/await** для роботи з подіями в режимі реального часу.

Приклад на мові javascript:

```
class Observable {
    constructor() {
        this.observers = [];}
    subscribe(observer) {
        this.observers.push(observer);}
    unsubscribe(observer) {
        this.observers = this.observers.
filter(subscriber => subscriber !==
observer);}
    notify(data) {
        this.observers.forEach(observer =>
observer.update(data)); }}
class Observer {
    constructor(name) {
        this.name = name;}
    async update(data) {
        console.log(`${this.name} отримав
повідомлення:`, data);
        // Імітація асинхронної операції,
наприклад, виклик API
        await new Promise(resolve =>
setTimeout(resolve, 1000));
        console.log(`${this.name} обробив
дані:`, data);}}
// Приклад використання
const observable = new Observable();
const observer1 = new Observer('Спостерігач 1');
const observer2 = new Observer('Спостерігач 2');
observable.subscribe(observer1);
observable.subscribe(observer2);
// Імітація події через 2 секунди
setTimeout(() => {
    observable.notify('Нові дані!');}, 2000);
```

Опис коду:

- Observable – клас, що представляє об'єкт, за яким можуть спостерігати інші об'єкти (спостерігачі).
- **Observer** – спостерігач, який підписується на події та реагує на зміни, обробляючи їх асинхронно.
- У методі update використовуються асинхронні операції для симуляції обробки події.

У прикладі, коли спостерігач отримує повідомлення від спостережуваного об'єкта, він асинхронно обробляє ці дані. Це імітує реальні умови, коли система повинна реагувати на зміни даних у реальному часі без блокування основного потоку.

Висновки. Результати дослідження показують, що інтеграція та удосконалення вище вказаних патернів в процес розробки значно підвищує ефективність програмного коду, роблячи його більш структурованим, читабельним та легким у підтримці [5].

Використання патернів проектування покращує організацію нашого коду що полегшує його підтримку і додавання нового функціоналу,

а у деяких випадках може покращити швидкодію програми чи ефективність роботи з ресурсами.

Observer чудово підходить для реалізації асинхронного реагування на події. Це зручний патерн для роботи з системами, де важливо обробляти події у фоновому режимі або в паралельних потоках, не зупиняючи головний потік виконання програми.

Працюючи з будь-яким ресурсом до створення безшовного зображення головний секрет успіху – однорідність отриманої картинки. Для цього згенерованим узором слід залити велику поверхню, якщо око не реагує на стики, сприймає орнамент єдиною цілісною площиною, отже, все пройшло добре. Якщо ж видно дірки, нерівномірне заповнення або деякі деталі випадають із загальної концепції, потрібно доопрацювати патерн.

Список літератури:

1. Будай А. Дизайн-патерни – простіше простого. Львів, 2012. 90 с.

2. Бандура В.В., Чесановський М.С., Шекета В.І., Піх В.Я. Архітектура та проєктування програмного забезпечення: навч. посібник. Івано-Франківськ: ІФНТУНГ, 2021. 359 с.

3. Cooper J.W. Introduction to Design Patterns in C#. IBM TJ Watson Research Center, 2002. 424 p.

4. Serevsen B.G. Design Patterns. BGS Consulting ApS, 2008. 224 p.

5. Юхнович П. Навіщо потрібні патерни проєктування в програмуванні та де їх вивчати. URL: https://aw.club/global/uk/blog/design-patterns-in-programming.

6. Баран С.В. Розробка програмного забезпечення з використанням патернів проєктування: навч. посібник. Кривий Ріг: Державний університет економіки і технологій, 2023. 203 с.

7. Венгловська Ю.В., Марчук Д.К. Використання патернів у ігрових проєктах. URL: https://conf.ztu.edu.ua/wp-content/uploads/2023/06/67-1.c.127-129.

8. Сікайло В.О., Марчук Г.В. Дослідження процесів використання патернів проєктування при розробці ігор. URL: https://conf.ztu.edu.ua/wp-content/uploads/2020/05/37-1.c.37-39.

Grygorchuk G.V., Grygorchuk L.I. APPLICATION OF DESIGN PATTERNS TO INCREASE THE EFFICIENCY OF SOFTWARE CODE

The article examines the application of design patterns to enhance the efficiency of software code. Design patterns, or design templates, are proven solutions to common problems that arise during the software development process. They help improve code quality, reduce its complexity, and facilitate its maintenance and evolution. The paper analyzes the main types of design patterns, such as creational, structural, and behavioral patterns, with an emphasis on their practical application in various aspects of programming. The design pattern names, abstracts, and identifies key aspects of the structure, allowing it to be applied to create design solutions that enable reuse

The study investigates the features of applying design patterns. It is proven that their use helps structure the code in such a way that new functionality can be implemented or existing code can be modified more easily. The categories of patterns–creational, structural, and behavioral–are described. The ways and peculiarities of using each of them are outlined. The three main types of patterns and their practical implementation using object-oriented programming (OOP) are discussed in detail. Specifically, it shows how the implementation of these patterns contributes to increased flexibility and scalability of software systems. The study includes examples of specific patterns used in real projects, demonstrating how their implementation can reduce development costs, improve code performance, and promote greater flexibility in software solutions. The results of the study show that integrating design patterns into the development process significantly enhances the efficiency of software code, making it more structured, readable, and easier to maintain. The conclusions highlight that the use of design patterns is an important aspect of professional programming, contributing to the creation of reliable and scalable software products.

Key words: *design patterns, software architecture, object-oriented programming, design templates, code quality.*