УДК 004.415:004.42

УСПАДКУВАННЯ ПРОТИ КОМПОЗИЦІЇ: ПРАВИЛЬНА ПОБУДОВА ІЄРАРХІЇ КЛАСІВ ТА ПРИЧИНИ, ЧОМУ НЕ ВАРТО ЗЛОВЖИВАТИ УСПАДКУВАННЯМ

Діденко Т.В.[1]; Євдокимов М.О.[2]
[1,2]Національний університет «Одеська політехніка»
[1,2]Україна, Одеса
[1]https://orcid.org/0009-0006-7880-2638; [2]m.o.yevdokymov@op.edu.ua

Анотація. *У роботі досліджено вразливості механізму успадкування в об'єктно-орієнтованому програмуванні, зокрема проблему «крихкого базового класу» (fragile base class problem). Обґрунтовано, що надмірне використання успадкування призводить до високої міжкласової зв'язаності та порушення принципу інкапсуляції. На прикладі програмного коду мовою Java наведено докази переваги композиції над успадкуванням для забезпечення гнучкості, розширюваності та спрощення тестування програмних систем.*

Ключові слова: *об'єктно-орієнтоване програмування; Java; композиція; успадкування; крихкий базовий клас; інкапсуляція; зв'язаність.*

Вступна частина. У сучасному інжинірингу програмного забезпечення одна з найбільш популярних парадигм – об'єктно-орієнтоване програмування (ООП). Попри численні переваги, підхід має концептуальні ризики, пов'язані з неправильним проектуванням ієрархій класів. Зокрема, використання успадкування як основного інструменту повторного використання коду нерідко викликає проблему «крихкого базового класу» (fragile base class problem) [1]: модифікація батьківського класу може ненавмисно порушити коректність похідних класів. Тому постає потреба в альтернативних підходах до проектування, насамперед у використанні композиції для мінімізації міжкласової зв'язаності [2]. Принцип «Favor object composition over class inheritance» («Надавайте перевагу композиції об'єктів замість успадкування класів») сформульовано в праці Е. Гамми, Р. Хелма, Р. Джонсона та Дж. Вліссідеса (Gang of Four) [3] і поглиблено Дж. Блохом [4], який проаналізував ризики розширення класів, не призначених для успадкування.

Метою дослідження є проведення порівняльного аналізу механізмів успадкування та композиції в об'єктно-орієнтованому програмуванні (на прикладі мови Java), виявлення практичних ризиків надмірного використання успадкування та обґрунтування переваг композиції для побудови гнучких і простих у тестуванні програмних систем.

Для досягнення поставленої мети визначено такі завдання:

1. Дослідити семантичні та структурні відмінності між відношеннями успадкування («Is-a») та композиції («Has-a»).

2. Проаналізувати проблему «крихкого базового класу» на практичному прикладі програмного коду Java.

3. Порівняти успадкування та композицію за критеріями зв'язаності, інкапсуляції та придатності до тестування.

Основна частина. Успадкування реалізує зв'язок типу «Is-a» [5, 6]. При цьому дочірній клас успадковує всі поля та методи батьківського класу, стаючи тісно пов'язаним із його внутрішньою реалізацією. Композиція реалізує зв'язок типу «Has-a» [6, 7], за якого клас-

контейнер містить екземпляр іншого класу як свій приватний атрибут та делегує йому виконання необхідних операцій.

Для демонстрації вразливостей успадкування розглянемо початкову реалізацію базового класу Car та похідного класу ElectricCar мовою Java:

```
class Car {
    public void move() { System.out.println("Car move"); }
    public void start() { System.out.println("Car start"); }
    public void stop() { System.out.println("Car stop"); }
}
class ElectricCar extends Car {
    public void charge() { System.out.println("Electric car charging"); }
}
```

При виклику методів для екземпляра ElectricCar програма функціонує передбачувано. Однак, якщо розробник базового класу Car додасть новий метод refuelWithGas() та модифікує метод stop() для автоматичної заправки бензином:

```
public void refuelWithGas() { System.out.println("Car is refueling with gas"); }
public void stop() {
    System.out.println("Car stop");
    refuelWithGas();
}
```

У результаті такої зміни виконання методу stop() для ElectricCar виведе некоректне повідомлення про заправку електричного автомобіля бензином. Це наочно демонструє проблему крихкого базового класу [1]: похідний клас виявився вразливим до змін у внутрішній логіці батьківського класу.

Для усунення жорсткої залежності рекомендовано використовувати відношення композиції [4, 8]. Перероблена структура класів має такий вигляд:

```
class Car {
    public void move() { System.out.println("Car move"); }
    public void start() { System.out.println("Car start"); }
    public void stop() { System.out.println("Car stop"); }
}
class ElectricCar {
    private Car car = new Car();
    public void charge() { System.out.println("Electric car charging"); }
    public void drive() {
        car.start();
        car.move();
        car.stop();
        charge();
    }
}
class GasCar {
    private Car car = new Car();
    public void refuelWithGas() { System.out.println("Gas car refueling"); }
    public void drive() {
        car.start();
        car.move();
        car.stop();
        refuelWithGas();
    }
}
```

У даній реалізації забезпечується слабка зв'язаність (loose coupling) [4]: будь-які зміни або додавання нових методів у класі Car не впливають на логіку ElectricCar чи GasCar, оскільки взаємодія здійснюється виключно через явний виклик необхідних методів. Результати порівняння наведено в табл. 1.

Таблиця 1

Порівняння характеристик успадкування та композиції

Критерій	Успадкування	Композиція
Зв’язаність	Висока (зміна базового класу змінює поведінку нащадків)	Низька (компоненти слабко залежать один від одного)
Інкапсуляція	Порушується (дочірній клас має доступ до внутрішньої реалізації)	Зберігається (взаємодія виключно через публічні інтерфейси)
Тестування	Складне (потребує ініціалізації всієї ієрархії класів)	Просте (компоненти легко замінити мок-об'єктами/заглушками)

Висновки. Проведений порівняльний аналіз підтвердив, що неефективне або зловмисне застосування механізму успадкування спричиняє високий рівень зв'язаності класів та провокує проблему «крихкого базового класу». На практичному прикладі мовою Java доведено, що надання переваги композиції дозволяє зберегти інкапсуляцію, підвищити гнучкість архітектури та істотно спростити модульне тестування. Отже, під час проектування програмного забезпечення слід керуватися принципом переваги композиції над успадкуванням, обмеживши використання успадкування лише випадками строгої та незмінної поліморфної ієрархії.

Використання інструментів ШІ. Під час підготовки тез доповідей інструменти штучного інтелекту використовувалися для граматичного редагування тексту, коригування стилістики українськомовного та англомовного анотаційних блоків та виправлення синтаксичних конструкцій у прикладах коду Java.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

[1] Lavysh, Z. (2021). *Fragile base class problem for Dart developers*. Medium. https://medium.com/@lavysh.zahar/fragile-base-class-problem-for-dart-developers-028c855ff89c

[2] Arney, D. (2020). *Loose coupling explained in simple terms*. Medium. https://medium.com/@dinesharney/loose-coupling-explained-in-simple-terms-f4ec17e7ca9e

[3] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

[4] Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley Professional.

[5] Codecademy. (2023). *What is inheritance in object-oriented programming?* https://www.codecademy.com/resources/blog/what-is-inheritance

[6] Vinotech. (2022). *The real difference between is-a and has-a*. Medium. https://medium.com/@vino7tech/the-real-difference-between-is-a-and-has-a-oop-interview-prep-43ba9814b321

[7] Janssen, T. (2022). *OOP concepts for beginners: What is composition?* Stackify. https://stackify.com/oop-concepts-composition/

[8] Kumar, P., & Kurup, M. (2023). *Gang of 4 design patterns explained: Creational, structural, and behavioral*. DigitalOcean. https://www.digitalocean.com/community/tutorials/gangs-of-four-gof-design-patterns

Didenko T.V., Yevdokymov M.O.

Inheritance vs composition: proper class hierarchy building and reasons not to overuse inheritance

Abstract. *The paper investigates vulnerabilities of the inheritance mechanism in object-oriented programming, specifically the fragile base class problem. It is substantiated that overuse of inheritance leads to high coupling and violation of encapsulation principles. Using Java code examples, the advantages of object composition over class inheritance are proven to ensure architectural flexibility, extensibility, and simplified testing of software systems.*

Keywords: *object-oriented programming; Java; composition; inheritance; fragile base class; encapsulation; coupling.*