**УДК 004.432.2**
**DOI: 10.18523/2617-3808.2025.8.138-148**

*Бублик В. В., Фітель Д. Р.*

# ОБ'ЄКТНО-ОРІЄНТОВАНА ПАРАДИГМА: PRO I CONTRA

*У статті наведено критику парадигми об'єктно-орієнтованого програмування (ООП) та її найбільш поширених реалізацій. Досліджено історію виникнення та подальшої еволюції ООП, її сильні й слабкі сторони, а також спільні й відмінні риси між ООП та іншими парадигмами. Проаналізовано приклади вдалого співіснування парадигм ООП і узагальненого програмування на прикладі шаблонів у мові C++.*

**Ключові слова:** парадигма програмування, об'єктно-орієнтована парадигма, узагальнене програмування.

## Вступ

Стаття є розширеною версією доповіді авторів [2] на Міжнародній конференції TAAPSD 24, присвяченої особливостям застосування об'єктно-орієнтованої парадигми. Щоб уникнути суперечок про виникнення концепції об'єктно-орієнтованого програмування, одразу зазначимо, що в епоху становлення алгоритмічних мов програмування, скажімо, Фортрану, створеного в 1954–1957 рр. [9], В. С. Королюк і К. Л. Ющенко створили перший варіант мови адресного програмування, фіналізований у 1963 р. [3]. Адресне програмування виявилося предтечою важливих концепцій структурованих даних — структур і указників. На жаль, через відомі причини адресне програмування залишилося непоміченим на Заході, хоча його структури і указники досягли практично нинішньої форми, відомої з мови програмування C [3], створеної Деннісом Рітчі в 1972–1973 рр. під значним впливом із боку Алголу і Фортрану. За оцінкою Дональда Кнута, «the way C handles pointers, for example, was a brilliant innovation; it solved a lot of problems that we had before in data structuring and made the programs look good afterwards» [11].

Щоб дістатися об'єктно-орієнтованої парадигми, залишилося лише додати до структур і указників класи і об'єкти, успадкування і віртуальність. Усе це, одначе, було ще в 1967 р. у першій мові об'єктно-орієнтованого програмування, яка одержала назву SIMULA 67. З історією створення мови можна ознайомитися за статтею її авторів Крістена Нюгора і Уле-Югана Дала [13]. За свідченням авторів, головного впливу мова SIMULA 67 зазнала з боку алгоритмічної мови ALGOL 60.

Нарешті, за свідченням Б'ярне Страуструпа, коло Фортран-Алгол-C-SIMULA зімкнулося поєднанням у 1983 р. мов C і SIMULA 67 в новій мові програмування C++ [14]. За понад 40 років свого існування і інтенсивних застосувань мова C++ розвивалася в напрямі зростання рівня вбудованих до неї абстракцій, набуваючи багатьох нових рис. При цьому C++ залишалася мультипарадигменною, не наполягаючи на штучному впровадженні об'єктів усюди і скрізь, порівняно з деякими іншими мовами програмування.

C++:

```
int main()
{
    std::cout << «Hello, World!“ << std::endl;
}
```

```
Java:
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println(«Hello, World!»);
  }
}
```

У контексті мови C++ несподіваним видається висловлювання, приписане Бобом Кроуфордом (у важкодоступній роботі [6]) світовому програмісту номер один Едсгеру Дейкстрі: «Object-oriented programming is an extremely bad idea which could only have originated in California». Це звучить тим більш дивно, тому що одна з найцитованіших праць про структурне програмування [7], написана Дейкстрою у співавторстві з Уле-Юганом Далом і Тоні Хоаром, містить об'єктно-орієнтований розділ. Справді, впровадження об'єктів, успадкування та поліморфізму може зробити програми складнішими для розуміння та підтримки. А критичне ставлення Дейкстри можна зрозуміти, зважаючи на його позицію про те, що (добре розроблена) програма має близьку структурну аналогію з будь-якою (добре розробленою) математичною теорією [8]. Тому, на думку Дейкстри, робота програміста суттєво не відрізняється від роботи творчого математика.

В останньому твердженні прихована головна причина розбіжності підходу Дейкстри з сучасними тенденціями в інженерії програмного забезпечення: превалювання інженерного підходу над математичним. Зрештою, об'єктно-орієнтована парадигма — це не більше, ніж інструмент, призначений для розв'язування широкого кола задач. Але цей інструмент проявив себе у багатьох всесвітньо відомих проєктах. Особливо у випадках, коли проєкти мають великий обсяг, а структури даних виявляються надто складними. Інженерний підхід до програмування певним чином використовує рух знизу догори. Спочатку вивчається проблема, визначаються і виділяються її найважливіші частини. Вони відокремлюються від усього іншого, несуттєвого з погляду розробника, і утворюють абстрактне подання проблеми. Саме у виділенні і побудові релевантних абстракцій прихована потужність об'єктно-орієнтованої парадигми.

### Узагальнені абстракції: варіативні шаблони

Нового поштовху парадигмі надала робота Андрея Александреску [4], у якій поєднано об'єктно-орієнтовану парадигму з парадигмою узагальненого програмування. На час виходу книжки узагальнене програмування вже з 1970-х років застосовували в деяких мовах програмування, а на межі 1980–1990-х воно потрапило до C++ у вигляді шаблонів функцій і класів. Починаючи з C++11 мова перетворилася на найпотужніший інструмент метапрограмування. Виявилося, що разом об'єктна орієнтованість і узагальненість досягають результатів, недосяжних ними поодинці.

Але все одно тотальна «об'єктизація» не завжди виправдана, як і механізми успадкувань, які вважають головним досягненням об'єктно-орієнтованого підходу. На думку Петера Готшлінга, «the most important benefit of classes in C++ for us is not the inheritance mechanisms but the ability to establish new abstractions and to provide alternative realizations for them» [10]. Тезу підтверджує наведений у книжці приклад шаблону для задачі пошуку екстремуму методом градієнтного спуску, максимально наближений до математичного алгоритму.

```
template <typename Value, typename T1, typename T2,
    typename FF, typename GG>
auto gradient_descent(Value& u, T1 s, T2 eps, FF f, GG g)
{
    auto val = f(u), delta = val;
    do {
          u -= s * g(u);
          auto new_val = f(u);
          delta = abs(new_val - val);
          val = new_val;
    } while (delta > eps);
    return u;
}
```

Цей код, написаний для функцій двох змінних, допускає прості варіанти узагальнень до *n*-місних функцій. Визначимо структуру R, відповідну *n*-вимірному простору,

```
struct R
{
    static const int n=10;
    double x[n]{};
};
```

на якому задаємо оператори множення і віднімання, задіяні в алгоритмі,

```
const R inline operator*(double s, const R& u)
{
    R res;
    for (int i = 0; i < R::n; ++i)
          res.x[i] = s * u.x[i];
    return res;
}

R& operator-=(R& u, const R& v)
{
    for (int i = 0; i < R::n; ++i)
          u.x[i] -= v.x[i];
    return u;
}
```

Цільову параболічну функцію і її градієнти задаємо лямбда виразами прямо в операторі виклику

```
gradient_descent(u, h, eps,
        [](const R& u)
        {
            double res = 0;
            for (int i = 0; i <R:: n; ++i)
                res += u.x[i] * u.x[i];
            return res;
        },
        [](const R& u)->const R
        {
            R res;
            for (int i = 0; i < R::n; ++i)
                res.x[i] = 2 * u.x[i];
            return res;
        });
```

Цікаво, що застосування варіативних шаблонів дозволить уникнути явної залежності від розмірності простору. Спочатку задано варіативну структуру простору довільної розмірності. Тут застосована особлива техніка часткової спеціалізації шаблону структури R, не визначеного, а лише попередньо декларованого

```
template <typename ...T> struct R;

template <typename T, typename ...P>
struct R<T, P...>
{
    T x;
    R<P...> y;
};

template<>
struct R<> {};
```

Завдяки такому визначенню в одному і тому самому фрагменті коду можуть співіснувати різні розмірності, наприклад,

```
R<double, double, double> u = { 1, 2, 3 };
R<double, double, double, double> v = { 7, 8, 9, 10 };
```

Наведемо варіативні рекурсивні визначення для операторів множення і віднімання, застосовані у функції gradient_descent. Зверніть увагу, параметр розмірності простору n став повністю зайвим.

```
template <typename ...T>
inline const R<T...> operator*(double s, const R<T...>& u);

template <typename T, typename ...P>
inline const R<T, P...> operator*(double s, const R<T, P...>& u)
{
    return { s * u.x, s * u.y};
}

template <>
inline const R<> operator*(double s, const R<>& u)
{
    return u;
}

template <typename ...T>
R<T...>& operator-=(R<T...>& u, const R<T...>& v);

template <typename T, typename ...P>
inline R<T, P...>& operator-=(R<T, P...>& u, const R<T, P...>& v)
{
    u.x -= v.x;
    u.y -= v.y;
    return u;
}

template <>
inline R<>& operator-=(R<>& u, const R<>& v)
{
    return u;
}
```

І, нарешті, цільова функція і її градієнти

```
template <typename ...T>
inline double f(const R<T...>& u);

template <typename T, typename ...P>
inline double f(const R<T, P...>& u)
{
    return u.x*u.x+f(u.y);
}

template <>
inline double f(const R<>& u)
{
    return 0;
}

template <typename ...T>
inline R<T...> g(const R<T...>& u);

template <typename T, typename ...P>
inline R<T, P...> g(const R<T, P...>& u)
{
    return { 2 * u.x, g (u.y)};
}

template <>
inline R<> g(const R<>& u)
{
    return {};
}
```

Виклик, як і раніше, може містити лямбда вирази у вигляді обгорток для варіативних функцій.

```
gradient_descent(u, h, eps,
        [](auto u) { return f(u); },
        [](auto u) { return g(u); });
```

Обидва приклади повністю відповідають процедурному стилю програмування, наявному в C++. Хоча застосування структур і операторів над ними (по суті об'єктів) виводить нас за рамки чистого C, не кажучи вже про варіативність шаблонів. Зауважимо, що, на думку загальновизнаного експерта в об'єктно-орієнтованому програмуванні на C++ Скотта Меєрса, перетворення функцій над класами на їхні методи безпідставно порушує їхню безпечність [12]. Ясно, що деякі функції класів мусять розміщуватися всередині класу. Це конструктори, деструктори, присвоєння і деякі інші. Вичерпний алгоритм розміщення функцій всередині і поза класами наводять Герб Саттер і Андрей Александреску [15].

### Статичний і динамічний поліморфізм

Однією з причин розміщення функцій усередині класів може бути поліморфізм. Поліморфізм вимагає віртуальних методів, і це важлива суперечлива тема в об'єктно-орієнтовній частині C++. Незважаючи на тенденцію віртуалізації функцій у сучасних мовах програмування, C++ успішно поєднує використання невіртуальних і віртуальних функцій.

Традиційний спосіб організації ієрархій полягає у створенні абстрактного класу або інтерфейсу з чисто віртуальними функціями, реалізованими в похідних класах. Простим прикладом може бути інтерфейс функторів

```
class Functor
{
public:
    virtual double operator()(double) const =0;
    virtual ~Functor(){};
};
```

У похідному класі повинна бути реалізація віртуального оператора, наприклад, так

```
class Elliptic: public Functor
{
private:
    double _a, _b;
public:
    Elliptic (double a=1, double b=2):_a(a),_b(b){};
    ~Elliptic(){};

    double operator()(double x) const
    {
          return 1.0
                /sqrt(_a*_a*cos(x)*cos(x)
                +(_b*_b)*(sin(x)*sin(x)));
    }
};
```

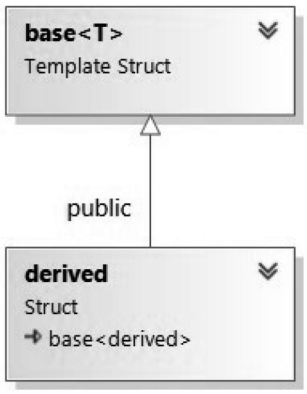


**Рис. 1.** Дивно рекурсивний шаблон

Але існує спосіб уникнення використання віртуальних функцій. Як приклад, розглянемо так званий «дивно рекурсивний шаблон» побудови неполіморфної ієрархії класів функторів. Рис. 1 пояснює будову шаблону, у якому базовий клас ієрархії параметризується похідним класом. Така архітектура дає змогу уникнути віртуальності в похідному класі.

Варіант застосування для статичної ієрархії функторів.

```
template<typename AnyFunctor>
struct Functor
{
    AnyFunctor& _functor;
    Functor(AnyFunctor& functor) :_functor(functor) {}
    const double operator()(double x) const
    {
          return _functor(x);
    }
};

class Elliptic : public Functor<Elliptic>
{
private:
    double _a, _b;
public:
    Elliptic(double a = 1, double b = 2):_a(a),_b(b),
              Functor<Elliptic>(*this){}
    ~Elliptic() {}

    double operator()(double x) const
    {
          return 1.0
                / sqrt(_a * _a * cos(x) * cos(x)
                + (_b * _b) * (sin(x) * sin(x)));
    }
};
```

Використання неполіморфного функтора дозволяє його відкритий (inline) виклик.

```
template<class AnyFunctor>
inline double process(const Functor<AnyFunctor>& f, double x)
{
    return f(x);
}
Приклад виклику:
Elliptic el(1, 2);
cout << process(el, 3) << endl;
```

Такий гібридний підхід дозволяє за певних умов досягти ефекту, який зазвичай потребує динамічного поліморфізму, без втрати швидкодії, до якої призводить виклик віртуальних функцій. Усе те саме, але без виклику віртуальної функції.

Іншим прикладом може бути використання статичного поліморфізму для вирішення проблеми подвійної диспетчеризації [1].

## Сучасна критика ООП

Тож чи справді Дейкстра настільки критично висловився про ООП? І якщо так, то в чому могла бути суть критики?

Такий погляд, імовірно, пов'язаний із тим фактом, що Дейкстра працював над можливістю формального доведення коректності процедурних програм за допомогою логіки Флойда — Хоора. ООП, на відміну від процедурного, не має за собою формального математичного апарату, і тому важче формально аргументувати коректність об'єктно-орієнтованих програм. Але навряд чи це єдина або вичерпна причина.

Спробуємо уявити, якою б могла бути більш детальна критика сучасних реалізацій ООП.

Проблеми починаються з самого визначення парадигми. На жаль, поширені «доступні» визначення надто загальні, і спільні ознаки зводяться до понять класу і об'єктів. Мабуть, найбільш відоме визначення — формула «ООП = Інкапсуляція + успадкування + поліморфізм» — зовсім не відповідає суті парадигми, а лише перелічує окремі атрибути, які часто притаманні її реалізаціям. Найбільше уваги, як правило, приділяють успадкуванню, яке на практиці виявляється найбільш проблематичною рисою.

Спершу дамо визначення парадигмі ООП.

- ООП ґрунтується на об›єктах, пов'язаних ієрархіями вкладення, а не на алгоритмах.
- Кожен об'єкт суть екземпляр певного класу.
- Класи утворюють ієрархію успадкувань.

### Елементи об'єктної моделі (за Г. Бучем [5])

- Абстрагування
- Інкапсуляція
- Модульність
- Ієрархічність
- Типізація
- Паралелізм
- Збережність

Ключова властивість ООП — ***абстрагування***, тобто можливість визначення нових абстракцій. Інші елементи певною мірою слугують для забезпечення абстрагування.

Інкапсуляція забезпечує вищий рівень абстракції цілої конструкції порівняно з рівнем її складових, виділяючи її істотні характеристики.

**Пошук адекватних абстракцій — головне завдання об'єктно-орієнтованого програмування.**

*This is why the most important benefit of classes in C++ for us is not the inheritance mechanisms but the ability to establish new abstractions and to provide alternative realizations for them* (Peter Gottschling [10]).

### Стереотипи

ООП страждає від кількох поширених стереотипів, які значно ускладнюють його розуміння новачками.

***«Все є об'єктом».***

Імовірно, така фраза — це результат того, що деякі «чисті» ООП-мови змушують визначати всі функції в межах класів і автоматично загортають примітивні типи даних в об'єкти.

***«Об'єкти взаємодіють між собою».***

Під взаємодією мають на увазі те, що метод одного об'єкта може викликати метод іншого. Але ця взаємодія не визначає керування ходом виконання програми — об'єкти не є автономними підпрограмами, кожен з власним стеком виконання програми — це ознака потоків, а не об'єктів.

### Надлишковість

Порівняно з процедурними, ООП мовам часто притаманна надлишковість, яка найпростіше демонструється прикладом Hello World на Java:

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

У цій «чистій» ООП програмі немає жодного об'єкта, як і інкапсуляції, успадкування чи поліморфізму. Мінімальна ООП програма не є об'єктно-орієнтованою, бо ООП — це факультативний набір абстракцій, який можна поєднати з іншими, більш базовими парадигмами.

### Успадкування і абстрактні класи

Успадкуванню класів приділяють більше уваги, ніж будь-якому іншому інструменту в ООП, і ним зловживають найбільше. Це потужний інструмент, який, як правило, важко використати правильно

і важко прочитати в готовому коді. Тимчасом як ООП парадигма має на меті зробити проєктування програм простішим, надаючи можливість зосередитись на вищому рівні абстракції, успадкування класів на практиці робить програми складнішими. Тому виникають підстави для критики.

- Успадкування — механізм визначення ієрархій класів, а не повторного використання. Успадкування часто використовують, щоб надати новому класу певний функціонал наявного. Але на практиці завжди краще виокремити потрібний функціонал в абстракцію і агрегувати її.
- Успадкування ускладнює читання і налагодження програм замість того, щоб спрощувати.
- Абстрактні класи завжди краще замінити інтерфейсами.
- Перевизначення методів (override) майже завжди порушує принцип підстановки Лісков (Liskov Substitution Principle). Під час перевизначення дуже легко порушити інваріанти базового класу.
- Успадкування може призвести до конфлікту сигнатур.
- Множинне успадкування — ще більш складний, і, як наслідок, ризикований інструмент.

Квінтесенція антипатернів успадкування — це абстрактний клас, який містить спільну реалізацію, але окремі функції залишає для визначення похідним класам.

```
public abstract class MyAbstractClass
 {
     public int DoWork(int value)
     {
         int preProcessed = this.PreProcess(value);
         int processed = this.Process(preProcessed);
         int postProcessed = this.PostProcess(processed);
         return postProcessed;
     }
     protected abstract int Process(int value);
     private int PreProcess(int value) { … }
     private int PostProcess(int value) { … }
 }
```

Однак така архітектура швидко наражається на ряд проблем:

- Неможливо суттєво змінити хід виконання у похідних класах, можна лише визначити окремий функціонал, передбачений базовим класом.
- Нащадкам легко порушити контракт, особливо якщо клас має змінні атрибути.
- Під час налагодження потрібно перемикатися між різними рівнями абстракції.
- Важко тестувати логіку в базовому класі.
- Важко тестувати логіку в похідних класах.
- Ніякого reuse на практиці.
- Ще гірше, якщо абстрактний метод залежить від атрибутів, визначених у базовому класі.

На практиці завжди краще змінити таке «успадкування» агрегацією чітко визначених абстракцій:

```
public class MyAbstractClass
{
    private readonly IProcessor preProcessor;
    private readonly IProcessor processor;
    private readonly IProcessor postProcessor;
    public MyAbstractClass(IProcessor preProcessor,
           IProcessor processor, IProcessor postProcessor)
    {
       this.preProcessor = preProcessor;
       this.processor = processor;
       this.postProcessor = postProcessor;
    }
    public int DoWork(int value)
    {
        int preProcessed = this.preProcessor.Process(value);
        int processed = this.processor.Process(preProcessed);
        int postProcessed = this.postProcessor.Process(processed);
        return postProcessed;
    }
}
```

### Інтерфейси та незмінність

Повертаючись до конкретних інструментів, наявних в ООП мовах, безумовно, найпотужнішим і найчистішим інструментом для створення абстракцій є інтерфейс. Інтерфейс — *мінімальний* контракт між частинами програми без зайвих умов чи обмежень. Інтерфейс робить можливим поліморфізм і заохочує повторне використання.

Поліморфізм випливає напряму з можливості визначати абстракції через інтерфейси. Схематика поліморфізму описує ідеальну ієрархію класів. Як приклад розглянемо ієрархію узагальнених стеків. Кожен клас другого рівня слугує реалізацією абстракції стеку за рахунок застосування чисто віртуальних функцій, наявних в інтерфейсі.

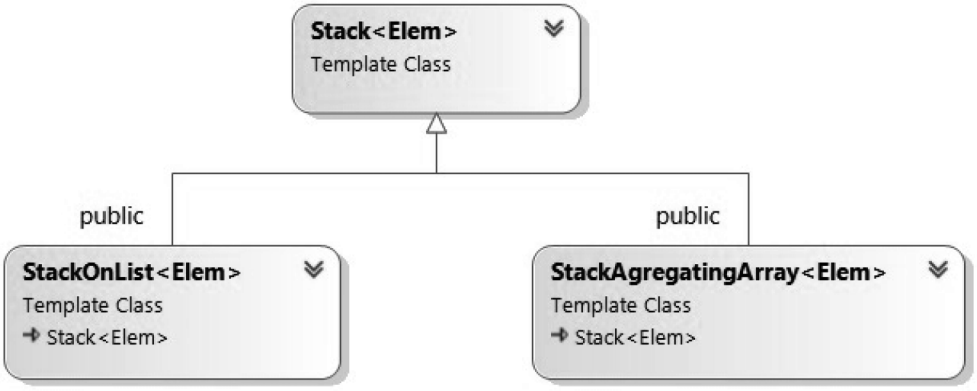


**Рис. 2.** Ієрархія успадкувань стеку

Така ієрархія допускає її доповнення і розширення, не втручаючись до внутрішньої будови її елементів. Наприклад, її можна розширити до конструкції стеку з підгляданням, доповнивши успадкування іншими видами ієрархії, а саме агрегацією.

Важливо розуміти, що між похідними класами не повинно бути успадкування. Успадкуванню підлягає інтерфейс. Інші класи можуть використовувати реалізацію один одного знову ж із застосуванням агрегації.

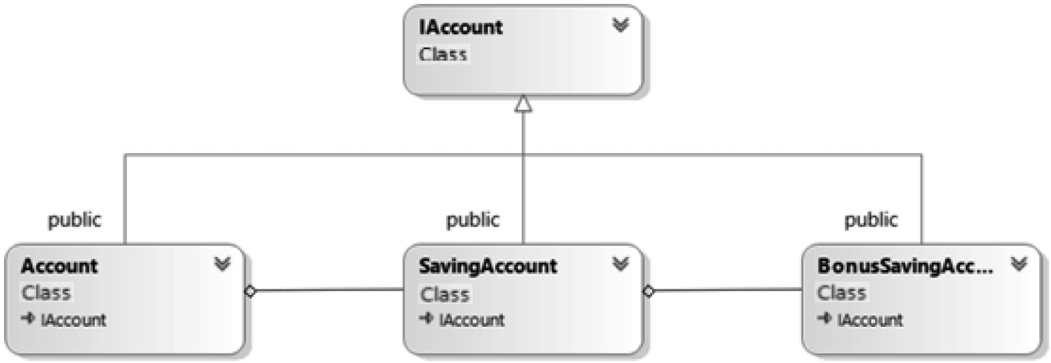


**Рис. 3.** Спільна ієрархія успадкування інтерфейсу і агрегації реалізацій

Існує багато інших корисних властивостей, акумульованих в об'єктно-орієнтованій парадигмі. Одна з найважливіших — це незмінність. Це, мабуть, найкорисніша властивість незалежно від парадигми. Вона значно спрощує читання і налагодження програм, робить можливими програми з багатьма потоками виконання, зближує об'єктно-орієнтовану і функціональну парадигми, особливо на рівні метапрограмування, детальний розгляд можливостей якого виходить за рамки цієї статті. **Все, що може бути незмінним, мусить бути незмінним!**

Нарешті, важливий аспект незмінності — інваріанти, наявність яких переводить прості зібрання даних у класи. Конструктор створює об'єкт у коректному стані, він обов'язково перевіряє коректність значень параметрів. Кожен метод переводить об'єкт з коректного стану в коректний стан. Особливо це стосується пересувної семантики. Після переміщення об'єкт перебуває в невизначено-

му, але дієспроможному стані. Кожен відкритий метод перевіряє коректність значень параметрів. Об'єкт перебуває в коректному стані після аварійної ситуації в одному з методів. Об'єкт перебуває в коректному стані, якщо він використовується багатьма потоками одночасно. Не повинно бути неявних умов на порядок виклику відкритих методів. Якщо компілятор дозволяє викликати метод, то жодних додаткових обмежень не повинно бути.

## Висновки

Об'єктно-орієнтована парадигма після майже шістдесяти років досі залишається популярною, незважаючи на те, що певні її недоліки були добре відомі та задокументовані ще у 1980-х роках [6]. Наведена вище критика стосується як концепції ООП, так і конкретних інструментів, які стали стандартними в більшості об'єктно-орієнтовних мов. Тим не менш, певні ідеї, зароджені в ООП, довели свою ефективність і здатність бути застосованими за нових умов.

Поза сумнівом, найважливіша риса ООП — це можливість створення абстракцій за допомогою механізмів мови, адже вона дозволяє масштабувати розробку програм, водночас роблячи їх простішими для розуміння. Значна частина критики ООП походить якраз від того, що деякі мови роблять такий механізм обов'язковим для всіх програм, що не завжди є виправданим. Але збалансоване створення абстракцій із дотриманням належних принципів організації архітектури програмного забезпечення слугує явним свідченням дієвості об'єктно-орієнтованої парадигми.

Найбільш вдалий механізм створення абстракцій, який походить від об'єктно-орієнтованої парадигми — інтерфейс, адже він дозволяє виокремити необхідний контракт між частинами програми, при цьому не ускладнюючи їх, на відміну від успадкування. Успадкування класів зазвичай ускладнює програми, і в більшості випадків може бути замінене агрегацією класів.

Механізми абстрагування, які надає ООП, часто підсилюються іншими парадигмами програмування. У статті наведено приклади вдалого поєднання ООП з узагальненим програмуванням у мові C++, яке дозволило досягти результатів, які раніше були недосяжними цими парадигмами поодинці.

### *Список літератури*


1. Бублик В. В. До питання створення статичного патерну проєктування для подвійної диспетчеризації модельних сигнатур / В. В. Бублик // Наукові записки НаУКМА. Комп'ютерні науки. — 2021. — Том 4. — С. 64–71.
2. Бублик В. В. Деякі особливості застосування об'єктно-орієнтованої парадигми [Електронний ресурс] / В. В. Бублик, Д. Р. Фітель // Теоретичні та прикладні аспекти побудови програмних систем: праці 15 міжнародної науково-практичної конференції, Київ, 23–24 грудня 2024. — Режим доступу: https://taapsd.ukma.edu.ua/.
3. Ющенко Е. Л. Адресное программирование / Е. Л. Ющенко. — Киев : Техн. лит., 1963. — 286 с.
4. Alexandrescu A. Modern C++ Design: Generic Programming and Design Patterns Applied / A. Alexandrescu. — Addison Wesley, 2001. — 352 p.
5. Booch G. Object-Oriented Analysis and Design with Applications / G. Booch. — 3rd edition. Addison-Wesley Professional, 2007. — 720 p.
6. Crawford B. Object-Oriented Programming: The Good, the Bad, and the Ugly / B. Crawford // TUG Lines. — 1989. — Vol. 32. — Aug.–Sep. — Pp. 7–11.
7. Dahl O.-J. Structured Programming / O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare. — Academic Press, 1972. — 220 p.
8. Dijkstra E. W. Hoe wiskundig programmeren is [Electronic resource] / E. W. Dijkstra // EWD 261. — Mode of access: https://www.cs.utexas.edu/~EWD/transcriptions/EWD02xx/EWD261.html.
9. Fortran. Programmer's Reference Manual. The Fortran Automatic Coding System for the IBM 704 EDPM [Electronic resource] // IBM Corp. — 1956. — 51 p. — Mode of access: http://bitsavers.informatik.uni-stuttgart.de/pdf/ibm/704/704_FortranProgRefMan_Oct56.pdf.
10. Gottschling P. Discovering Modern C++ / P. Gottschling. — Addison Wesley, 2015. — 472 p.
11. Knuth D. Computer Literacy Bookshops Interview / D. Knuth. — December 7th, 1993.
12. Meyers S. How Non-Member Functions Improve Encapsulation [Electronic resource] / S. Meyers. — Mode of access: https://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197.
13. Nygaard K. The Development of the SIMULA Languages / K. Nygaard, O.-J. Dahl // ACM SIGPLAN Notices. — 1978. — Vol. 13, no. 8. — Pp. 245–272.
14. Stroustrup B. The Design and Evolution of C / B. Stroustrup. — 1st edition. —Addison-Wesley Pub Co, 1994. — 506 p.
15. Sutter H. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices / H. Sutter, A. Alexandrescu. — Addison-Wesley, 2004. — 240 p.


### *References*


Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional.

Booch, G. (2007). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional.

Boublik, V. (2021). Do pytannia stvorennia statychnoho paternu proektuvannia dlia podviinoi dyspetcheryzatsii modelnykh syhnatur. *NaUKMA Research Papers. Computer Science*, *4*, 64–71 [in Ukrainian].

Boublik, V., & Fitel, D. (2024). Deiaki osoblyvosti zastosuvannia obiektno-oriientovanoi paradyhmy. *Teoretychni ta prykladni aspekty pobudovy prohramnykh system: pratsi 15 mizhnarodnoi naukovo-praktychnoi konferentsii*. Kyiv, December 23–24, 2024. https://taapsd.ukma.edu.ua/ [in Ukrainian].

Crawford, B. (1989). Object-Oriented Programming: The Good, the Bad, and the Ugly. *TUG Lines*, *32*, 7–11.

Dahl, O.-J., Dijkstra, E. W., & Hoare, C. A. R. (1972). *Structured Programming*. Academic Press, 1972.

Dijkstra, E. W. Hoe wiskundig programmeren is. *EWD 261*. https://www.cs.utexas.edu/~EWD/transcriptions/EWD02xx/EWD261.html.

Fortran. Programmer's Reference Manual. The Fortran Automatic Coding System for the IBM 704 EDPM. IBM Corp. 1956. http://bitsavers.informatik.uni-stuttgart.de/pdf/ibm/704/704_FortranProgRefMan_Oct56.pdf.

Gottschling, P. (2015). *Discovering Modern C++*. Addison Wesley, 2015.

Knuth, D. (1993) *Computer Literacy Bookshops Interview*. December 7.

Meyers, S. (2000). *How Non-Member Functions Improve Encapsulation*. https://www.drdobbs.com/cpp/how-non-member-functions-improve-encapsu/184401197.

Nygaard, K., & Dahl, O.-J. (1978). The Development of the SIMULA Languages. *ACM SIGPLAN Notices*, *13* (8), 245–272.

Stroustrup, B. (1994). *The Design and Evolution of C*. Addison-Wesley Pub Co.

Sutter, H., & Alexandrescu, A. (2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley.

Yushchenko, E. L. (1963). *Adresnoe programmyrovanye*. K. Tech. Lit. [in Russian].

*V. Boublik, D. Fitel*

## OBJECT-ORIENTED PARADIGM: PROS AND CONS

*This paper critically examines the object-oriented programming (OOP) paradigm, tracing its theoretical foundations and evaluating its most widely adopted modern implementations. Particular attention is given to the criticisms that have emerged over time, notably those stemming from Edsger Dijkstra's remark that "object-oriented programming is an exceptionally bad idea…." We aim to interpret and contextualize Dijkstra's dissatisfaction with contemporary OOP, offering a practical perspective on its implications.*

*We begin by exploring the origins of the object-oriented paradigm, highlighting its evolution from procedural programming and the fundamental conceptual shift it introduces—namely, the incorporation of programmable abstractions. Given OOP's widespread adoption across numerous programming languages, we distill its core principles and examine their shared characteristics. Additionally, we provide a refined definition of the object-oriented paradigm based on these key principles, emphasizing its role as a powerful and flexible mechanism for creating abstractions, rather than a simplistic combination of more specific programming language features.*

*Our analysis also outlines the advantages and limitations inherent in common OOP principles, including the SOLID principles. We present specific recommendations, along with examples of both successful and flawed implementations of various OOP concepts. Importantly, we stress that the object-oriented paradigm functions best as an optional tool for abstraction rather than an imposed constraint.*

*Finally, we examine the paradigm's most effective attributes, particularly its ability to coexist with and enhance other programming methodologies. To illustrate this adaptability, we present a case study on template metaprogramming in C++, demonstrating how OOP can facilitate the creation of sophisticated custom abstractions with precise control over their behavior and degrees of reusability that are not achievable by a single programming paradigm.*

**Keywords**: programming paradigm, object-oriented paradigm, generic programming, template metaprogramming.

*Матеріал надійшов 28.05.2025*