УДК 519.681.2
DOI https://doi.org/10.32689/maup.it.2025.3.18

Олексій ПІСКУНОВ
кандидат фізико-математичних наук, старший науковий співробітник,
доцент кафедри прикладної математики,
Державне некомерційне підприємство
«Державний університет «Київський авіаційний інститут»,
oleksii.piskunov@npp.kai.edu.ua
***ORCID:** 0000-0002-9200-3422*

Валерій ХРЕБЕТ
кандидат фізико-математичних наук, доцент, доцент кафедри прикладної математики,
Державне некомерційне підприємство
«Державний університет «Київський авіаційний інститут»,
valerii.khrebet@npp.kai.edu.ua
***ORCID:** 0000-0002-0191-1768*

Наталя ТУПКО
кандидат фізико-математичних наук, доцент, доцент кафедри прикладної математики,
Державне некомерційне підприємство
«Державний університет «Київський авіаційний інститут»,
natalia.tupko@npp.kai.edu.ua
***ORCID:** 0000-0002-5432-5498*

АРИФМЕТИЧНІ ОБЧИСЛЕННЯ ТА НЕБЕЗПЕЧНІСТЬ УНІВЕРСАЛЬНОГО ПОЛІМОРФІЗМУ

Анотація. *У статті розглянуто алгебраїчний підхід до проєктування та тестування програмного забезпечення.*

Метою статті *є розробка мовою C# двох класів: цілих та дійсних чисел, які знаходяться у відношенні наслідування. При цьому перевизначена операція ділення в кільці цілих чисел та полі дійсних чисел повинна призводити до помилок у розрахунках поліморфних функцій. І це незалежно від того, який із класів є базовим, а який – похідним. Зазначені класи будуть використовуватися для демонстрації виникнення неочевидних помилок під час обчислень у поліморфній реалізації симплекс-методу. У цій роботі розроблені класи були протестовані та застосовані для отримання очевидної помилки в результаті виконання дуже простої поліморфної функції. Усе це демонструє небезпеку універсального поліморфізму при арифметичних обчисленнях.*

Методи дослідження. *Під час дослідження використовуються базові положення методу формальної розробки RAISE та методу проєктування за контрактом Бертрана Мейєра, які дозволяють застосовувати формальну логіку до класів, що розробляються.*

Наукова новизна *дослідження полягає в тому, що на функціях дійсного аргументу були перевірені формальні ознаки, за допомогою яких можна передбачати появу збоїв у працюючому програмному забезпеченні. По-перше, це підтверджує корисність застосування розглянутих формальних методів проєктування програмного забезпечення. По-друге, навіть на найпростіших типах даних продемонстровано небезпечність універсального поліморфізму для арифметичних обчислень, який використовується у мові C#. Аналогічні приклади небезпеки поліморфізму гарантовано можна отримати при наслідуванні між дійсними та комплексними числами.*

Висновки. *Алгебраїчне проєктування та тестування базується на математичних принципах, що дозволяє: уникати двозначності й неоднозначності в описі функціональності; забезпечувати точність та однозначність у формулюванні вимог до програми; виявляти й усувати помилки ще на стадії розробки.*

Ключові слова: *універсальний поліморфізм, RAISE Specification Language, аксіоматика, функція дійсного аргументу.*

Oleksii PISKUNOV, Valerii KHREBET, Natalia TUPKO. ARITHMETIC COMPUTATIONS AND THE NON SAFETY OF UNIVERSAL POLYMORPHISM

Abstract. *The article examines an algebraic approach to software design and testing.*

The purpose *of the study is to develop, in C#, two classes – integers and real numbers – that are in an inheritance relationship. In this setting, the overridden division operation in the ring of integers and the field of real numbers must lead to errors in the calculations of polymorphic functions, regardless of which class is the base and which is the derived one. These classes are intended to demonstrate the occurrence of non-obvious errors during computations in a polymorphic implementation of the simplex method. In this work, the developed classes were used to produce an explicit error as the result of executing a very simple polymorphic function. All of this demonstrates the danger of universal polymorphism in arithmetic computations.*

Research methods. *The study employs the basic principles of the RAISE formal development method and Bertrand Meyer's Design by Contract methodology, which make it possible to apply formal logic to the classes under development.*

The scientific novelty *of this study lies in the fact that, for functions of a real variable, formal indicators were tested that make it possible to predict the occurrence of failures in operational software. First, this confirms the usefulness of applying the considered formal methods of software design. Second, even for the simplest data types, the study demonstrates the dangers of universal polymorphism used in C# for arithmetic computations. Similar examples of the risks associated with polymorphism can certainly be obtained in cases of inheritance between real and complex numbers.*

Conclusions. *Algebraic design and testing are based on mathematical principles, which makes it possible to: avoid ambiguity and vagueness in the description of functionality; ensure precision and unambiguity in formulating program requirements; detect and eliminate errors already at the development stage.*

Key words: *universal polymorphism, RAISE Specification Language, axiomatics, function of a real argument.*

Постановка проблеми та аналіз останніх досліджень і публікацій. Проєкт виконано в межах розробки курсу з модульного тестування [3], у рамках якого було розглянуто наступні питання:

- розробка та відлагоження двох класів для демонстрації небезпечності універсального поліморфізму на прикладі симплекс методу;
- побудова простішої поліморфної функції для демонстрації небезпеки універсального поліморфізму, що порушує принцип підстановки Лісков;
- дослідження можливостей мови C#, які дозволяють розробляти приклади порушення принципу підстановки, виявляти та передбачати їх появу.

Згідно з принципом підстановки [13], якщо доведено, що певна властивість виконується для всіх об'єктів *b* типу *B*, то ця властивість повинна виконуватися і для всіх об'єктів *d* типу *D*, де тип *D* є підтипом *B*. Далі, під такою властивістю розумітимемо наступне: нехай функція *P*, яка має в якості формального параметра деякий об'єкт класу *B*, відповідає своєї специфікації для будь-якого об'єкту цього класу. Нехай тепер *D* – клас, похідний від класу *B*. Якщо для деякого об'єкта d похідного класу *D* застосування функції *P* до цього об'єкта *P(d)* перестає задовольняти специфікації цієї функції, то класи *B* і *D* мають різні типи, хоча *D* є похідним класом для *B.* Цей результат прямо суперечить такій властивості мови програмування як універсальний поліморфізм. Під поліморфізмом будемо розуміти властивість мови програмування, яка дозволяє виразу задовольняти вимогам декількох типів одночасно. Згідно з Карделлі [9] виділятимемо чотири види поліморфізму:

- спеціальний поліморфізм перетворення;
- спеціальний поліморфізм перевантаження функцій;
- універсальний поліморфізм включення підтипів;
- універсальний параметричний поліморфізм шаблонів.

На відміну від спеціального поліморфізму, універсальний дозволяє писати код (розробляти функції) який має коректно виконуватись на ще не створених класах. Ця властивість повинна суттєво економити зусилля програмістів унаслідок більш широкого повторного використання коду.

Представлені у статті класи є контр прикладами для обох універсальних поліморфізмів. Вони порушують принцип підстановки і тому демонструють їх небезпечність. Один із найвідоміших прикладів порушення принципу підстановки Лисків є приклад Р. Мартіна [14], в якому показано, що клас квадратів не можна успадковувати від класу прямокутників. У наведеній моделі спадкування порушувалася робота нібито поліморфної функції (працювала некоректно) і тому, згідно з принципом підстановки, клас квадратів (який розробив для цього прикладу Р. Мартін) не задовольняв вимогам класу прямокутників. При цьому, у своїй доповіді Р. Мартін не давав пояснень, чому в цьому випадку не можна використовувати універсальний поліморфізм і як передбачати можливі проблеми до виявлення помилки під час виконання. Головним наслідком цього прикладу виявилося те, що універсальний поліморфізм небезпечний і засоби статичної типізації мови C++, втім як і C#, не вирішують завдання запобігання помилкам проектування. Однак, відповідно до свого опису мови компілятор C++ трактував, що об'єкт класу квадрат задовольняв вимогам двох типів: вимогам похідного класу квадратів і базового класу прямокутників.

Поліморфізм включення підтипів мови C# так само відносить будь-який похідний клас до типу базового класу. Згідно з документацією Microsoft [2] до мови C#:

Під час виконання об'єкти похідного класу можуть бути оброблені так само, як і базові об'єкти класу в таких місцях, як параметри методу.

А також [1]:

Зазвичай наслідування використовується для вираження зв'язку «є» між базовим класом і ... похідним класом. Похідний клас – це тип базового класу.

Таким чином, ця властивість мови C# дозволяє змінним базового класу присвоювати об'єкти похідного класу:

```
B b = new D( );
```

І саме ця властивість мови дозволяє передавати об'єкти похідного класу у функції, які очікують об'єкти базового класу. Більш докладно формулювання універсального поліморфізму для мови C# можна подивитися в специфікації мови [10, 12.6.2.3 Run-time evaluation of argument lists] та [10, 10.2.12 Implicit conversions involving type parameters].

Виклад основного матеріалу. У роботі [4] за допомогою ідеї алгебраїчного проектування класу, яка була запропонована в дослідженні [11], проведено формальний аналіз прикладу Р.Мартіна [14] та, в термінах попарного застосування методів класу, показано причину невідповідності типів. Спочатку кожному методу класу ставиться у відповідність його функціональний тип. Це можна зробити за твердженням Б. Мейєра: «Клас – це запрограмований абстрактний тип даних» [15]. Таким чином, кожному методу класу ставиться у взаємооднозначну відповідність функція абстрактного типу даних. Функціональний тип цієї функції і вважатиметься функціональним типом методу. Потім усі функціональні типи методів деякого класу *B* поділяються на чотири групи в такий спосіб. Нехай *domen(B)* означає множину об'єктів класу *B*, а *X* та *Y* – довільні множини (включаючи порожні множини). Тоді маємо наступні групи функціональних типів:

- множина об'єктів класу відсутня у функціональному типі, це статичні методи класу: X → Y;
- множина об'єктів класу присутні тільки з правого боку від функціональної стрілки, це конструктори класу: X → *domen(B)* × *Y;*
- множина об'єктів класу присутня тільки з лівого боку від функціональної стрілки, це методи, які витягують з об'єкта деяку інформацію: (так звані геттери, від слова get): *domen(B)* × X → *Y;*
- множина об'єктів класу присутня з обох сторін функціональної стрілки, це методи, які змінюють стан об'єкта (так звані сеттери, від слова set): *domen(B)* × X → *domen(B)* × *Y.*

Далі вимоги до функцій типів обох класів виписувалися в алгебраїчному вигляді. Тобто за допомогою попарного застосування функцій. Причиною порушення принципу підстановки у Р. Мартіна були несумісні одна з одною вимоги четвертої групи методів. На проблеми, пов'язані з методами четвертого функціонального типу, окремо вказували Б. Мейєр [15] та Л. Карделлі [8]. Наявність таких методів не дозволяє змінювати множину об'єктів похідного класу внаслідок одночасної ко- та контра- варіантності цих методів. Це негайно тягне за собою такий факт, що множина об'єктів класу квадратів (у разі комп'ютерів вона природно кінцева) має збігатися з множиною об'єктів прямокутників. Але, у разі прикладу Р.Мартіна, причиною були інші вимоги до попарного застосування методів. Ці вимоги були несумісні одна з одною для класів квадратів та прямокутників. І такого роду алгебраїчні вимоги до передумов, постумов та інваріантів абстрактного типу даних (як це пропонується в роботах [15; 13] та [11]):

- не можуть бути в принципі перевірені на відповідність поточними компіляторами C++ і C#;
- повністю відповідають аксіомам таких алгебраїчних систем, як група та напівгрупа [16].

Таким чином, використовуючи метод алгебраїчного запису для вимог при проєктуванні свого абстрактного типу даних, можна легко будувати приклади порушення принципу підстановки або передбачати його можливе порушення. Особливо легко це робити для четвертої групи функціональних типів (сеттерів). Оскільки тип будь-якої операції алгебри потрапляє в четверту групу функціональних типів і має несумісні вимоги при переходах від напівгрупи до групи, від кільця до поля, то можна легко проєктувати свої приклади порушення принципу підстановки і небезпеки універсального поліморфізму. До речі, це має бути очевидним для програмістів, які опанували загальне програмування (тобто універсальний поліморфізм) за монографією алгебраїста А.А.Степанова [16].

Детальніше про алгебраїчне проєктування програмного забезпечення (ПЗ) представлено в роботі [7]. Крім того, у звіті [6] представлено один з найпростіших прикладів алгебраїчного проєктування, реалізованого за допомогою аксіом математика 19-го століття Германа Грассмана. У звіті [5] наведено приклад з перевизначенням операції віднімання. Операція віднімання є всюди визначеною у разі групи. Це є наслідком існування нейтрального елемента (identity element), операції обернення (inverse operation) та аксіоми скорочення (cancellation axiom) у групі. Для ілюстрації формулювання алгебраїчних вимог до типів даних, перепишемо аксіоматику групи мовою RAISE Specification Language [17] згідно з [16].

Listing 1. Аксіоматика групи:

```
scheme group =   class
    type
       T
    value
        e     : T         -- identity element
      ,op     : T >< T -> T -- group operation
```


ISSN 2786-5460 (Print), ISSN 2786-5479 (Online)

```
      ,inverse : T -> T     -- inverse operation
  axiom
       [associativity]
    all x,y,z : T :-  op (x, op(y,z)) is op (op(x,y), z)
     ,[identity]
       all x : T    :- (op (x, e) is x ) /\ (op (e, x) is  x)
     ,[cancellation]
        all x : T    :-  op (x, inverse(x)) is e
  end
```

З цієї аксіоматики випливає, що через відсутність операція обернення та аксіоми скорочення в напівгрупі, операція віднімання не є всюди визначеною в межах цієї алгебраїчної структури. Зокрема, у прикладі з відніманням [5] порушення принципу підстановки приводило до аварійного завершення роботи додатку, що має бути особливо помітним розробникам ПЗ. У поточній роботі наводиться аналогічний приклад при перевизначені операції ділення, що призводить до явно некоректного результату. Така поведінка, хоча й менш критична порівняно з аварійним завершенням роботи додатку, все ж є досить очевидною та зрозумілою для програмістів. У наступному прикладі, що передбачає застосування симплекс-методу, отриманий результат буде неправильним, але таким, що мало відрізняється від правильного. Для розробників останній приклад некоректної поведінки при реалізації універсального поліморфізму є ознакою найгіршого розвитку подій.

Розглянемо код прикладу**.** Спочатку подамо обидві форми функції *P* для універсального поліморфізму. Функція зобов'язана перетворити результат обчислення в ціле число за допомогою традиційного відкидання дробової частини.

Listing 2. Функція для демонстрації поліморфізму включення:

```
using System;
partial  class Program    {
      public
      static int P ( real one,  real two, real divisor) {
           real out1 = (one * two ) /  divisor;
           return (int) out1;
       }
}
```

Версію для параметричного поліморфізму розроблено в термінах додаткового абстрактного базового класу *number*. Клас *real* є похідним класом від *number* і базовим для класу *integer*. Друга версія функції так само, як і перша, працює з очікуваною помилкою з об'єктами типу *integer*.

Listing 3. Функція для демонстрації параметричного поліморфізму:

```
using System;
partial  class Program {
       static int
       P<T> ( T one,  T two, T divisor) where T : number  {
            number out1 = (one * two ) / divisor;
            return (int) out1;
       }
}
```

Таким чином, згідно з вимогами специфікації, очікується, що якщо функції передати значення 5.0, 3.0 і 2.0 як параметри, то функція *P* повинна повернути число 7 (внаслідок (5.0 * 3.0)/2.0 =7.5). Далі, наведемо часткове визначення класу дійсних чисел, що розробляється *real*.

Listing 4. Методи базового класу:

```
partial class real   {
      public double v;
      public virtual real sum( real r){ return new real (v+r.v);}
      public virtual real dif( real r){ return new real (v-r.v);}
      public virtual real mul( real r){ return new real (v*r.v);}
      public virtual real div( real r){ return new real (v/r.v);}
}
```

Для арифметичних обчислень клас містить чотири віртуальні функції: додавання, віднімання, множення та ділення. Модифікатор *virtual* у методів дозволяє перевизначити їх у похідному класі з можливістю пізнього зв'язування. Самі арифметичні обчислення виконуються за допомогою вбудованих

у мову C# операцій над об'єктами класу *double*, до якого належить поле *v*. Потім, для цих методів вводяться чотири природні позначення, які дають можливість записувати алгоритми з об'єктами класу *real* у звичному інфіксному запису, тобто через операції.

Listing 5. Операції базового класу:

```
partial class real   {
      public static real operator + (real l, real r) { return l.sum(r); }
      public static real operator – (real l, real r) { return l.dif(r); }
      public static real operator * (real l, real r) { return l.mul(r);}
      public static real operator / (real l, real r) { return l.div(r);}
}
```

Після цього функція *P* була побудована як динамічна підвантажувана бібліотека. Для її тестування було створено окремий тестовий варіант, який успішно пройшов перевірку.

Listing 6. Перший тестовий варіант для функції *P*:

```
using System;
using NUnit.Framework;
public partial class demo {
      [Test]
public void integerUsage() {
  Assert.That(
      Program.P(new integer(5), new integer(3), new integer(2)), Is.EqualTo(7)
       );
     }
}
```

Результат виконання цього тестового варіанта:

```
 Test Run Summary
 Overall result: Passed
 Test Count: 1, Passed: 1, Failed: 0, Warnings: 0, Inconclusive: 0, Skipped: 0
```

Як бачимо, тест виконано успішно, отже, можна розпочинати промислову експлуатацію побудованої динамічної бібліотеки.

Зазначимо, що функція *P* є поліморфною за визначенням вище, оскільки задовольняє вимогам кількох типів:

- типу, в термінах якого її розроблено:

domen(real) × domen(real) × domen(real) → domen(int)

де клас *int* – клас цілих чисел, вбудований у мову C#;

- і, згідно з офіційною документацією, типу будь-якого класу похідного від класу *real*, наприклад, запланованого класу *integer:*

domen(integer) × domen(integer) × domen(integer) → domen(int).

Тепер визначаємо похідний клас цілих чисел *integer*, в якому використовується функція *round* (див. нижче округлення до цілих чисел). Вона розроблена відповідно до стандарту [12], який регламентує використання дійсних величин і задає кілька різних напрямків округлення до цілого.

Listing 7. Конструктори похідного класу:

```
using System;
partial  class integer: real {
      public integer (): base(0.0) {}
      public integer (int v)      { this.v = v;}
      public integer (double v)   { this.v = round(v);}
      public integer (string s)   {
       if (!double.TryParse (s, out v))
            this.v = 0.0;
        else
             this.v = round(v);
      }
}
```

Як видно з коду, усі конструктори гарантують появу цілочисельного значення в полі *v*. Переходимо до перевизначеня методів базового класу.

Listing 8. Перевизначені методи похідного класу:

ISSN 2786-5460 (Print), ISSN 2786-5479 (Online)

```
partial class integer: real {
  public  override  real sum( real r){
    return new integer (this.v + round(r.v));
  }
  public  override  real dif( real r){
    return new integer (this.v – round(r.v));
  }
  public  override real mul( real r){
    return new integer (this.v * round(r.v));
  }
  public  override  real  div( real r){
    return new integer (round( this.v / round(r.v)));
  }
}
```

Методи додавання, віднімання та множення не викликають питань. Вхідний параметр округлюється до цілого та виконується відповідна операція з цілим числом, яке зберігається в полі *v* поточного об'єкта *this*. Результати обчислення всіх трьох функцій повинні залишатися цілочисельними з урахуванням обмежень стандарту [12] щодо подання чисел у змінних типу *double.* Застосування метода ділення *div* може в результаті роботи привести до дробового числа, яке округлятиметься до цілого функцією *round*. Тобто засобами класу *integer*. Однак при розробці цього класу було використано рекомендацію стандарту [12] і для округлення вибрано банківське округлення. У цьому прикладі це значення *ieeeRound.toEven*. Тоді на даних тестового варіанта 6 отримаємо значення 15/2 = 7.5. І воно буде округлено до парного цілого числа, тобто до 8.

Listing 9: Другий тестовий варіант для функції *P:*

```
using System;
using NUnit.Framework;
public partial class demo {
  [Test]
 public void integerUsage() {
   Assert.That(
      Program.P(new integer(5), new integer(3), new integer(2)), Is.EqualTo(7)
   );
  }
}
```

В цьому випадку відкидання дробової частини функції P нічого не змінить. Після цього виконання тестового варіанта закінчиться відмовою:

```
Test Run Summary
Overall result: Failed
Test Count: 2, Passed: 1, Failed: 1, Warnings: 0, Inconclusive: 0, Skipped: 0
Failed Tests – Failures: 1, Errors: 0, Invalid: 0
```

Саме це й було передбачено до початку розробки класу *integer* в роботі [4]. При тестуванні версії функції P для випадку параметричного поліморфізму (дивитися Listing 3) результати виявилися аналогічними випадку з похідним класом. Крім того, тестування показує, що операції базового класу (дивитися Listing 5) в результаті пізнього зв'язування викликають методи похідного класу.

Розглянемо округлення до цілих чисел та .NET. Перетворення довільного дійсного значення до цілої змінної (надалі називатиметься округленням) описане у стандартах [12], починаючи з 1985 року. У кожному зі стандартів задається кілька різних напрямків округлення (rounding direction). Зокрема, у стандарті 2015 року їх п'ять:

- округлення в напрямку до нуля. Виконується відкиданням дробової частини значення з плаваючою комою. Звичне округлення в багатьох мовах програмування, зокрема, у давній мові Фортран-4;
- округлення в напрямку від нуля. Виконується до найближчого цілого числа. Якщо дробова частина значення з плаваючою комою дорівнює 0.5, то додатне значення округлюється до більшого цілого числа, а від'ємне — до меншого цілого;
- округлення в напрямку до найближчого цілого (банківське округлення). У разі, якщо дробова частина значення з плаваючою комою дорівнює 0.5, округлення виконується до найближчого парного цілого числа;
- округлення в напрямку до додатної нескінченності. Значення з плаваючою комою округлюється до більшого найближчого цілого;

• округлення в напрямку до від'ємної нескінченності. Значення з плаваючою комою округлюється до меншого найближчого цілого;

Стандарт не виділяє жодного з напрямків округлення як особливого. Усі вони однаково допустимі. Крім того, жодним чином не обмежується використання кількох напрямків округлення в межах одного коду. Однак, ймовірно, передбачається використання лише одного. Наприклад, у компіляторах C++ вибір напрямку здійснюється шляхом виклику спеціальної функції *fesetround*.

У .NET немає вбудованого механізму для прямого керування напрямком округлення. Проте клас *Math* надає різні методи, поведінка яких відповідає стандарту [12]. Наступний приклад містить метод *round* для ілюстрації усього сказаного.

Listing 10. Округлення до цілих:

```
using System ;
public enum ieeeRound {
  toZero
  , toAway
  , toEven
  , toPositive
  , toNegative
} ;
partial class integer {
 public static ieeeRound direction = ieeeRound . toEven ;
    protected static double round ( double v ) {
  double rc = 0 . 0 ;
  switch ( direction ) {
   case isoRound.toZero :      rc = Math . Truncate ( v ) ;
                    break ;
  case ieeeRound.toAway :          rc = Math . Round ( v , MidpointRounding .
                                AwayFromZero ) ;
  break ;
  case ieeeRound.toEven :          rc = Math . Round ( v , MidpointRounding .
                                  ToEven ) ;
                                    break ;
  case ieeeRound.toPositive :      rc = Math . Ceiling ( v ) ;
   break ;
  case ieeeRound toNegative :     rc = Math . Floor ( v ) ;
  break ;
  default :                                  rc = Math . Truncate ( v ) ;
                                   break ;
   }
  return rc ;
 }
}
```

Де *direction* – це статичне поле деякого класу *integer*.

Висновки. Проведене дослідження демонструє досягнення поставлених цілей:

• розроблено та налагоджено два класи, які можна використовувати для кодування симплекс-метод в термінах базового;

• представлено приклад простої поліморфної функції *P*, яка демонструє порушення принципу підстановки. Це свідчить про те, що відповідні класи задовольняють вимогам несумісних типів;

• показано, що механізм універсального поліморфізму, який реалізований у мові C# не є безпечним;

• такі можливості мови C# як перевизначене (*override*) легко дозволяють розробляти код, який може порушувати принцип підстановки і маскувати цей код під звичні операції;

• часткові класи мови C# дуже зручні при підготовці документації, тому що дозволяють включати в текст документів тільки необхідну частину коду, не перевантажуючи текст непотрібними деталями.

ISSN 2786-5460 (Print), ISSN 2786-5479 (Online)

Список використаних джерел:

1. Наслідування – C# | Microsoft Corporation. Microsoft Learn [Текст] (рос.). URL: https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/tutorials/inheritance (дата звернення: 07.07.2025).

2. Поліморфізм – C# | Microsoft Corporation. Microsoft Learn [Текст]. URL: https://learn.microsoft.com/ru-ru/dotnet/csharp/fundamentals/object-oriented/polymorphism (дата звернення: 07.07.2025).

3. Піскунов О. Г. Модульне тестування програмного забезпечення: робоча програма. 2024. URL: https://drive.google.com/file/d/1RBR3ORllP-XjA5idzDBxm-kHyjPr_X5d/view.

4. Піскунов О. Г. Про відмінності між поняттями типу і класу. *Вісник Київського університету. Комп'ютерні науки*. 2015. № 3. С. 106–114. URL: https://www.researchgate.net/publication/344177979.

5. Піскунов О. Г., Мічуда А. М. Перевизначення додавання: небезпечне наслідування у групі цілих чисел. 2023. URL: https://www.researchgate.net/publication/366867037.

6. Піскунов О. Г., Рудик В. І., Петренко І. А. Арифметика Пеано: від специфікації до класу. 2022. URL: https://www.researchgate.net/publication/365979331.

7. Піскунов О. Г., Тупко Н. П., Петренко І. А. Алгебраїчне проєктування програмного забезпечення. *Інформаційні технології та суспільство*. 2024. № 5 (11). С. 50–59.

8. Cardelli L., Abadi M. A theory of objects. Springer, 1996. 396 p. URL: ttps://drive.google.com/open?id=1mShdblP3LnooSSfk8Osh_SAtIc54Jczu.

9. Cardelli L., Wegner P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys (CSUR).* 1985. Vol. 17, No. 4. P. 471–523.

10. ECMA International. ECMA-334: C# Language Specification, 7th Edition. Technical Report ECMA-334. December 2023.

11. Haxthausen A. Lecture Notes on The RAISE Development Method. 1999. URL: http://www2.imm.dtu.dk/courses/02263/E20/Files/methodnotes99.pdf.

12. IEEE Standard for Floating-Point Arithmetic. – NY : IEEE Computer Society, 2019. 83 p. URL: https://en.wikipedia.org/wiki/IEEE_754.

13. Liskov B., Wing J. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS).* 1994. Vol. 16, No. 6. P. 1811–1841.

14. Martin R.C. The Liskov Substitution Principle [Електронний ресурс] // C++ Report. March 1996. URL: https://objectmentor.com/resources/articles/lsp.pdf.

15. Meyer B. Object-oriented Software Construction. – 2-nd edition. Santa Barbara : ISE Inc, USA, 2000. 1284 p.

16. Stepanov A. A., Rose D. E. From Mathematics to Generic Programming. Upper Saddle River, NJ : Addison-Wesley/Pearson, 2014.

17. The RAISE Language Group. The RAISE Specification Language. UNU/IIST, Prentice Hall Europe, Denmark, 1992. URL: https://raisetools.github.io/material/documentation/raise-language.pdf.

Дата надходження статті: 05.09.2025
Дата прийняття статті: 20.10.2025
Опубліковано: 04.12.2025