УДК 004.42 DOI 10.18372/2073-4751.87.21690

Піскунов О.Г
orcid.org/0000-0002-9200-3422
e-mail: oleksii.piskunov@npp.kai.edu.ua

Тупко Н.П.
orcid.org/0000-0002-5432-5498
e-mail: natalia.tupko@npp.kai.edu.ua

ЧАСТКОВЕ ЗАСТОСУВАННЯ МЕТОДІВ ФОРМАЛЬНОЇ РОЗРОБКИ

Національний університет «Київський авіаційний інститут»

Вступ

З часу появи у 1969 році [1] першої спроби застосування формальних методів до розробки компілятора PL/I та повної систематизації Віденського методу розробки у 1978 році [2], минуло майже пів століття. Численні дослідники, зокрема Б. Мейєр [3], А. Е. Хакстхаузен [4, 5], Дж. Джеки [6] та інші, неодноразово підкреслювали важливість і практичну цінність застосування формальних підходів у практичній роботі програмістів. Водночас автори оглядових праць 2010 року [7] та нещодавнього дослідження 2023 року [8]) звертають увагу на відсутність помітного прогресу у широкому впровадженні цих методів.

У зв'язку з цим у попередніх роботах автори статті зосередилися на експериментальному застосуванні формальних підходів, зокрема:

– застосувати алгебраїчний метод проєктування ПЗ до розробки простого класу для символьної додавання за специфікацією Г.Грассмана 1861 року [9,10];

– використовувати мову RSL для вирішення деяких питань, пов'язаних з розробкою та тестуванням ПЗ [11, 12].

У цій роботі продовжено зазначені дослідження. Зокрема, демонструється можливість застосування методів формальної розробки до окремого класу програмного проєкту та реалізації його вимог із використанням бібліотеки Guidelines Support Library (GSL) [13].

Постановка проблеми

Постановка проблеми полягає в необхідності отримання практичного результату від часткового застосування методу формальної розробки програмного забезпечення — програмування за контрактом.

Аналіз останніх досліджень і публікацій

З часу появи першого формального методу — Віденського методу розробки програмного забезпечення у 1969 році — і донині такі підходи застосовуються та пояснюються на прикладі повної розробки: від проєктування до завершення розробки. Водночас автори оглядових праць 2010 року [7] та нещодавнього дослідження 2023 року [8] зазначають, що подібний підхід часто відштовхує більшість практикуючих програмістів. Це пояснюється високим порогом входження у формальні методи, а також складністю опанування додаткових спеціалізованих мов.

Мета

Обґрунтувати та експериментально перевірити можливість часткового застосування методів формальної розробки програмного забезпечення (на прикладі програмування за контрактом з використанням бібліотеки GSL) для окремих модулів проекту з метою раннього виявлення відмов у налагоджувальній версії програми без суттєвого впливу на продуктивність фінального додатку.

Інакше кажучи, метою роботи є перевірка гіпотези про можливість

досягнення раннього виявлення помилок шляхом часткового впровадження формальних методів (програмування за контрактом з використанням GSL) для окремих модулів, з мінімізацією впливу на продуктивність кінцевого програмного продукту.

Виклад основного матеріалу

Для демонстрації розглянемо класичний приклад порушення принципу підстановки Лісков [14], який ілюструє проблеми, що виникають при спробі взаємного успадкування класів квадрат і прямокутник, а також проблеми, до яких це наслідування веде. Принцип описує умови, за яких об'єкт похідного класу можна передавати у функцію *g,* яка як формальний параметр очікує об'єкт базового класу і робота функції *g* буде відповідати своїй специфікації. Далі розглянуто реалізацію вимог щодо відомого прикладу порушення принципу підстановки Лісков за допомогою контрактів GSL. Приклад, який з погляду автора, пояснював принцип, був наведений у статті [15] і використаний у його монографії [16].

Зауваження. Очевидно, що, попри поширені твердження в підручниках з об'єктно-орієнтованого програмування, похідний клас не повинен бути базовим класом. Інакше кажучи, він не зобов'язаний відповідати вимогам, висунутим до базового класу. Таким чином, успадкування, загалом, не гарантує збереження цих вимог.

Формальний опис прикладу було дано в [17] як практичне застосування алгебраїчного проектування. Зараз вчинимо так само. Формальні вимоги, які описують зазначені класи прикладу в алгебраїчній формі, представлені в наступних лістингах.

```
  scheme Figure =  class
    type
      F,
      UReal  =  {| r: Real :- r >
0.0 |}
    value
      setWidth  : F >< UReal -> F,
      setHeight : F >< UReal -> F,
      getWidth  : F          ->
UReal,
      getHeight : F          ->
UReal
  end
```

Listing 1: Загальні вимоги до фігур

Згідно [15] у кожному класі присутні чотири методи, які задають/витягують довжину та ширину фігури. Довжина і ширина фігури повинні задаватися дійсними числами більшими за *0.0*.

Далі, напишемо вимоги до попарного (алгебраїчного) застосування цих методів.

```
  Figure
  scheme Rectangle = extend Figure
with class
    axiom
      [Rgh_sw]
        all w: UReal, this: F:-
getHeight(setWidth(this,  w))     =
getHeight(this),
      [Rgw_sw]
  all w: UReal, this: F:- getWidth
(setWidth(this, w))  = w,
      [Rgh_sh]
        all h: UReal, this: F:-
getHeight(setHeight(this, h)) = h,
      [Rgw_sh]
        all h: UReal, this: F:-
getWidth  (setHeight(this,  h))  =
getWidth(this)
  end
```

Listing 2: Вимоги до методів прямокутника

Аксіоми для методів прямокутника читаються цілком очевидно. Розглянемо наприклад, аксіому [*Rgh_sw*] : робота методу, який задає ширину не впливає на довжину, що виймається з об'єкта. Аксіома [*Rgw_sw*] ще більш зрозуміла: ширина фігури, що виймається з об'єкта, повинна співпадати з шириною, яка була задана раніше.

```
   Figure
   scheme Square = extend Figure
with class
    axiom
      [Sgh_sw]
        all w: UReal, this: F:-
getHeight(setWidth(this, w))  = w,
      [Sgw_sw]
        all w: UReal, this: F:-
getWidth (setWidth(this, w))  = w,
      [Sgh_sh]
        all h: UReal, this: F:-
getHeight(setHeight(this, h)) = h,
      [Sgw_sh]
```

```
        all h: UReal, this: F:-
 getWidth (setHeight(this, h)) = h
   end
```

Listing 3: Вимоги до методів квадрата

Аксіоми для методів квадрата теж цілком зрозумілі: значення для розміру будь-якої сторони збігається зі значенням розміру сторони, яка задавалася раніше, незалежно від того чи задавалася ширина або довжина квадрата.

Спадкування буде виконане, як у другому прикладі з [15]. Таким чином, базовим класом вважається клас *Rectangle*, а похідним – *Square.* Наступні лістинги містять оригінальний код прикладу.

```
  #include <float.h>
  class Rectangle{
    public:
      Rectangle():       w(DBL_MIN),
h(DBL_MIN)  {}
      virtual  void  SetWidth(double
w)  {this->w=w;}
      virtual void SetHeight(double
h) {this->h=h;}
      double    GetHeight()    const
{return h;}
      double   GetWidth()      const
{return w;}
    private:
      double w;
      double h;
  };
```

Listing 4: Базовий клас прямокутник

```
  #include "Rectangle.h"
  class Square : public Rectangle {
    public:
      virtual  void  SetWidth(double
w);
      virtual void SetHeight(double
h);
  };
  void Square::SetWidth(double w){
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
  }
  void Square::SetHeight(double h){
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
  }
```

Listing 5: Похідний клас квадрат

```
  #include <stdio.h>
  #include <assert.h>
  #include "Square.h"
  void g(Rectangle& r) {
    r.SetWidth(5);
    r.SetHeight(4);
    assert((r.GetWidth()          *
r.GetHeight()) == 20.0);
    return;
  }
  int main() {
    fprintf(stderr, "Rectangle:\n");
    Rectangle r;
    r.SetWidth(1.0);
    r.SetHeight(3.0);
    g(r);
   fprintf(stderr, "Square:\n");
    Square s;
    s.SetWidth(5.0);
    fprintf(stderr, "before g:\n");
    g(s);
  }
```

Listing 6: Початковий приклад використання фігур

Компіляція, згідно з визначенням спадкування в компіляторі (клас *Square* є класом *Rectangle*), завершується побудовою додатка. Спроба його виконати призводить до передбачуваного результату [15]. Припинення роботи додатка у функції *g* на рядку з макросом *assert*:

```
  Rectangle:
  Square:
  Before g:
  Assertion failed: (r.GetWidth() *
r.GetHeight()) == 20.0, file a.cpp,
line 7
```

Далі, розглянемо яким чином вимоги специфікацій з лістингів (наприклад, 1) можна додати в код методів, що розробляються і яка від цього буде користь. Швидкість виконання коду поки що не цікавить.

Вимога використовувати лише позитивні дійсні числа з лістингу 1 легко перетворюється на передумови методів установки (англ. setter) таким чином:

```
  virtual void SetWidth ( double w)
{
     _Expects (w > 0.0);
```

У випадку методів читання (англ. getter) цю вимогу очевидно можна трактувати як перевірку інваріанту:

```
  double GetHeight () const {
```

```
      _Ensures ( w > 0 . 0 && h > 0
. 0 ) ;
       return h ;
  }
```

Аксіоми з лістингів 2 і 3 можна використовувати в коді наступним чином:

```
  virtual void SetWidth ( double w)
{
         _Expects (w > 0.0);
         double oldH = GetHeight ();
      // [ Rgh_sw ]: I need to save
h before update
         this - >w=w ;
     // setWidth (f , h ): update is
here
                     // [ Rgh_sw ]
           [ Rgw_sw ]
    _Ensures ( GetHeight () == oldH
&& GetWidth () == w );
   }
```

Аксіома [*Rgh_sw*] перевіряється першим порівнянням у макросі *_Ensures*: значення довжин після зміни ширини (*GetHeight()*) і до (oldH) має збігатися. Аксіома[*Rgw_sw*] перевіряється другим порівнянням. Змінна *w*, це вхідний параметр методу (до використання методу запису), порівнюється зі значенням, яке повертає метод *GetWifth()* (тобто після використання методу запису).

Таким чином, після переробки тексту обох класів намагаємося перебудувати та виконати додаток з лістингу 6. Результат виконання додатку не змінився:

```
  Rectangle:
  Square:
  Before g:
  Assertion failed: (r.GetWidth() *
r.GetHeight()) == 20.0, file a.cpp,
line 7
```

Тобто, додавання перевірки контактів в обидва класи поки не впливає на результати роботи додатку і не вдалося досягти більш раннього отримання відмови під час виконання. Версії додатку без використання бібліотеки GSL та з використанням GSL працюють однаково. Додавання контрактів до цих класів не допомогло знайти справжній дефект в коді прикладу.

Але, з погляду проєктування за контрактами, перевірка перед- і пост- умов у цьому конкретному прикладі виявляються не дуже коректними. Похідний клас повинен задовольняти вимогам і базового, і похідного класу одночасно. У специфікації до класу *Square* майбутнє наслідування під час кодування потрібно записати так:

```
  -- Figure
  -- scheme Square = extend Figure
with class
  Rectangle
  scheme Square = extend Rectangle
with class
```

Тобто, розширювати аксіомами квадратів потрібно не початкову схему *Figure*, а наступну – *Rectangle*. Тоді зрозуміло, що, наприклад, в макросі *_Ensures* похідного класу потрібно додати код для перевірки обох аксіом зі зв'язкою I:

```
      [Rgh_sw]
        all  w:  UReal,  this:  F:-
getHeight(setWidth(this,  w))     =
getHeight(this),
      [Sgh_sw]
        all  w:  UReal,  this:  F:-
getHeight(setWidth(this, w))  = w
```

Це означає, що результат роботи методу *getHeight* одночасно повинен дорівнювати і фактичному параметру *w*, і старому результату роботи метода *getHeight*. Тобто, в макросі має бути не дві умови зі зв'язкою &&, а три. Це має призвести до того, що перевірка контракту похідного класу зупинить роботу додатка до вже розглянутого вище моменту виконання. Переробимо похідний клас:

```
  #include "Rectangle.h"
  class Square : public Rectangle {
    public:
      virtual  void  SetWidth(double
w);
      virtual void SetHeight(double
h);
  };
  void Square::SetWidth(double w){
    _Expects(w > 0.0);
    double oldH = GetHeight();    //
[Rgh_sw]: I need to get h before
update
    Rectangle::SetWidth(w);
```

```
    Rectangle::SetHeight(w);
    // axioms: [Sgw_sw]          &&
[Sgh_sw]        &&   [Rgh_sw]
    _Ensures( GetWidth() == w &&
GetHeight() == w && GetHeight() ==
oldH);

//   Fault have to be here
  //   _Ensures( GetWidth() == w &&
GetHeight() == w );
  }
  void Square::SetHeight(double h){
    _Expects(h > 0.0);
    double oldW =  GetWidth();    //
[Rgw_sh]: I need to get w before
update
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
    // axioms:  [Sgh_sh]          &&
[Sgw_sh]       &&  [Rgw_sh]
    _Ensures( GetHeight() == h   &&
GetWidth() == h &&   GetWidth() ==
oldW);
  }
  Тепер результат роботи додатку
змінився:
  Rectangle:
  Square:
  [!] Violation of contract in
Expects:
  [!]          file/lineNo/function:
'.\Rectangle.h'/20/' \
      double              __thiscall
Rectangle::GetHeight(void) const'
```

У журналі роботи програми немає повідомлення *'Before g:'*. Це означає, що зупинка роботи відбулася на рядку *s.SetWidth(5.0);*, під час перевірки аксіоми [*Rgh_sw*]. Отриманий результат показує, що коректна перевірка умов контракту дозволяє виявляти дефект коду до застосування функції *g* і може бути виконана під час локального тестування. Для цього не потрібно використовувати код клієнтського додатку, який використовує об'єкт класу із контрактом. В даному випадку для модульного тестування треба вибрати клас *Square*.

Отриманий результат прямо суперечить думці автора прикладу [15] (клієнт – це код, який використовує/викликає методи класу):

"This leads us to a very important conclusion. A model, viewed in isolation, can not be meaningfully validated. The validity of a model can only be expressed in terms of its clients".

Це призводить до дуже важливого висновку. Модель, що розглядається ізольовано, не піддається змістовній перевірці. Коректність моделі може бути виражена лише у термінах її клієнта.

Нагадаємо, що ця відмова могла б бути отримана під час модульного тестування представлених класів, під час виконання найпростіших димових тестів.

Далі, спробуємо застосувати принцип підстановки за остаточною його редакцією [14], а не за редакцією 1988 року, як робить це автор прикладу:

Нехай *f(r)* – деяка властивість, що виконується (або твердження, що дійсне) на об'єктах r типу *T*. Тоді властивість *f(s)* має виконуватися на об'єктах типу *S*, якщо *S* є підтипом типу *T*.

Як властивість *f(r)* будемо використовувати таке твердження: для всіх об'єктів *r* класу *Rectangle* виконання функції *g* досягає оператора *return*.

Оскільки в першому, без контрактному варіанті класів на об'єкті *s* класу *Square* виконання функції *g* не досягало оператора *return*, то стає очевидним порушення принципу підстановки без додаткових апеляцій автора прикладу до зміненої поведінки функції *g* та до якихось осмислених припущень розробника цієї функції *g*. Розробник ПЗ повинен виконувати вимоги специфікації без власних припущень.

Висновки

У роботі отримано наступні результати:

- алгебраїчні вимоги до застосування функцій досить очевидно можуть кодуватися в макросах перевірки інваріантів, перед- та пост- умов;
- мова формальних специфікацій RSL досить легко поєднується з бібліотекою GSL;
- формальний підхід до розробки окремих модулів добре поєднується з V-подібною стратегією розробки всього

додатку;

- часткове застосування методів формальної розробки до деяких (і навіть до одного) модулів додатку може суттєво полегшити його розробку;
- формальний підхід до розробки двох окремих модулів дозволив одержати результат прямо протилежний автору прикладу, а саме: висновок про неможливість використання об'єктів класу спадкоємця було отримано під час перевірки контракту цього об'єкта до виклику функції *g*.

Література

1. Walk K., Lucas P. On the formal description of PL/I. *Annual Review in Automatic Programming*. 1969. Vol. 6. P. 105–182.
2. Bjørner D. (ed.). The Vienna Development Method: The Meta-Language. Berlin ; Heidelberg : Springer-Verlag, 1978. Vol. 61. (Lecture Notes in Computer Science).
3. Meyer B. Object-oriented Software Construction. 2nd ed. Santa Barbara : ISE Inc, 2000. 1284 p.
4. Haxthausen A.E. Lecture notes on the raise development method. Department of Information Technology, Technical University of Denmark. Lyngby, 1999. (Lecture notes, DTU IMM).
5. Haxthausen A.E. Алгебраїчне проєктування класу / пер. з англ. О.Г. Піскунов. 2010. 23 с. URL: https://www.researchgate.net/publication/334173954 (дата звернення: 03.04.2026). DOI: 10.13140/RG.2.2.32601.53600
6. Jacky J. The Way of Z: Practical Programming with formal Methods. New York : Cambridge University Press, 1997. DOI: 10.1109/MC.2010.22
7. Parnas D.L. Really rethinking 'formal methods'. Computer (IEEE Computer Society). 2010. Vol. 43. P. 28–34. DOI: 10.1007/s10270-023-01124-2
8. Gleirscher M., Marmsoler D., Woodcock J. A manifesto for applicable formal methods. Software and Systems Modeling. 2023. Vol. 22, no. 6. P. 1737–1749. DOI: 10.32689/maup.it.2023.5.7
9. Піскунов О.Г., Тупко Н.П., Петренко І.А. Алгебраїчне проєктування програмного забезпечення. Інформаційні технології та суспільство. 2024. № 5 (11). С. 50–59. DOI:10.1007/s10270-023-01124-2
10. Grassmann H.G. Die Ausdehnungslehre: vollständig und in strenger Form bearbeitet. Leipzig : Otto Wigand, 1862.
11. Піскунов О.Г. Класи та проєктування програмного забезпечення. 2026. 172 с. URL: https://www.researchgate.net/publication/395422671 (дата звернення: 03.04.2026). DOI: 10.32689/maup.it.2025.3.18
12. Піскунов О.Г., Тупко Н.П., Хребет В.Г. Арифметичні обчислення та небезпечність універсального поліморфізму. Інформаційні технології та суспільство. 2025. Вип. 3 (18). С. 128–135. DOI: 10.32689/maup.it.2025.3.18
13. Stroustrup B., Sutter H. C++ Core Guidelines. 2024. URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines (дата звернення: 03.04.2026). DOI: 10.1145/197320.197383
14. Liskov B., Wing J. A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems. 1994. Vol. 16, no. 6. P. 1811–1841.
15. Martin R. The Liskov Substitution Principle. C++ Report. 1996. URL: https://objectmentor.com/resources/articles/lsp.pdf (дата звернення: 03.04.2026).
16. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Upper Saddle River : Prentice Hall, 2017. 432 p. (Robert C. Martin Series).
17. Піскунов О.Г. Про відмінності між поняттями типу та класу. Вісник Київського Національного університету ім. Т.Г.Шевченка: комп'ютерні науки. 2015. № 3. С. 106–114.

Піскунов О.Г., Тупко Н.П.

ЧАСТКОВЕ ЗАСТОСУВАННЯ МЕТОДІВ ФОРМАЛЬНОЇ РОЗРОБКИ

У статті розглянуто проблему практичного застосування методів формальної розробки програмного забезпечення, зокрема їх часткового впровадження в окремі модулі програмних систем. Актуальність дослідження зумовлена складністю повномасштабного використання формальних методів у реальних проєктах через високий поріг входження та необхідність застосування спеціалізованих мов специфікацій. Метою роботи є обґрунтування та експериментальна перевірка можливості використання підходу програмування за контрактом із застосуванням бібліотеки GSL для раннього виявлення помилок у процесі розробки.

У роботі проаналізовано класичний приклад порушення принципу підстановки Лісков на основі відомого прикладу наслідування класів «прямокутник» і «квадрат». Формальні специфікації вимог до цих класів задано в алгебраїчній формі та реалізовано у вигляді перед-умов, пост-умов і інваріантів у програмному коді. Показано, що без урахування повного набору вимог базового і похідного класів перевірка контрактів не забезпечує раннього виявлення дефектів. Наголошено на важливості коректного та повного кодування аксіом базового та похідного класу при реалізації вибраною мовою програмування.

Експериментальні результати демонструють, що коректна реалізація контрактів дозволяє виявити помилки ще на етапі модульного тестування, до взаємодії з клієнтським кодом. Це спростовує твердження автора прикладу про неможливість перевірки коректності моделі без її використання клієнтом. Практична значущість роботи полягає у підтвердженні доцільності часткового застосування формальних методів, що сприяє підвищенню надійності програмного забезпечення без істотного впливу на продуктивність кінцевого продукту.

Ключові слова: формальні методи; програмування за контрактом; принцип підстановки Лісков; GSL; раннє виявлення помилок; модульне тестування; алгебраїчна специфікація.

Piskunov O.H., Tupko N.P.

THE PARTIAL APPLICATION OF FORMAL SOFTWARE DEVELOPMENT METHODS

The article examines the problem of the practical application of formal software development methods, in particular, their partial implementation in individual modules of software systems. The relevance of the study is due to the difficulty of the full-scale use of formal methods in real projects because of the high entry barrier and the need to use specialized specification languages. The aim of the work is to substantiate and experimentally verify the possibility of using the design-by-contract approach with the GSL library for early error detection during the development process.

The work analyzes a classic example of a violation of the Liskov substitution principle based on the well-known inheritance example of the "rectangle" and "square" classes. Formal specifications of the requirements for these classes are given in algebraic form and implemented as preconditions, postconditions, and invariants in the program code. It is shown that without considering the full set of requirements of the base and derived classes, contract checking does not ensure early defect detection. The importance of correct and complete encoding of the axioms of the base and derived classes when implementing them in the chosen programming language is emphasized.

Experimental results demonstrate that the correct implementation of contracts makes it possible to detect errors already at the unit testing stage, before interaction with client code. This refutes the original author's claim that it is impossible to verify the correctness of the model without its use by a client. The practical significance of the work lies in confirming the feasibility of the partial application of formal methods, which helps to increase software reliability without significantly affecting the performance of the final product.

Keywords: *formal methods; programming by contract; Liskov substitution principle; GSL; early error detection; unit testing; algebraic specification.*

Стаття подана до редакції: 11*/09/2026*
Стаття прийнята до опублікування:18/09/2026
Стаття опублікована: 10/11/2026

Стаття поширюється на умовах ліцензії CC BY 4.0