МІНІСТЕРСТВО НАУКИ ТА ОСВІТИ УКРАЇНИ

Київський авіаційний інститут

Факультет комп’ютерних наук та технологій

Кафедра прикладної математики

Курсова робота

Тема: «Загроза універсального поліморфізму. Аналіз порушень LSP у мові Eiffel з використанням Design by Contrac»

З дисципліни «Об’єктно-орієнтоване програмування»

Виконав студент групи

Б-122-23-1-ШІ:

Щербинський Дмитро

Прийняв:

Піскунов Олексій Германович

Київ-2025

Зміст

1. Вступ

Сучасна розробка програмного забезпечення значною мірою спирається на парадигму об’єктно-орієнтованого програмування (ООП), де механізми успадкування та поліморфізму відіграють ключову роль у побудові гнучких систем. Проте, некоректне застосування цих механізмів, зокрема універсального поліморфізму, може призводити до неочевидних архітектурних помилок, які ставлять під загрозу надійність програмного продукту. Однією з найфундаментальніших проблем у цьому контексті є порушення принципу підстановки Ліскoв (Liskov Substitution Principle – LSP), особливо при спробі перенести математичні відношення множин (наприклад, $Z \subset R$) безпосередньо в ієрархію класів.

Актуальність теми зумовлена тим, що в багатьох популярних мовах програмування (таких як C# або Java) порушення контракту базового класу класом-нащадком часто залишається непоміченим на етапі компіляції та виконання, призводячи до "тихого" спотворення даних. Це особливо критично для обчислювальних алгоритмів, де точність операцій є визначальною.

Об’єктом дослідження в даній роботі є поведінка числових ієрархій типів в умовах застосування методології проектування за контрактом (Design by Contract – DBC).

Предметом дослідження є виявлення та аналіз порушень LSP при реалізації симплекс-методу мовою Eiffel, де клас цілих чисел успадковується від класу дійсних чисел із перевизначенням арифметичних операцій.

Метою роботи є демонстрація переваг використання вбудованих контрактів (передумов, постумов та інваріантів) для детекції логічних помилок проектування, які в традиційних мовах призводять до некоректних обчислень без явних повідомлень про збій.

У роботі висувається гіпотеза, що строга контрактна система мови Eiffel дозволить виявити конфлікт між математичною абстракцією та її програмною реалізацією (зокрема, проблему округлення) на етапах компіляції або виконання, що є критично важливим для забезпечення коректності програмного забезпечення. Як тестовий полігон для перевірки гіпотези обрано алгоритм розв’язання задач лінійного програмування (симплекс-метод), оскільки він є чутливим до порушення числових інваріантів.

2. Постановка задачі

Загальне завдання

Необхідно розробити програмну систему мовою Eiffel для розв’язання задач лінійного програмування (ЗЛП) модифікованим симплекс-методом. Система повинна демонструвати поведінку поліморфних числових типів у контексті суворих контрактних зобов'язань.

Вимоги до математичної моделі

Алгоритм має працювати з канонічною формою задачі лінійного програмування, зведеною до Slack-форми (форми з вільними змінними).

Вхідні дані задаються у вигляді:

- Вектора коефіцієнтів цільової функції *c*.
- Матриці обмежень *A*.
- Вектора вільних членів *b*.

Математичні інваріанти, які система повинна контролювати під час виконання:

1. **Допустимість розв’язку (Feasibility):** Всі базисні змінні x_i повинні залишатися невід'ємними (b_i >= 0) на кожній ітерації.
2. **Коректність операції Pivot:** Значення опорного елемента дільника має бути строго більшим за нуль.
3. **Точність обчислень:** Результат операцій ділення та множення має відповідати очікуванням базового класу з заданою точністю epsilon.

Архітектурні вимоги та ієрархія класів

Для перевірки гіпотези про порушення принципу підстановки Лісков (LSP) необхідно реалізувати наступну ієрархію класів:

1. **Абстрактний тип NUMBER:** Визначає інтерфейс арифметичних операцій (add, subtract, multiply, divide).
2. **Базовий клас REAL_NUMBER:**
   - Імплементує операції з використанням дійсних чисел (плаваюча кома).
   - Містить *контракти* (постумови), що гарантують математичну точність операцій (наприклад, Result * other ~ Current).
3. **Похідний клас INTEGER_NUMBER:**
   - Успадковується від REAL_NUMBER.
   - Перевизначає арифметичні операції, додаючи примусове **округлення** до цілого числа.

- Специфічна вимога: не послаблювати постумови базового класу (не використовувати ensure then для скасування перевірок точності).

Вимоги до алгоритму та середовища виконання

Клас, відповідальний за симплекс метод (SIMPLEX_SOLVER) має бути реалізований як узагальнений (generic) клас, параметризований типом REAL_NUMBER. Це дозволить використовувати один і той самий алгоритмічний код для обох числових типів без змін (так як INTEGER_NUMBER наслідує REAL_NUMBER).

Необхідно реалізувати два сценарії виконання:

1. **Еталонний сценарій:** Запуск алгоритму з параметром REAL_NUMBER. Очікуваний результат: успішне знаходження оптимуму.
2. **Тестовий сценарій (LSP Violation):** Запуск алгоритму з параметром INTEGER_NUMBER. Очікуваний результат: аварійна зупинка програми через порушення контракту (Contract Violation Exception).

Критерії успішного виконання роботи

Робота вважається виконаною, якщо:

- Реалізовано механізм *Design by Contract* (передумови, постумови, інваріанти класу).
- Програмно зафіксовано момент, коли об'єкт класу INTEGER_NUMBER не може задовольнити постумови, визначені в класі REAL_NUMBER (наприклад, при діленні 7/2 ціле число поверне 3 або 4, що суперечить постумові дійсного числа, яка очікує 3.5 +- epsilon).
- Отримано трасування стеку (stack trace), що вказує на конкретний контракт, який було порушено під час виконання ітерації симплекс-методу.

3. Теоретична частина

4.1 Симплекс-метод

Концепція та геометрична основа симплекс-методу

Симплекс-метод є фундаментальним і донині найпоширенішим індустріальним алгоритмом розв'язування задач лінійного програмування. Його теоретична основа спирається на два ключові факти з опуклого аналізу:

1. Допустима область задачі лінійного програмування (система лінійних нерівностей та невід'ємність змінних) є опуклим багатогранником у n-вимірному просторі.
2. Лінійна цільова функція на опуклій множині досягає свого максимуму (або мінімум зводиться до максимуму зміною знаку) в одній з вершин цього багатогранника.

Отже, замість перебору всіх точок простору достатньо досліджувати лише вершини. Симплекс-метод реалізує систематичний перехід від однієї вершиною до сусідньої вершини, що покращує (або принаймні не погіршує) значення цільової функції. Такий перехід називається кроком симплекс-методу або пивот-операцією (pivot).

Стандартна форма задачі та введення допоміжних змінних

Задача лінійного програмування у стандартній формі записується так:

maximize $z = c_1x_1 + c_2x_2 + \dots + c_nx_n$

subject to

$A\,x \le b$

$x \ge 0$

де A – матриця розміром m×n, b – вектор правих частин ($b \ge 0$), x – вектор змінних.

Для переходу до канонічної форми (системи рівнянь) вводяться m допоміжних невід'ємних змінних $x_{n+1}, \dots, x_{n+m}$ (так звані slack-variables або змінні-залишки):

$a_{11}x_1 + \dots + a_{1n}x_n + x_{n+1} = b_1$

…

$a_{m1}x_1 + \dots + a_{mn}x_n + x_{n+m} = b_m$

$x_j \ge 0 \;\; (j = 1..n+m)$

Така форма називається slack-формою і є ключовою для роботи симплекс-методу з кількох причин:

- При $b_i \geq 0$ очевидне початкове базисне допустиме рішення: $x_1 = \ldots = x_n = 0$, $x_{n+i} = b_i$ (i = 1..m).
- Система рівнянь дозволяє чітко розділити змінні на базисні (B) та небазисні (N).
- Уся інформація про поточний базис представляється симплекс-таблицею, а зміна базису зводиться до елементарних операцій рядків (аналог операції Гаусса).

Базисні та небазисні змінні. Симплекс-таблиця

У будь-який момент алгоритм підтримує розбиття змінних на два неперетинні підмножини:

- B – набір індексів базисних змінних (|B| = m),
- N – набір індексів небазисних змінних (|N| = n).

Небазисні змінні фіксуються в нулі, а базисні виражаються через них за допомогою системи рівнянь. У матриці коефіцієнтів поточного базису B (m×m) є одинична матриця, що дозволяє легко обчислювати поточне рішення та приведені витрати (reduced costs).

Пивот-операція – серце симплекс-методу

Пивот складається з двох етапів:

1. Вибір entering variable (змінної, що входить у базис) Обирається небазисна змінна x_e з найвигіднішим (найменшим при максимізації) приведеним коефіцієнтом $\hat{c}_j$ у рядку цільової функції. Якщо всі $\hat{c}_j \geq 0$ – досягнуто оптимум.
2. Вибір leaving variable (змінної, що виходить з базису) – тест відношення (ratio test) Для кожного рядка i, де коефіцієнт при x_e додатний ($a_{ie} > 0$), обчислюється $ratio_i = b_i / a_{ie}$ Обирається рядок l з мінімальним $ratio_i$ (правило Бленда або найменшого індексу при рівності). Змінна, що стоїть у базисі в цьому рядку, стає небазисною.

Після вибору (e, l) виконується елементарне перетворення рядків таблиці (пивотування за елементом A[l][e]), щоб новий базис знову містив одиничну підматрицю.

Критична роль точності обчислень у тесті відношення

Найвразливішим місцем симплекс-методу з погляду чисельної стабільності є саме обчислення $ratio_i = b_i / a_{ie}$ та подальше оновлення таблиці.

Розглянемо простий приклад: $b_l = 7, \quad a_{le} = 2$

Точне ділення дає ratio = 7/2 = 3.5 Якщо після пивоту нова базисна змінна отримає значення 3.5, то всі обмеження залишаться невід’ємними.

Тепер уявімо, що ми працюємо з цілочисельною арифметикою з відсіканням (як у багатьох мовах при цілому діленні): *7 // 2 = 3*

Тоді нова базисна змінна отримує значення 3, а залишок 1 «зникає». При подальшому оновленні інших елементів рядка l та рядка цілі виникають похибки, які накопичуються. У гіршому випадку:

- обирається неправильна leaving variable,
- нове базисне рішення стає недопустимим ($b_i < 0$),
- з’являються цикли (stalling або cycling),
- алгоритм втрачає теоретичну скінченність і коректність.

Тому для класичного симплекс-методу обов’язково потрібні типи даних, що забезпечують точне (або з контрольованою похибкою) представлення раціональних чисел під час всіх операцій, особливо ділення.

4.2 Поліморфізм як фундаментальна парадигма ООП

Парадигми об'єктно-орієнтованого програмування

Об'єктно-орієнтоване програмування (ООП) – це методологія розробки програмного забезпечення, що базується на представленні програми як сукупності об'єктів, кожен з яких є екземпляром певного класу. Класи утворюють ієрархію наслідування.

Основними парадигмами ООП є:

1. **Абстракція:** Виділення значущих характеристик об'єкта та ігнорування незначних деталей для спрощення моделі.
2. **Інкапсуляція:** Механізм, що об'єднує дані та методи, які маніпулюють цими даними, і захищає їх від зовнішнього втручання або неправильного використання.
3. **Наслідування (Inheritance):** Механізм, що дозволяє створювати нові класи на основі існуючих. Похідний клас переймає (наслідує) властивості та поведінку базового класу, встановлюючи відношення "є" (is-a).
4. **Поліморфізм:** Здатність об'єктів з різною внутрішньою реалізацією (різних класів) реагувати на однакові повідомлення (виклики методів).

Саме взаємодія наслідування та поліморфізму забезпечує ключову перевагу ООП – **повторне використання коду** та **розширюваність** (extendibility), дозволяючи програмісту використовувати методи бібліотечних класів для нових типів даних, розроблених пізніше.

Поліморфізм: сутність та класифікація

У широкому сенсі поліморфізм в інформатиці – це властивість програмних сутностей (змінних, функцій) мати багато форм. В контексті типізації це означає здатність коду обробляти дані різних типів.

Розрізняють такі основні види поліморфізму:

1. **Ad-hoc поліморфізм (Спеціальний):**
   - *Перевантаження (Overloading):* Існування кількох функцій з однаковим іменем, але різними сигнатурами (типами або кількістю параметрів). Компілятор вибирає потрібну функцію на етапі компіляції (раннє зв'язування).
   - *Приведення типів (Coercion):* Автоматичне або явне перетворення одного типу в інший для виконання операції.
2. **Універсальний поліморфізм (Universal Polymorphism):**
   - Код, написаний універсально, може працювати з нескінченною кількістю типів, що мають спільну структуру.
   - Включає в себя *параметричний поліморфізм* (Generics) та *поліморфізм включення* (Subtyping).

Механізм, що забезпечує поліморфізм включення в часі виконання, називається **пізнім зв'язуванням** (late binding). Це означає, що адреса викликаного методу визначається не під час компіляції, а безпосередньо в момент виклику, базуючись на динамічному типі об'єкта.

Універсальний поліморфізм та його небезпека

Універсальний поліморфізм є найпотужнішим інструментом ООП, що дозволяє писати узагальнені алгоритми. Наприклад, функція, що приймає аргумент типу REAL_NUMBER, завдяки поліморфізму включення може приймати будь-який об'єкт похідного класу (наприклад, INTEGER_NUMBER).

Однак, універсальний поліморфізм несе в собі приховані загрози, пов'язані з типізацією функцій вищого порядку та заміщенням методів. Ці загрози описуються через поняття **коваріантності** та **контраваріантності**:

1. Контраваріантність за областю визначення (аргументами):

   Якщо функція *f* очікує аргумент типу *T*, то безпечно передати їй функцію, яка вміє обробляти ширший тип (супертип *S*, де $T \subset S$). Звуження області визначення у похідному класі (вимога більш специфічного типу) є небезпечним і може призвести до помилок під час виконання.

Приклад: Якщо метод базового класу приймає будь-яке REAL, то метод спадкоємця не може вимагати лише додатні REAL.

2. Коваріантність за множиною значень (результатом):

   Якщо функція повинна повернути результат типу T, то безпечно повернути результат вужчого типу (підтип S, де $S \subset T$). Розширення множини значень у похідному класі є порушенням контракту.

Небезпека універсального поліморфізму полягає в тому, що синтаксично коректне наслідування (наприклад, INTEGER від REAL) може порушувати семантичні правила підтипізації. Якщо похідний клас змінює поведінку так, що вона виходить за межі очікувань клієнта базового класу (порушує принцип підстановки Лісков), універсальний поліморфізм перетворюється на джерело важковловлюваних помилок.

У мові Eiffel ця проблема вирішується шляхом строгого контролю коваріантності аргументів та механізму *Design by Contract*, який забороняє посилення передумов та послаблення постумов у спадкоємцях, гарантуючи безпеку поліморфізму.

4.3 Принцип підстановки Лісков (LSP)

Формальне визначення та сутність

Принцип підстановки Лісков (Liskov Substitution Principle – LSP) є третім принципом у наборі S.O.L.I.D. і визначає фундаментальний критерій коректності використання механізму наслідування в об'єктно-орієнтованому проектуванні. Він був сформульований Барбарою Лісков у 1987 році під час конференції OOPSLA.

Формальне визначення звучить так:

> *«Нехай q(x) є властивістю, що доводиться для об'єктів x типу T. Тоді q(y) повинно бути істинним для об'єктів y типу S, де S є підтипом T».*

У простішому формулюванні для програмістів це означає, що **функції, які використовують посилання на базові класи, повинні мати можливість використовувати об'єкти похідних класів, не знаючи про це і не порушуючи коректність роботи програми**.

Порушення цього принципу призводить до крихкої архітектури, де додавання нового класу-нащадка ламає існуючий код, який покладався на поведінку базового класу.

Коваріантність та контраваріантність у контексті LSP

Принцип підстановки тісно пов'язаний з поняттями варіативності типів при перевизначенні методів:

- **Контраваріантність аргументів:** Для дотримання LSP, типи аргументів методу в нащадку повинні бути *надтипами* (або тими ж самими) по відношенню до аргументів базового методу. Це відповідає правилу послаблення передумов.

  > *Зауваження:* Мова Eiffel дозволяє *коваріантне* перевизначення аргументів (звуження типу), що зручно для моделювання реального світу, але потенційно небезпечно (проблема CAT-call), тому основний захист покладається на контракти.

- **Коваріантність результату:** Тип значення, що повертається методом у нащадку, повинен бути *підтипом* типу результату базового методу. Це відповідає правилу посилення постумов.

Проблема "Кола та Еліпса" в числових типах

Класичний приклад порушення LSP – спроба змоделювати математичні множини через наслідування. Хоча математично цілі числа є підмножиною дійсних ($Z \subset R$), в об'єктно-орієнтованому програмуванні клас INTEGER не є коректним підтипом класу REAL, якщо REAL є мутабельним або має операції ділення.

У контексті даної роботи порушення LSP проявляється так:

1. Базовий клас REAL має контракт для ділення: result * divisor ≈ operand (з певною точністю epsilon).
2. Клас INTEGER, успадковуючись від REAL, перевизначає ділення як цілочисельне (з відкиданням дробової частини або округленням).
3. **Результат:** 7/2 для REAL дає 3.5. Для INTEGER це дає 3.
4. **Порушення:** 3 * 2 = 6, що не рівно 7. Постумова базового класу не виконується.

Алгоритм симплекс-методу, написаний для роботи з абстракцією NUMBER (очікуючи поведінку поля дійсних чисел), при підстановці об'єкта INTEGER отримає невірні дані на кроці *Pivot* (розрахунок коефіцієнтів), що призведе до розбіжності розв'язку або зациклення. Використання механізму DBC дозволяє виявити цю архітектурну помилку у вигляді виключення Postcondition Violation під час виконання.

4.4 Розробка за контрактом (DbC)

Сутність методології

Розробка за контрактом (Design by Contract, DbC) – це методологія проектування програмного забезпечення, запропонована Бертраном Мейєром, яка розглядає взаємодію між програмними компонентами як формальну угоду (контракт) з чітко визначеними правами та обов'язками сторін.

В основі DbC лежить метафора ділового контракту між **Клієнтом** (модуль, що викликає метод) та **Постачальником** (модуль, чий метод викликається):

- Клієнт зобов'язаний задовольнити певні вхідні умови перед викликом методу.
- Якщо умови виконані, Постачальник зобов'язується виконати роботу та повернути стан, що відповідає вихідним вимогам.

Ця методологія дозволяє інтегрувати специфікації (вимоги до ПЗ) безпосередньо у програмний код, роблячи їх частиною виконуваної програми, а не лише документації.

Складові елементи контракту

Формально контракт класу складається з трьох ключових елементів, які базуються на поняттях абстрактних типів даних (АТД) та аксіоматичній семантиці:

1. Передумови (Preconditions):

   Це вимоги, які повинні бути істинними перед початком виконання методу. Вони визначають область допустимих вхідних даних.

   Відповідальність: Клієнт. Якщо передумова порушена, це означає помилку в коді клієнта (він викликав метод некоректно).

2. Постумови (Postconditions):

   Це гарантії, які метод надає після свого завершення (за умови, що передумови були виконані). Вони описують, як змінився стан об'єкта та який результат повернуто.

   Відповідальність: Постачальник. Якщо постумова порушена, це означає помилку в самому методі (він не виконав обіцянку).

3. Інваріанти класу (Class Invariants):

Це глобальні умови цілісності, які повинні залишатися істинними для об'єкта протягом усього його життєвого циклу. Інваріант має бути істинним після завершення конструктора та після виконання будь-якого публічного методу класу.

Приклад для ЗЛП: Всі вільні члени у симплекс-таблиці повинні бути невід'ємними (b_i >= 0).

Контракти та спадкування (DbC і LSP)

Особлива роль DbC проявляється при побудові ієрархії класів. Для дотримання принципу підстановки Лісков (LSP), перевизначення методів у класах-нащадках повинно підкорятися строгим правилам роботи з контрактами:

- **Правило для передумов:** Похідний клас може вимагати *менше*, але не більше. Передумова може бути лише **послаблена**. Це реалізується через логічну диз'юнкцію (OR) батьківської та нової передумови.
- **Правило для постумов:** Похідний клас повинен гарантувати *стільки ж* або *більше*. Постумова може бути лише **посилена**. Це реалізується через логічну кон'юнкцію (AND) батьківської та нової постумови.

Спроба "обійти" ці правила (наприклад, додати нову обов'язкову вимогу в INTEGER_NUMBER, якої не було в REAL_NUMBER) призведе до порушення поліморфізму, оскільки клієнт, що працює через інтерфейс базового класу, не знатиме про нові обмеження.

Переваги DbC для надійності систем

Використання DbC, особливо в мовах з нативною підтримкою (як Eiffel), надає суттєві переваги порівняно з традиційним захисним програмуванням (Defensive Programming):

1. **Автоматична детекція помилок:** Порушення контракту викликає виключення під час виконання, що чітко вказує на місце та причину логічної помилки, а не маскує її під "некоректні дані".
2. **Документування:** Контракти слугують "живою" документацією коду, яка завжди є актуальною, оскільки перевіряється компілятором/середовищем виконання.
3. **Чіткий розподіл відповідальності:** DbC усуває необхідність у надлишкових перевірках аргументів всередині методу, якщо вони вже описані в передумовах, що спрощує код та підвищує його читабельність.

У контексті даної курсової роботи саме механізм DbC дозволяє виявити архітектурну невідповідність між класами INTEGER_NUMBER та

REAL_NUMBER, перетворюючи неявну математичну помилку округлення на явну помилку виконання програми (Contract Violation).

4.5 Мова програмування Eiffel

Загальна характеристика та філософія Eiffel – це об'єктно-орієнтована мова програмування зі строгою статичною типізацією, розроблена Бертраном Мейєром. Вона була створена не просто як інструмент кодування, а як втілення методології конструювання надійного програмного забезпечення. Ключовою особливістю мови є те, що механізм *Design by Contract* (DbC) інтегрований у її синтаксис на фундаментальному рівні, а не доданий через зовнішні бібліотеки чи анотації, як у більшості інших мов (C#, Java, C++).

Для даної роботи Eiffel було обрано як інструментальний засіб через його здатність виявляти архітектурні помилки на етапах компіляції та виконання, які в інших середовищах залишаються непоміченими.

Інтегрована підтримка контрактів Синтаксис Eiffel зобов'язує програміста явно визначати семантику класів та методів. Основні конструкції включають:

- **require**: Блок передумов перед тілом методу (do). Якщо умова не виконується, виникає виключення на стороні клієнта (викликаючого коду).
- **ensure**: Блок постумов після тіла методу. Гарантує правильність результату. В контексті роботи це дозволяє перевірити точність обчислень (наприклад, (Result - expected_value).abs <= epsilon).
- **invariant**: Блок інваріантів класу, який перевіряється після створення об'єкта та після кожного публічного виклику.
- **check**: Інструкція для перевірки тверджень у довільному місці коду (аналог assert в інших мовах).
- **loop ... invariant ... variant**: Спеціальні конструкції для доведення коректності циклів (інваріант циклу) та їх завершуваності (варіант циклу).

Механізми спадкування та адаптації контрактів Eiffel надає унікальні можливості для керування контрактами при спадкуванні, що є критично важливим для дотримання принципу підстановки Лісков (LSP):

1. **Послаблення передумов (require else):** При перевизначенні методу в нащадку, Eiffel дозволяє лише *послабити* вимоги до вхідних даних. Нова передумова автоматично об'єднується з батьківською логічним оператором OR.
    - *Синтаксис:* require else new_condition.
    - *Логіка:* parent_precondition OR new_condition.
2. **Посилення постумов (ensure then):** Нащадок зобов'язаний гарантувати все, що гарантував предок, і може додати власні гарантії. Нова постумова об'єднується з батьківською оператором AND.

- *Синтаксис:* ensure then new_condition.
- *Логіка:* parent_postcondition AND new_condition.

Саме конструкція ensure then унеможливлює "тихе" порушення семантики базового класу REAL_NUMBER класом INTEGER_NUMBER: якщо ціле число повертає округлений результат, воно неминуче порушить точну постумову дійсного числа, і система згенерує виключення.

Узагальнене програмування (Generics) Eiffel підтримує параметричний поліморфізм через generic-класи. На відміну від C# (де використовується where T : constraints), Eiffel використовує синтаксис [T -> CONSTRAINING_TYPE]. Це дозволяє реалізувати алгоритм симплекс-методу один раз у класі SIMPLEX_SOLVER[T -> NUMBER], де NUMBER – це абстрактний предок для REAL та INTEGER. Компілятор гарантує, що тип T матиме всі методи, визначені в класі NUMBER (наприклад, plus, minus, divide), а середовище виконання забезпечить перевірку контрактів саме для конкретного типу, з яким запущено алгоритм.

Коваріантність типів Важливою особливістю типізації Eiffel є підтримка коваріантності аргументів методів. Це означає, що якщо метод у батьківському класі приймає аргумент типу A, то в нащадку можна перевизначити цей метод так, щоб він приймав аргумент типу B (де B – нащадок A).

- *Перевага:* Це дозволяє природно моделювати предметні області (наприклад, метод add у класі INTEGER приймає саме INTEGER, а не абстрактний NUMBER).
- *Ризик:* Це створює проблему "CAT-call" (виклик методу на об'єкті невірного типу через поліморфне посилання). Eiffel вирішує цю проблему через строгі перевірки типів та механізм select при множинному спадкуванні, але в контексті даної роботи основний захист покладається на контракти.

4. Практична частина

4.1 Архітектура програмної системи

Для реалізації експерименту було розроблено програмну систему **SimplexEiffel**, архітектура якої спроектована за принципом слабкої зв'язаності (low coupling) та чіткого розділення відповідальності. Система складається з трьох логічних шарів (Layers):

1. Kernel – Числова абстракція:

   Цей рівень визначає базові математичні примітиви. Саме тут закладено механізм перевірки гіпотези про порушення LSP.

   - Клас NUMBER: Абстрактний предок, що задає інтерфейс арифметичних операцій.
   - Клас REAL_NUMBER: Реалізація чисел з плаваючою комою (IEEE 754). Цей клас виступає еталоном ("контрактним ідеалом"), оскільки містить суворі постумови для перевірки точності операцій.
   - Клас INTEGER_NUMBER: Клас-нащадок, який формально наслідує REAL_NUMBER, але перевизначає арифметичні операції з примусовим округленням. Це створює "бомбу уповільненої дії" для поліморфного коду, який очікує поведінку дійсних чисел.
2. Simplex – Алгоритмічний шар (логіка розв'язання):

   Цей рівень реалізує модифікований симплекс-метод. Ключовою особливістю є використання узагальненого програмування (Generics).

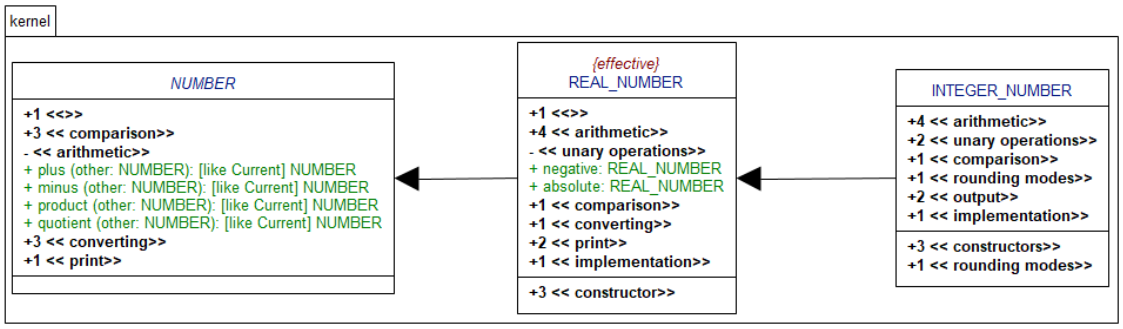
   - SIMPLEX_CONTEXT [T]: Контейнер даних (State Object), що зберігає матрицю обмежень A, вектори b та c. Він параметризований типом T, що дозволяє динамічно змінювати тип чисел без зміни коду структур даних.
   - SIMPLEX_SOLVER [T]: Головний клас-контролер. Він реалізує цикл алгоритму, пошук змінних для входу/виходу з базису. Важливо, що клас обмежений параметром [T -> REAL_NUMBER], тобто він "вірить", що працює з дійсними числами, навіть якщо йому передати INTEGER_NUMBER.
   - SIMPLEX_PIVOT [T]: Клас, що інкапсулює одну ітерацію перерахунку симплекс-таблиці (Pivot Operation).
3. Application – Прикладний шар (інтерфейс):

   Забезпечує взаємодію з користувачем та введення/виведення.

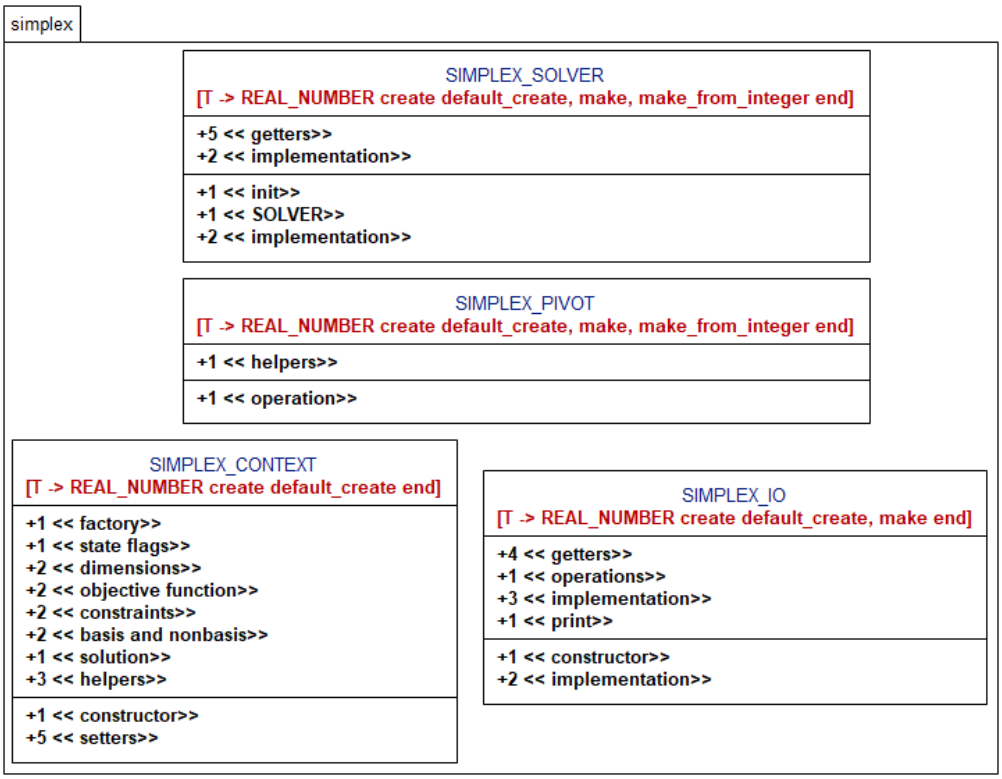
   - SIMPLEXAPP: Точка входу в програму. Залежно від аргументів командного рядка (-i або без нього), ініціалізує розв'язувач з типом INTEGER_NUMBER або REAL_NUMBER.

- SIMPLEX_IO: Парсер вхідних даних зі стандартного потоку введення.

Така архітектура дозволяє ізолювати "помилкову" поведінку в одному класі (INTEGER_NUMBER) та спостерігати, як вона руйнує роботу коректного алгоритму (SIMPLEX_SOLVER) через механізм поліморфізму.



Діаграма класів системи SimplexEiffel, кластеру kernel. Показано відношення наслідування, де INTEGER_NUMBER є підтипом REAL_NUMBER, що уможливлює поліморфну підстановку.



Діаграма класів системи SimplexEiffel, кластеру simplex. Показано компоненти логіки, пов’язану з самим алгоритмом симплекс методу.

4.2 Реалізація інваріантів та контроль стану

Для забезпечення цілісності даних під час виконання складних матричних перетворень, у класі SIMPLEX_CONTEXT реалізовано механізм **інваріантів класу (Class Invariants)**. Це глобальні твердження, які система перевіряє автоматично після кожного публічного виклику методу.

У розробленій системі визначено наступні критичні інваріанти:

1. Структурна цілісність:

   Розміри векторів та матриць повинні відповідати кількості змінних (n) та обмежень (m).

   Фрагмент коду

   ```
   x_size_matches_total: x.count = num_variables + num_constraints
   basis_and_nonbasis_disjoint: (across B as bi all not N.has (bi.item) end)
   ```

   Цей інваріант гарантує, що жодна змінна не може одночасно бути базисною і небазисною, що є фундаментом коректності симплекс-методу.

2. Математична допустимість (Feasibility):

   Згідно з теорією лінійного програмування, вільні члени (b) у канонічній формі завжди мають бути невід'ємними.

   Фрагмент коду

   ```
   all_b_non_negative:
      use_invariants implies
      (across B as bi all
         (attached b_values[bi.item] as bv and then bv.value >= -0.0001)
      end)
   ```

   Порушення цього інваріанту є індикатором того, що алгоритм "вилетів" з допустимої області рішень, що часто трапляється при помилках округлення.

В мові Eiffel ці перевірки вмикаються/вимикаються динамічно. У класі SIMPLEX_PIVOT перед початком маніпуляцій інваріанти тимчасово вимикаються (ctx.disable_invariants), оскільки проміжні стани таблиці можуть бути некоректними, і вмикаються знову (ctx.enable_invariants) лише після завершення перерахунку, що гарантує атомарність переходу між станами.

4.3 Застосування методології Design by Contract (DbC)

Для демонстрації порушення принципу підстановки Лісков (LSP) було використано три основні компоненти DbC: передумови, постумови та успадкування контрактів.

А. Передумови (Preconditions)

Передумови захищають методи від некоректних вхідних даних. У класі SIMPLEX_PIVOT метод pivot вимагає, щоб змінна, що виходить, була базисною, а та, що входить – небазисною:

Фрагмент коду

```
require
   valid_leaving: ctx.is_basic (leaving_var)
   valid_entering: ctx.is_nonbasic (entering_var)
```

Це класичний приклад захисного програмування, реалізованого через контракти.

Б. Постумови (Postconditions) та їх порушення

Постумови визначають очікуваний результат. Саме тут відбувається конфлікт між REAL та INTEGER.

У базовому класі REAL_NUMBER операція ділення гарантує точність:

Фрагмент коду

```
quotient alias "/" (other: NUMBER): REAL_NUMBER
   ensure then
      exact_division: (Result.value - (value / other.value)).abs < 0.000001
```

Цей контракт стверджує: "Результат ділення, помножений на дільник, повинен давати ділене (з похибкою epsilon)".

Однак, у класі INTEGER_NUMBER ділення реалізовано через подвійне округлення:

Фрагмент коду

```
quotient alias "/" (other: NUMBER): INTEGER_NUMBER
   do
      create Result.make(round_value(value / round_value(other.value)))
   end
```

Успадковуючись від REAL_NUMBER, клас INTEGER_NUMBER автоматично успадковує і його постумову exact_division. Оскільки цілочисельне ділення

математично не може задовольнити постумову дійсного числа, при виконанні програми виникає виключення **Postcondition Violation**.

B. Аналіз конфлікту в контексті LSP

У рамках методології Eiffel, спадкоємець може лише посилювати постумови (ensure then), але не скасовувати їх. Клас SIMPLEX_SOLVER покладається на контракт базового класу REAL_NUMBER. Коли ми підставляємо об'єкт INTEGER_NUMBER (LSP substitution), алгоритм продовжує працювати, вважаючи, що числа точні.

Коли виконується операція Pivot:

1. Обчислюється коефіцієнт (factor).
2. Якщо використовується INTEGER_NUMBER, коефіцієнт округлюється.
3. Після завершення операції Eiffel Runtime System перевіряє успадковану постумову.
4. Система детектує невідповідність і зупиняє виконання.

Таким чином, механізм DbC дозволяє перетворити семантичну помилку (неправильне використання спадкування для моделювання числових множин) на явну помилку часу виконання, підтверджуючи гіпотезу роботи.

4.4 Тестування

Метою етапу тестування була верифікація коректності роботи симплекс-методу в еталонному режимі (REAL_NUMBER) та експериментальне підтвердження гіпотези про виявлення порушень LSP у режимі INTEGER_NUMBER за допомогою контрактів.

Тестування проводилося на наборі вхідних даних, що покривають різні сценарії: цілочисельні розв'язки, дробові розв'язки, недопустимі умови та вироджені випадки.

Зведена таблиця результатів

Нижче наведено порівняльну характеристику виконання алгоритму для двох типів даних.

Тест	**Опис задачі**	**Результат (REAL)**	**Результат (INTEGER)**	**Висновок**
Problem 1	Стандартна ЗЛП (дробові проміжні значення)	**Успіх** (Z=10)	**Аварійна зупинка**	Детекція порушення LSP (Postcondition)
Problem 2	"Integer Safe" (цілі числа на всіх етапах)	**Успіх** (Z=34)	**Успіх** (Z=34)	Випадковий успіх підстановки
Problem 3	Стандартна ЗЛП	**Успіх** (Z=32.57..)	**Аварійна зупинка**	Детекція порушення LSP (Postcondition)
Problem 5	Задача з зацикленням	**Зупинка** після досягнення max_iter (Z=0)	**Аварійна зупинка**	Помилка ініціалізації (Pre/Postcondition)
Problem 6	Недопустимий план (b_i < 0)	**Аварійна зупинка**	**Аварійна зупинка**	Спрацювання інваріанту класу
Problem 7	Багатовимірна задача	**Успіх** $Z=24)	**Аварійна зупинка**	Детекція порушення LSP

Аналіз сценаріїв виконання

Сценарій 1: Успішна підстановка (Problem 2)

У задачі №2 всі коефіцієнти підібрані так, що під час виконання симплекс-методу (ділення на опорний елемент) результати залишаються цілими числами.

- **Результат:** Обидві версії програми повернули Z=34.
- **Аналіз:** Це демонструє, що INTEGER_NUMBER може технічно замінити REAL_NUMBER, якщо дані не призводять до втрати точності. Проте це "крихкий" успіх, який залежить від конкретних даних, а не від коректності архітектури.

Сценарій 2: Детекція порушення LSP (Problem 1, 3, 7)

Це основний сценарій дослідження. У задачі №1 при спробі виконати Pivot-операцію виникає необхідність ділення, де результат є дробовим.

- **Результат REAL:** Обчислено оптимум 10.
- **Результат INTEGER:** Програма аварійно завершилася з наступним трасуванням стеку:

```
simplexeiffel: system execution failed.
Following is the set of recorded exceptions:

****************************** Thread exception ****************************
In thread           Root thread           0x0 (thread id)
****************************************************************************
-------------------------------------------------------------------------------
Class / Object      Routine               Nature of exception          Effect
-------------------------------------------------------------------------------
INTEGER_NUMBER      quotient @5           correct_division:
<000001BAC6A62418>                        Postcondition violated.      Fail
-------------------------------------------------------------------------------
INTEGER_NUMBER      quotient @5
<000001BAC6A62418>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEX_PIVOT       pivot @13
<000001BAC6A60938>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEX_SOLVER      solve @22
<000001BAC6A60908>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEXAPP          run_integer_solver @7
<000001BAC6A5D5A8>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEXAPP          make @8
<000001BAC6A5D5A8>                        Routine failure.             Rescue
-------------------------------------------------------------------------------
SIMPLEXAPP          root's creation
<000001BAC6A5D5A8>                        Routine failure.             Exit
-------------------------------------------------------------------------------
```

- **Аналіз:** Виключення Postcondition violated у методі quotient (ділення) підтверджує гіпотезу. Клас INTEGER_NUMBER спробував повернути округлене значення, яке не задовольнило строгу постумову базового класу REAL_NUMBER (яка вимагає точності ϵ). Механізм DbC не дозволив програмі продовжити виконання з "спотвореними" даними.

Сценарій 3: Робота інваріантів (Problem 6)

Вхідний файл містить обмеження з від'ємним вільним членом (b = -5), що суперечить визначенню канонічної форми для запуску симплекс-методу.

- **Результат:**

```
simplexeiffel: system execution failed.
Following is the set of recorded exceptions:

****************************** Thread exception ***************************
In thread           Root thread           0x0 (thread id)
***************************************************************************
-------------------------------------------------------------------------------
Class / Object      Routine               Nature of exception          Effect
-------------------------------------------------------------------------------
SIMPLEX_CONTEXT     _invariant @3         all_b_non_negative:
<00000195E5F5A608>                        Class invariant violated.    Fail
-------------------------------------------------------------------------------
SIMPLEX_CONTEXT     _invariant
<00000195E5F5A608>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEX_CONTEXT     enable_invariants @2
<00000195E5F5A608>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEX_IO          read_from_stdin @41
<00000195E5F5ADD8>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEXAPP          run_real_solver @5
<00000195E5F5A5A8>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEXAPP          make @9
<00000195E5F5A5A8>                        Routine failure.             Rescue
-------------------------------------------------------------------------------
SIMPLEXAPP          root's creation
<00000195E5F5A5A8>                        Routine failure.             Exit
-------------------------------------------------------------------------------
```

- **Аналіз:** Система перевірки інваріантів (all_b_non_negative) спрацювала ще на етапі ініціалізації контексту (SIMPLEX_IO.read_from_stdin), до запуску алгоритму. Це доводить ефективність DbC для валідації вхідних даних незалежно від типу чисел.

Сценарій 4: Порушення контракту при створенні (Problem 5)

Вхідні дані містять дробові числа (наприклад, 0.5).

- **Результат INTEGER**

```
simplexeiffel: system execution failed.
Following is the set of recorded exceptions:

****************************** Thread exception ***************************
In thread           Root thread           0x0 (thread id)
***************************************************************************
-------------------------------------------------------------------------------
Class / Object      Routine               Nature of exception          Effect
-------------------------------------------------------------------------------
INTEGER_NUMBER      make @2               value_set:
<000001A78B4C7088>                        Postcondition violated.      Fail
-------------------------------------------------------------------------------
INTEGER_NUMBER      make @2
<000001A78B4C7088>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEX_IO          parse_value @3
<000001A78B4C5948>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEX_IO          read_from_stdin @29
<000001A78B4C5948>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEXAPP          run_integer_solver @5
<000001A78B4C45A8>                        Routine failure.             Fail
-------------------------------------------------------------------------------
SIMPLEXAPP          make @8
<000001A78B4C45A8>                        Routine failure.             Rescue
-------------------------------------------------------------------------------
SIMPLEXAPP          root's creation
<000001A78B4C45A8>                        Routine failure.             Exit
-------------------------------------------------------------------------------
```

- **Аналіз:** Помилка виникла в конструкторі. Постумова make вимагає value_set: value = v. Оскільки INTEGER_NUMBER округлює 0.5 до 1, умова 1 = 0.5 є хибною. Це ілюструє, що підтип не може коректно ініціалізуватися даними, які є валідними для базового типу, але виходять за межі домену підтипу.

Висновки з тестування

Проведені тести підтвердили працездатність системи та теоретичні припущення:

1. У середовищі Eiffel порушення математичної семантики при наслідуванні (INTEGER замість REAL) призводить до **явного виключення** (Exception) під час виконання.
2. Традиційні мови без вбудованого DbC у Сценарії 2 просто продовжили б обчислення з округленими значеннями, видавши невірний результат (наприклад, зациклення алгоритму або субоптимальне рішення) без жодних повідомлень про помилку.
3. Система контрактів успішно локалізувала помилку саме в місці її виникнення (метод ділення або конструктор), а не в глибині алгоритму симплекс-методу, що значно спрощує налагодження.

5. Висновок

У даній курсовій роботі було проведено комплексне дослідження проблеми безпеки універсального поліморфізму в об’єктно-орієнтованому програмуванні на прикладі мови Eiffel. Головним фокусом роботи став аналіз порушень принципу підстановки Лісков (LSP) при моделюванні числових ієрархій та роль методології Design by Contract (DbC) у виявленні таких архітектурних помилок.

У ході виконання роботи було отримано наступні результати:

1. **Теоретичне обґрунтування проблеми.** Проаналізовано природу конфлікту між математичною теорією множин та об’єктною ієрархією класів. Встановлено, що хоча математично цілі числа є підмножиною дійсних ($Z \subset R$), в ООП наслідування класу INTEGER від класу REAL є некоректним, якщо базовий клас передбачає операції ділення або вимагає певної точності. Таке наслідування порушує LSP, оскільки похідний клас змінює поведінку (округлення), що суперечить постумовам базового класу.
2. **Програмна реалізація.** Розроблено систему «SimplexEiffel» для розв’язання задач лінійного програмування симплекс-методом. Архітектура системи, побудована на основі узагальненого програмування (Generics), дозволила провести "чистий" експеримент із підстановкою різних числових типів у єдиний алгоритмічний модуль.
3. **Експериментальне підтвердження гіпотези.** Тестування показало, що спроба використати клас INTEGER_NUMBER (з округленням) у алгоритмі, розрахованому на REAL_NUMBER, призводить до порушення контрактів. На відміну від традиційних мов (C++, C#, Java), де така підстановка могла б призвести до тихого спотворення даних або зациклення алгоритму, середовище виконання Eiffel миттєво зупинило роботу програми з помилкою Postcondition violated.
4. **Ефективність Design by Contract.** Продемонстровано, що вбудовані механізми перевірки контрактів (зокрема ensure та invariant) дозволяють перетворити неявні логічні помилки проектування на явні виключення під час виконання. Це дає змогу локалізувати проблему безпосередньо в місці її виникнення (наприклад, у методі ділення), а не аналізувати некоректні результати роботи всього алгоритму.

Підсумовуючи, можна стверджувати, що універсальний поліморфізм є потужним, але таким інструментом, який вимагає суворого дотримання семантичних правил підтипізації. Використання мов із підтримкою Design by Contract, таких як Eiffel, є ефективним засобом захисту від порушень принципу LSP, гарантуючи надійність та передбачуваність поведінки складних програмних систем.

6. Література

1) Піскунов О. Г. Компілятори C++ та розробка консольних додатків (Частина 1) [Електронний ресурс]. – 2021. – 163 с.
   Режим доступу: https://www.researchgate.net/publication/357093594
2) Піскунов О. Г. Розробка консольних додатків з елементами C++ (Частина 2) [Електронний ресурс]. – 2021. – 98 с.
   Режим доступу: https://www.researchgate.net/publication/350384089
3) Microsoft. C/C++ Language and Standard Libraries Documentation [Електронний ресурс]. – 2022.
   Режим доступу: https://docs.microsoft.com/ru-ru/cpp/cpp/c-cpp-language-and-standard-libraries
4) Омельчук Л. Л. Об’єктно-орієнтоване програмування. Лабораторний практикум. – 2022.
5) Класи та проектування програмного забезпечення [Електронний ресурс].
   Режим доступу: https://www.researchgate.net/publication/395422671
6) Деякі інструментальні засоби документування ПЗ [Електронний ресурс].
   Режим доступу: https://www.researchgate.net/publication/385698190
7) Meyer B. Object-Oriented Software Construction. – 2nd ed. – Prentice Hall, 1997. – 1296 p. (Класика: Design by Contract, LSP, формальні специфікації.)
8) Meyer B. Eiffel: The Language. – Prentice Hall, 1992.
   (Офіційна мова Eiffel, контракти, типова модель поліморфізму.)
9) Meyer B. Touch of Class: Learning to Program Well with Objects and Contracts. – Springer, 2009. (DBC у навчальному форматі.)