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 Contract»

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…… 2

2 Постановка задачі…… 3

3 Теоретична частина…… 6

3.1 Симплекс метод
3.2 Поліморфізм як фундаментальна парадигма ООП
3.3 Принцип підстановки Лісков (LSP)
3.4 Розробка за контрактом (DbC)
3.5 Мова програмування Eiffel

4 Практична частина…… 20

4.1 Архітектура та документація програмної системи
4.2 Реалізація інваріантів та контроль стану
4.3 Засторсування методології Design by Contract (DbC)
4.4 Технічне середовище розробки та компіляція додатку
4.5 Тестування

5 Висновок…… 41

6 Література…… 42

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 Лісков (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:

$$\text{maximize} \quad z = c_1x_1 + c_2x_2 + \ldots + c_nx_n$$

$$\text{subject to}$$

$$A\,x \le b$$

$$x \ge 0$$

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 $x_{n+1}, \ldots, x_{n+m}$ (так звані slack-variables або змінні-залишки):

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

$$\ldots$$

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

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

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: 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

4.5.1. Філософія та архітектурні особливості

Eiffel – це не просто об'єктно-орієнтована мова програмування, а цілісне середовище для реалізації методології побудови якісного програмного забезпечення. Розроблена Бертраном Мейєром у 1986 році, мова базується на концепції, що надійність (reliability) та супровідність (maintainability) коду є важливішими за швидкість написання.

На відміну від «гібридних» мов (C++, Python), Eiffel є чистою об'єктно-орієнтованою мовою: тут абсолютно все є об'єктом, від простих цілих чисел до складних систем. Це дозволяє уникнути багатьох проблем типізації, притаманних змішаним підходам.

Фундаментальною відмінністю Eiffel є те, що механізм Design by Contract (DbC) інтегрований у синтаксис на рівні ядра мови. Якщо в C# чи Java контракти реалізуються через сторонні бібліотеки (наприклад, Code Contracts) або асерти, які часто ігноруються, то в Eiffel контракти є невід'ємною частиною визначення класу. Вони формують «паспорт» методу, гарантуючи його коректну поведінку.

Принцип єдиної відповідальності на рівні мови

Eiffel втілює принцип Command-Query Separation (CQS): кожна підпрограма або змінює стан об'єкта (команда), або повертає значення (запит), але ніколи не робить обидві речі одночасно. Це забезпечує передбачуваність коду та спрощує розуміння побічних ефектів.

4.5.2. Інтегрована контрактна система

Синтаксис Eiffel зобов'язує розробника мислити в термінах специфікацій, а не лише реалізації. Основні блоки включають:

- **require (Передумови)**: Визначають зобов'язання клієнта. Спроба викликати метод з порушеними передумовами вважається багом у коді, який викликає, а не в самому методі.
- **ensure (Постумови)**: Визначають зобов'язання постачальника (класу). Це гарантія того, що метод виконав роботу правильно. У контексті даної роботи саме блок ensure дозволяє математично верифікувати точність обчислень (наприклад, (Result - expected).abs <= epsilon).
- **invariant (Інваріант класу)**: Логічний закон, який діє для об'єкта постійно. Це дозволяє автоматично відловлювати перехід об'єкта у некоректний стан (наприклад, порушення невід'ємності змінних у симплекс-методі).
- **check**: Аналог assert для перевірки припущень у середині алгоритмів.

- **loop ... invariant ... variant**: Унікальні конструкції для циклів, що дозволяють довести не лише правильність ітерацій, а й факт завершення циклу (захист від нескінченних циклів).

Рівні перевірки контрактів

Eiffel надає гнучкість у виборі рівня контрактних перевірок:

1. **No assertions**: Повне вимкнення всіх перевірок (максимальна продуктивність)
2. **Require**: Перевірка лише передумов
3. **Ensure**: Перевірка передумов та постумов
4. **Invariant**: Додатково перевірка інваріантів класу
5. **Loop**: Перевірка інваріантів циклів
6. **Check**: Повна перевірка включно з блоками check
7. **All**: Абсолютно всі можливі перевірки

Це дозволяє балансувати між надійністю під час розробки та продуктивністю в релізній версії.

4.5.3. Система типів та узагальнення

Статична типізація з виведенням типів

Eiffel використовує строгу статичну типізацію, що дозволяє виявляти помилки типів на етапі компіляції. Водночас мова підтримує локальне виведення типів для локальних змінних через ключове слово local, що зменшує вербозність коду.

Узагальнені класи (Generics)

Механізм узагальнень у Eiffel є більш потужним, ніж у Java або C#. Він підтримує:

- **Обмежені параметри типів**: class SORTED_LIST[G -> COMPARABLE] гарантує, що G підтримує порівняння
- **Множинні параметри типів**: class HASH_TABLE[V, K -> HASHABLE]
- **Коваріантність**: Нащадок може уточнювати типи параметрів

Void Safety

Сучасні версії Eiffel (починаючи з версії 7.0) включають механізм Void Safety, який на рівні компілятора запобігає помилкам NullPointerException. Компілятор аналізує потоки даних та гарантує, що об'єктні посилання не будуть розіменовані, якщо вони можуть бути Void (null).

4.5.4. Керування спадкуванням та LSP

Eiffel надає найдосконаліший серед сучасних мов механізм контролю контрактів при спадкуванні, що робить його ідеальним інструментом для дослідження Принципу підстановки Лісков:

1. **Послаблення передумов (require else)**: Нащадок може вимагати менше, ніж предок, але не більше. Це гарантує, що код, який працює з предком, зможе працювати і з нащадком.
2. **Посилення постумов (ensure then)**: Нащадок зобов'язаний гарантувати все, що обіцяв предок, і може додати нові гарантії. Саме цей механізм унеможливлює «тихе» заміщення точних обчислень (REAL) на наближені (INTEGER) без виникнення помилки, що є ключовим для експериментальної частини роботи.

Множинне спадкування

На відміну від Java та C#, Eiffel підтримує повноцінне множинне спадкування з елегантним вирішенням проблеми "діаманта":

- **Rename**: Перейменування успадкованих методів для уникнення конфліктів
- **Undefine**: Скасування успадкованої реалізації
- **Redefine**: Перевизначення методів з контролем контрактів
- **Select**: Вибір однієї з конфліктуючих реалізацій

4.5.5. Модель компіляції Eiffel

Архітектура компілятора

Компілятор Eiffel виконує багатопрохідну компіляцію з наступними етапами:

Етап 1: Лексичний та синтаксичний аналіз

- Токенізація вихідного коду
- Побудова абстрактного синтаксичного дерева (AST)
- Перевірка синтаксичної коректності

Етап 2: Аналіз типів та контрактів

- Перевірка сумісності типів
- Валідація контрактів спадкування (правила require else/ensure then)
- Перевірка інваріантів класу на непротиріччя
- Аналіз Void Safety

Етап 3: Розгортання узагальнень

- Інстанціювання узагальнених класів для конкретних типів
- Оптимізація коду для примітивних типів

Етап 4: Генерація проміжного коду

- Створення C-коду (в класичному компіляторі) або .NET IL (у версії для .NET)
- Інтеграція з runtime системою Eiffel

Етап 5: Оптимізація

- Inline-підстановка простих методів
- Dead code elimination
- Вибіркове вимкнення контрактних перевірок залежно від рівня assertion

Технологія Melting Ice

Melting Ice - революційна технологія інкрементальної компіляції, яка робить EiffelStudio одним із найшвидших IDE для мов зі статичною типізацією:

Принцип роботи:

1. **Freezing (Заморожування)**: Повна компіляція всієї системи в C-код з наступною компіляцією нативним компілятором (gcc/clang/MSVC). Це створює "заморожену" базову версію.
2. **Melting (Розморожування)**: При зміні коду лише змінені класи компілюються в інтерпретовану форму (bytecode) та "допаюються" до замороженої системи. Це відбувається за секунди, а не хвилини.
3. **Finalization (Фіналізація)**: Підготовка релізної версії з повною оптимізацією та вимкненням контрактів.

Переваги Melting Ice:

- Компіляція великих проектів (>100K рядків) за 2-5 секунд
- Миттєве тестування змін без повної перекомпіляції
- Збереження стану налагодження між компіляціями

Режими компіляції

Workbench Mode (Режим розробки):

- Усі контракти активні
- Максимальні перевірки під час виконання
- Підтримка Melting Ice
- Детальні повідомлення про помилки з трасуванням стека

Finalized Mode (Релізний режим):

- Контракти можна вимкнути або залишити вибірково
- Агресивна оптимізація компілятора
- Генерація нативного коду без інтерпретованих вставок
- Продуктивність на рівні C/C++

4.5.6. EiffelStudio: Більше ніж IDE

Розробка мовою Eiffel нерозривно пов'язана із середовищем EiffelStudio, яке надає унікальні можливості для підвищення якості коду та швидкості розробки:

Різні представлення коду (Views)

Це одна з найпотужніших фіч для аналізу архітектури:

- **Basic Text View**: Звичайний вихідний код класу
- **Contract View**: Показує лише сигнатури методів та їхні контракти (без тіла do). Це дозволяє програмісту читати клас як специфікацію, не відволікаючись на деталі реалізації
- **Flat View**: Автоматично розгортає ієрархію спадкування, показуючи всі методи (включаючи успадковані) так, ніби вони були написані в цьому класі
- **Interface View**: Публічний інтерфейс класу без приватних компонентів
- **Ancestors/Descendants**: Візуалізація ієрархії успадкування

Інструменти аналізу та рефакторингу

AutoTest:

- Автоматична генерація тестів на основі контрактів
- Використання методів push-button verification
- Генерація граничних випадків

Class/Feature Relations:

- Граф залежностей між класами
- Аналіз впливу змін (impact analysis)
- Виявлення невикористаного коду

Metrics and Statistics:

- Цикломатична складність методів
- Глибина ієрархії успадкування
- Зв'язаність та зчеплення класів

Автоматичне діаграмування (BON/UML)

Середовище вміє генерувати графічні діаграми класів безпосередньо з коду і навпаки (round-trip engineering). Зміна стрілки на діаграмі автоматично змінює код, що забезпечує синхронізацію документації з реальною архітектурою.

Інтегрований налагоджувач

- Покрокове виконання з візуалізацією стану об'єктів
- Інспекція контрактів у режимі реального часу
- Automatic breakpoints на порушеннях контрактів
- Call stack з повною інформацією про контекст

4.5.7. Самодокументування

Завдяки тому, що контракти є частиною коду, Eiffel вирішує одвічну проблему застарілої документації. У мовах на кшталт Java Javadoc пишеться в коментарях і може не відповідати реальному коду. В Eiffel документація генерується з виконуваних контрактів. Якщо змінюється логіка (наприклад, постумова), автоматично змінюється і документація. Це робить код самодокументованим (self-documenting code), що є стандартом для критично важливих систем.

4.5.8. Взаємодія з іншими мовами

Eiffel підтримує інтеграцію з зовнішнім кодом через механізм external:

- **C/C++ Integration**: Пряме викликання C-функцій та використання C-бібліотек
- **.NET Integration**: Версія Eiffel для .NET дозволяє використовувати будь-які .NET бібліотеки
- **Java Integration**: Через JNI або спеціальні обгортки

Це дозволяє використовувати величезну екосистему існуючих бібліотек, компенсуючи відносно невелику власну бібліотечну базу.

4.5.9. Переваги та недоліки мови

Для об'єктивної оцінки інструментарію необхідно виділити його сильні та слабкі сторони в контексті сучасної розробки.

Переваги:

1. **Надійність**: Жорстка статична типізація та DbC дозволяють виявляти до 90% логічних помилок ще на етапі розробки та тестування.

2. **Множинне спадкування**: На відміну від Java/C#, Eiffel підтримує повноцінне множинне спадкування з вирішенням конфліктів імен (renaming) та повторним використанням коду.
3. **Void Safety**: Сучасні версії Eiffel мають механізм захисту від розіменування нульових вказівників (NullPointerException) на рівні компілятора.
4. **Читабельність**: Синтаксис, наближений до англійської мови (відсутність фігурних дужок, використання ключових слів), робить код зрозумілим навіть для непрограмістів (бізнес-аналітиків).
5. **Продуктивність компіляції**: Технологія Melting Ice забезпечує швидкість ітерації розробки, недосяжну для більшості компільованих мов.
6. **Математична обґрунтованість**: Контракти дозволяють формально верифікувати коректність алгоритмів.

Недоліки:

1. **Академічність та поріг входження**: Мова вимагає високої культури програмування та розуміння теорії типів. Це не мова для «швидких скриптів».
2. **Екосистема**: Кількість бібліотек та фреймворків значно менша порівняно з Python, Java або JavaScript, що ускладнює використання Eiffel для веб-розробки або Data Science.
3. **Продуктивність виконання в Workbench**: У режимі налагодження (з увімкненими перевірками всіх контрактів) виконання програм може бути повільнішим. Однак, у релізній версії контракти можна вимкнути, досягаючи продуктивності рівня C++.
4. **Обмежена спільнота**: Менша кількість розробників означає менше прикладів, туторіалів та відповідей на Stack Overflow.
5. **Комерційні обмеження**: Хоча існує відкрита версія (EiffelStudio GPL), повнофункціональна комерційна версія є платною.

4.5.10. Галузі застосування

Eiffel знаходить своє застосування в областях, де критично важлива надійність:

- **Фінансові системи**: Банківське програмне забезпечення, торгівельні платформи
- **Транспорт**: Системи керування залізничним транспортом (використовується в Swiss Federal Railways)
- **Медичне обладнання**: Програмне забезпечення для медичних пристроїв
- **Aerospace**: Компоненти авіаційних систем
- **Освіта**: Викладання принципів якісного програмного забезпечення

4. Практична частина

4.1 Архітектура та документація 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):

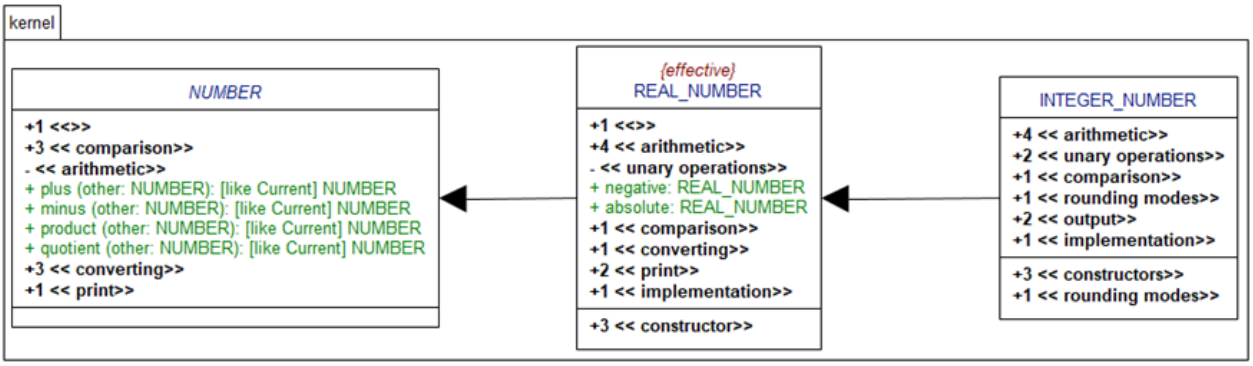
4.1.1 Структура кластерів системи

1. Kernel – Числова абстракція

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

Основні класи:

- **NUMBER**: Абстрактний предок, що задає інтерфейс арифметичних операцій.
- **REAL_NUMBER**: Реалізація чисел з плаваючою комою (IEEE 754). Цей клас виступає еталоном ("контрактним ідеалом"), оскільки містить суворі постумови для перевірки точності операцій.
- **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.

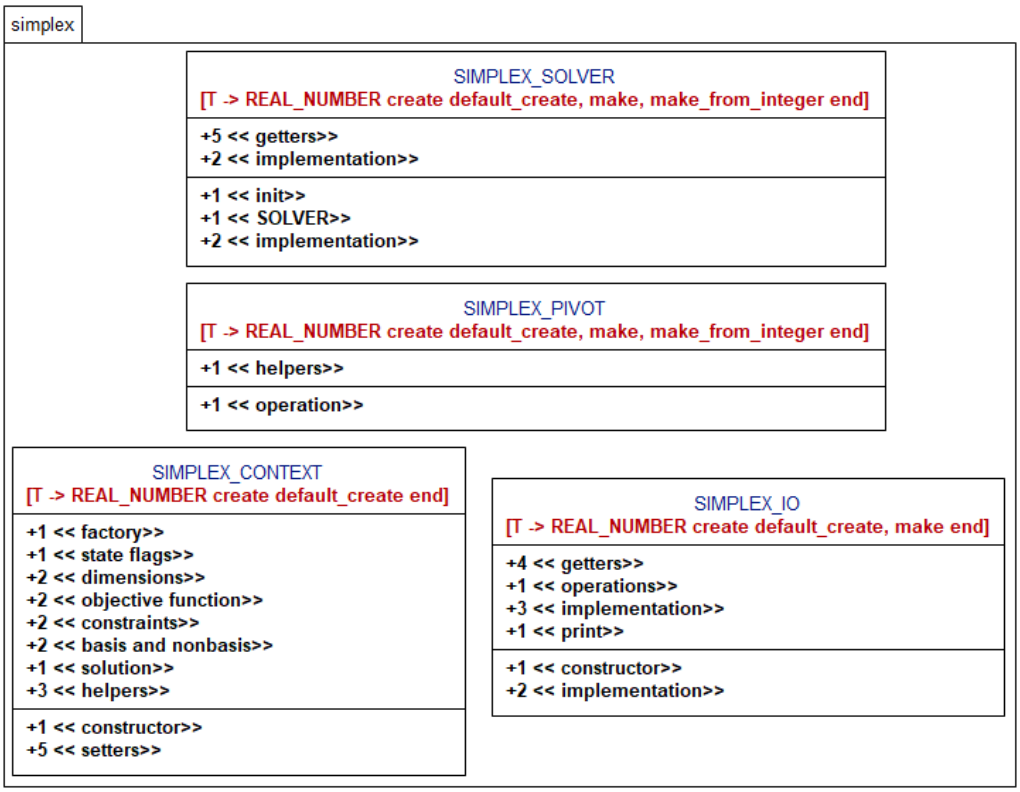


2. Simplex – Алгоритмічний шар (логіка розв'язання)

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

Основні класи:

- **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).
- **SIMPLEX_IO [T]**: Парсер вхідних даних зі стандартного потоку введення.



3. Application – Прикладний шар (інтерфейс)

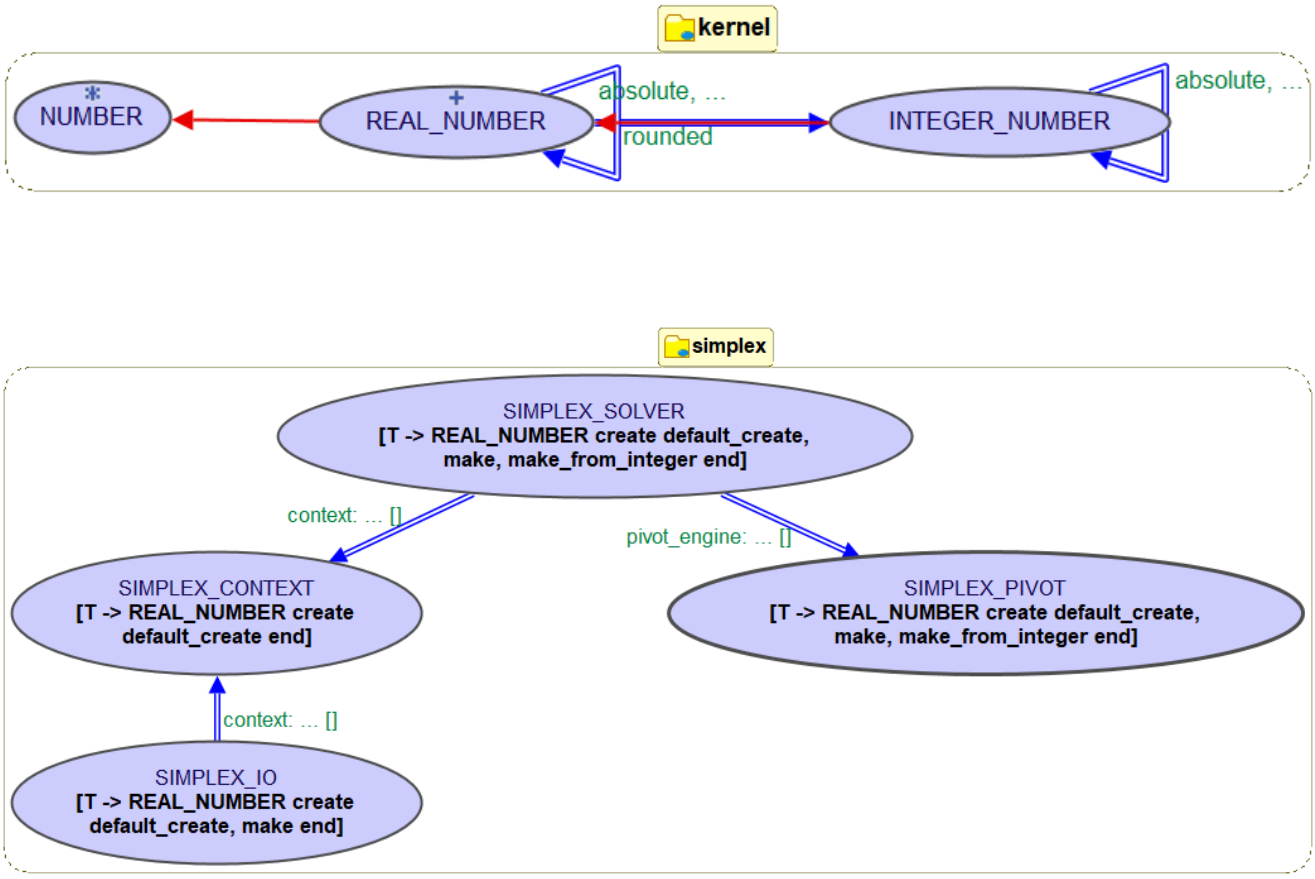
Забезпечує взаємодію з користувачем та введення/виведення.

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

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

4.1.2 Діаграми архітектури

BOM (Bill of Materials) - Огляд структури проекту



4.1.3 Документація ключових класів

Середовище EiffelStudio автоматично генерує вичерпну документацію для всіх класів проекту, включаючи описи відношень (Relations), специфікації контрактів (Contracts), текстові описи (Text) та структурні схеми (Chart). Для цілей даного дослідження найбільш важливими описи Relations, що демонструють ієрархію наслідування та зв'язки між компонентами системи, оскільки саме вони наочно ілюструють архітектурну проблему порушення LSP.

Нижче наведено описи Relations для трьох ключових класів, які безпосередньо пов'язані з експериментом:

Клас NUMBER (Абстрактний базовий тип)

NUMBER є коренем числової ієрархії та визначає контрактний інтерфейс для всіх арифметичних операцій. Relations показує, що цей клас є предком для REAL_NUMBER, а також використовується як параметр типу в узагальнених класах симплекс-алгоритму.

deferred class
NUMBER

Ancestors
ANY

Descendants
REAL_NUMBER

Clients
INTEGER_NUMBER
NUMBER*
REAL_NUMBER
SIMPLEX_PIVOT [T -> REAL_NUMBER **create** default_create, make, make_from_integer **end**]

Suppliers
BOOLEAN
NUMBER*
REAL_64
STRING_8

Клас INTEGER_NUMBER

INTEGER_NUMBER є критичним компонентом експерименту, оскільки саме його підстановка замість REAL_NUMBER призводить до порушення LSP. Relations демонструє, що клас успадковується від REAL_NUMBER, що робить його синтаксично коректним для поліморфної підстановки, але семантично некоректним через зміну поведінки арифметичних операцій.

class
INTEGER_NUMBER

General
cluster: **KERNEL**
description: "Integer numbers with rounding - the LSP violation experiment"
create: make, make_from_integer, default_create

Ancestors
REAL_NUMBER

Queries
absolute: INTEGER_NUMBER
debug_output: STRING_8
is_equal (other: [**like Current**] INTEGER_NUMBER): BOOLEAN
is_greater **alias** ">" (other: NUMBER): BOOLEAN
is_less **alias** "<" (other: NUMBER): BOOLEAN
minus **alias** "-" (other: NUMBER): INTEGER_NUMBER
negative **alias** "-": INTEGER_NUMBER
out: STRING_8
plus **alias** "+" (other: NUMBER): INTEGER_NUMBER
product **alias** "*" (other: NUMBER): INTEGER_NUMBER
quotient **alias** "/" (other: NUMBER): INTEGER_NUMBER
rounded: INTEGER_NUMBER
to_integer: INTEGER_32
to_string: STRING_8

Constraints
always integer

Клас SIMPLEX_PIVOT

SIMPLEX_PIVOT є місцем, де теоретична проблема LSP проявляється на практиці. Relations показує, що клас параметризований типом T, обмеженим до REAL_NUMBER, і використовує арифметичні операції цього типу. Саме під час виконання pivot-операції (ділення для обчислення коефіцієнтів) контракти базового класу порушуються, якщо T інстанціюється як INTEGER_NUMBER.

class
SIMPLEX_PIVOT [T -> REAL_NUMBER **create** default_create, make, make_from_integer **end**]

Ancestors
ANY

Clients
SIMPLEX_PIVOT [T -> REAL_NUMBER **create** default_create, make, make_from_integer **end**]
SIMPLEX_SOLVER [T -> REAL_NUMBER **create** default_create, make, make_from_integer **end**]

Suppliers
ARRAYED_LIST [G]
ARRAYED_LIST_ITERATION_CURSOR [G]
BOOLEAN
HASH_TABLE [G, K -> **detachable** HASHABLE]
INTEGER_32
ITERATION_CURSOR* [G]
NUMBER*
REAL_64
REAL_NUMBER
SIMPLEX_CONTEXT [T -> REAL_NUMBER **create** default_create **end**]
SIMPLEX_PIVOT [T -> REAL_NUMBER **create** default_create, make, make_from_integer **end**]

Детальний аналіз контрактів цих класів та механізмів їх порушення буде представлено у розділі 4.3 "Застосування методології Design by Contract (DbC)".

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. Перелік інваріантів з документації:

Constraints
non negative dimensions
x size matches total
c count matches n
b count matches m
a count matches m
b count matches m
n count matches n
basis and nonbasis disjoint
basis and nonbasis cover all
all b non negative

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. Середовище EiffelStudio автоматично генерує документацію контрактів (Contracts) для кожного класу, що включає специфікації передумов (require), постумов (ensure) та інваріантів (invariant). Ці документи є

критично важливими для розуміння того, як механізм DbC виявляє порушення LSP на етапі виконання.

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. Повна специфікація передумов класу SIMPLEX_PIVOT наведена нижче:

```
note
        description: "Performs the Pivot algebraic transformation"

class interface
        SIMPLEX_PIVOT [T -> REAL_NUMBER create default_create, make, make_from_integer end]

create
        default_create

feature -- operation

        pivot (ctx: SIMPLEX_CONTEXT [T]; leaving_var: INTEGER_32; entering_var: INTEGER_32)
                require
                        valid_leaving: ctx.is_basic (leaving_var)
                        valid_entering: ctx.is_nonbasic (entering_var)
                ensure
                        basis_swapped: ctx.is_basic (entering_var) and ctx.is_nonbasic (leaving_var)
                        feasible: across
                                        ctx.b as b
                                all
                                        attached ctx.b_values [b.item] as val and then val.value
                                                                                >= -0.00001
                                end

end -- class SIMPLEX_PIVOT
```

Б. Постумови (Postconditions) та їх порушення

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

Контракти базового класу NUMBER

Абстрактний клас NUMBER визначає фундаментальні контракти для всіх арифметичних операцій, включаючи точність обчислень. Ці контракти є обов'язковими для всіх нащадків:

```
note
        description: "Abstract numeric type - base for all number implementations"

deferred class interface
        NUMBER

feature

        value: REAL_64

feature -- comparison

        is_less alias "<" (other: NUMBER): BOOLEAN
                require
                        other_exists: other /= Void
                ensure
                        definition: Result = (value < other.value)

        is_greater alias ">" (other: NUMBER): BOOLEAN
                require
                        other_exists: other /= Void
                ensure
                        definition: Result = (value > other.value)

        is_equal (other: like Current): BOOLEAN
                        -- Is other attached to an object considered
                        -- equal to current object?

feature -- arithmetic

        plus alias "+" (other: NUMBER): like Current
                require
                        other_exists: other /= Void
                ensure
                        not_void: Result /= Void
                        commutative: (Result.value - (value + other.value)).abs < 0.0001

        minus alias "-" (other: NUMBER): like Current
                require
                        other_exists: other /= Void
                ensure
                        not_void: Result /= Void
                        correct_math: (Result.value - (value - other.value)).abs < 0.0001

        product alias "*" (other: NUMBER): like Current
                require
                        other_exists: other /= Void
                ensure
                        not_void: Result /= Void
                        correct_math: (Result.value - (value * other.value)).abs < 0.0001

        quotient alias "/" (other: NUMBER): like Current
```

```
                require
                        other_exists: other /= Void
                        not_zero: other.value /= 0.0
                ensure
                        not_void: Result /= Void
                        correct_division: (Result.value - (value / other.value)).abs < 0.0001

feature -- converting

        to_double: REAL_64
                ensure
                        same_value: Result = value

        to_integer: INTEGER_32

        to_string: STRING_8
                ensure
                        not_empty: not Result.is_empty

feature -- print

        out: STRING_8
                        -- New string containing terse printable representation
                        -- of current object

end -- class NUMBER
```

Реалізація контрактів у класі REAL_NUMBER

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)". Повна документація контрактів REAL_NUMBER:

```
note
        description: "Real numbers - precise floating-point arithmetic (IEEE 754)"

class interface
        REAL_NUMBER

create
        make,
        make_from_integer,
        default_create

feature

        value: REAL_64
```

```
feature -- arithmetic

        plus alias "+" (other: NUMBER): REAL_NUMBER
                ensure then
                        exact_sum: (Result.value - (value + other.value)).abs < 0.000001

        minus alias "-" (other: NUMBER): REAL_NUMBER
                ensure then
                        exact_difference: (Result.value - (value - other.value)).abs < 0.000001

        product alias "*" (other: NUMBER): REAL_NUMBER
                ensure then
                        exact_product: (Result.value - (value * other.value)).abs < 0.000001

        quotient alias "/" (other: NUMBER): REAL_NUMBER
                ensure then
                        exact_division: (Result.value - (value / other.value)).abs < 0.000001
                        inverse_holds: ((Result.value * other.value) - value).abs < 0.001

feature -- unary operations

        negative alias "-": REAL_NUMBER
                ensure
                        correct_negation: Result.value = - value

        absolute: REAL_NUMBER
                ensure
                        non_negative: Result.value >= 0.to_double
                        correct_for_positive: value >= 0.to_double implies Result.value = value
                        correct_for_negative: value < 0.to_double implies Result.value = - value

feature -- comparison

        is_equal (other: like Current): BOOLEAN
                        -- Is other attached to an object considered
                        -- equal to current object?
                ensure then
                        symmetry: (other /= Void implies (Result = other.is_equal (Current)))

feature -- converting

        rounded: INTEGER_NUMBER
                ensure
                        close_to_original: (Result.value - value).abs <= 0.5

feature -- print

        out: STRING_8
                        -- New string containing terse printable representation
                        -- of current object
                ensure then
                        not_void: Result /= Void

        debug_output: STRING_8
                        -- detailed printing for debugging
                ensure
                        not_empty: not Result.is_empty

end -- class REAL_NUMBER
```

Порушення контрактів у класі INTEGER_NUMBER

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**.

```
note
        description: "Integer numbers with rounding - the LSP violation experiment"
        warning: "This class VIOLATES contracts from REAL_NUMBER parent!"
        purpose: "Research tool to demonstrate Design by Contract catches semantic bugs"

class interface
        INTEGER_NUMBER

create
        make,
        make_from_integer,
        default_create

feature -- arithmetic

        plus alias "+" (other: NUMBER): INTEGER_NUMBER

        minus alias "-" (other: NUMBER): INTEGER_NUMBER

        product alias "*" (other: NUMBER): INTEGER_NUMBER
                        -- !!! double rounding !!!

        quotient alias "/" (other: NUMBER): INTEGER_NUMBER
                        -- !!! double rounding !!

feature -- unary operations

        negative alias "-": INTEGER_NUMBER

        absolute: INTEGER_NUMBER

feature -- comparison

        is_equal (other: like Current): BOOLEAN
                        -- Is other attached to an object considered
                        -- equal to current object?
```

```
feature -- rounding modes

	set_rounding_mode (mode: INTEGER_32)
				-- 0: to_even (banker's), 1: away_from_zero, 2: toward_zero, 3:
toward_negative, 4: toward_positive
			require
				valid_mode: mode >= 0 and mode <= 4
			ensure
				mode_set: rounding_mode = mode

	rounding_mode: INTEGER_32

feature -- output

	out: STRING_8
				-- New string containing terse printable representation
				-- of current object

	debug_output: STRING_8
				-- detailed printing for debugging

invariant
	always_integer: internal_value = internal_value.truncated_to_integer.to_double

end -- class INTEGER_NUMBER
```

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.

Контракти класу SIMPLEX_CONTEXT

Клас SIMPLEX_CONTEXT підтримує інваріанти симплекс-таблиці, зокрема допустимість базисних змінних:

```
note
	description: "Context for simplex algorithm."

class interface
	SIMPLEX_CONTEXT [T -> REAL_NUMBER create default_create end]

create
	make

feature -- factory

	make_zero: T

feature -- state flags

	use_invariants: BOOLEAN
```

```
feature -- dimensions

        num_constraints: INTEGER_32

        num_variables: INTEGER_32

feature -- objective function

        c: HASH_TABLE [T, INTEGER_32]

        v: T

feature -- constraints

        b_values: HASH_TABLE [T, INTEGER_32]

        a: HASH_TABLE [HASH_TABLE [T, INTEGER_32], INTEGER_32]

feature -- basis and nonbasis

        b: ARRAYED_LIST [INTEGER_32]

        n: ARRAYED_LIST [INTEGER_32]

feature -- solution

        x: ARRAY [T]

feature -- setters

        enable_invariants

        disable_invariants

        set_dimensions (a_m, a_n: INTEGER_32)

        reset

        set_v (new_v: T)

feature -- helpers

        is_valid_index (idx: INTEGER_32): BOOLEAN

        is_basic (idx: INTEGER_32): BOOLEAN

        is_nonbasic (idx: INTEGER_32): BOOLEAN

invariant
        non_negative_dimensions: num_constraints >= 0 and num_variables >= 0
        x_size_matches_total: x.count = num_variables + num_constraints
        c_count_matches_n: use_invariants implies c.count = num_variables
        b_count_matches_m: use_invariants implies b_values.count = num_constraints
        a_count_matches_m: use_invariants implies a.count = num_constraints
        b_count_matches_m: use_invariants implies b.count = num_constraints
        n_count_matches_n: use_invariants implies n.count = num_variables
        basis_and_nonbasis_disjoint: use_invariants implies (across
                                b as bi
```

```
                all
                        not n.has (bi.item)
                end)
        basis_and_nonbasis_cover_all: use_invariants implies (across
                        1 |..| (num_variables + num_constraints) as i
                all
                        b.has (i.item) or n.has (i.item)
                end)
        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)

end -- class SIMPLEX_CONTEXT
```

Сценарій порушення контракту

Коли виконується операція Pivot:

1. Обчислюється коефіцієнт (factor) через операцію ділення.
2. Якщо використовується INTEGER_NUMBER, коефіцієнт округлюється.
3. Після завершення операції Eiffel Runtime System перевіряє успадковану постумову exact_division.
4. Система детектує невідповідність (наприклад, 7/2 = 3, але 3*2 ≠ 7) і зупиняє виконання з виключенням **Postcondition violated**.

Таким чином, механізм DbC дозволяє перетворити семантичну помилку (неправильне використання спадкування для моделювання числових множин) на явну помилку часу виконання, підтверджуючи гіпотезу роботи. Порівняно з традиційними мовами (C++, Java, C#), де така підстановка призвела б до "тихого" спотворення результатів обчислень, Eiffel забезпечує автоматичну детекцію архітектурної помилки на рівні контрактної системи.

4.4 Технічне середовище розробки та компіляція додатку

4.6.1. Специфікація програмного забезпечення

Для реалізації даного проекту було використано наступне програмне забезпечення:

EiffelStudio

- **Версія**: EiffelStudio 25.02 (win64)
- **Платформа**: Windows 11 (64-bit)
- **Офіційний сайт**: https://www.eiffel.com/
- **Завантаження**: https://account.eiffel.com/downloads

C-компілятор

EiffelStudio використовує C як проміжну мову для генерації нативного коду. Для фінальної компіляції C-коду використовувався зовнішній компілятор:

GNU Compiler Collection (GCC)

- Дистрибутив: **MSYS2 / MinGW** (платформа - mingw64)
- Версія: **GCC 12.2.0**
- Компоновка та збірка виконуються через стандартний GCC toolchain

Даний компілятор автоматично підхоплюється EiffelStudio з оточення (PATH) і використовується для генерації виконуваного файлу (.exe).

4.6.2 Компіляція проекту SimplexEiffel

Компіляція відбувалася через EiffelStudio IDE

1. **Відкриття проекту:**
   - Запустіть EiffelStudio
   - Виберіть File -> Open Project
   - Знайдіть файл simplex_eiffel.ecf у кореневій папці проекту
   - Натисніть Open
2. **Налаштування режиму компіляції:**

   Для розробки (Workbench Mode): Project -> Compile (використовується за замовчуванням)

   Для релізу (Finalized Mode): Project -> Finalize (Генерується оптимізований нативний код)

3. **Процес компіляції:**

   При першій компіляції (Freezing) EiffelStudio згенерує C-код у папці EIFGENs/. C-компілятор створить виконуваний файл. При наступних змінах спрацює Melting Ice

4. **Результат компіляції:**

   Workbench: EIFGENs/simplex_eiffel/W_code/{project_name}.exe

   Finalized: EIFGENs/simplex_eiffel/F_code/{project_name}.exe

5. **Запуск з IDE:** Меню: Execution -> Run Without Debugging

4.5 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.

Автоматизація процесу тестування

Для забезпечення повторюваності експериментів та систематичного порівняння поведінки системи з двома типами даних було розроблено автоматизований тестовий сценарій у вигляді bat-скрипта. Скрипт виконує наступні кроки для кожного вхідного файлу:

1. Запускає програму у **режимі REAL** (без параметрів командного рядка)
2. Запускає програму у **режимі INTEGER** (з параметром -i)
3. Зберігає результати виконання (включаючи стандартний вивід та повідомлення про помилки) у окремі файли
4. Фіксує код завершення процесу для визначення успішності виконання

Фрагмент bat-скрипта:

```
echo ========================================
echo Starting SimplexEiffel Tests (Real & Integer)
echo ========================================

for %%f in (%INPUT_DIR%\*.txt) do (
    set "FILENAME=%%~nf"
    echo [TESTING] %%f ...

    :: 1. REAL MODE
    type "%%f" | "%EXE_PATH%" > "%OUTPUT_DIR%\!FILENAME!_real.txt" 2>&1
    if !errorlevel! equ 0 (
        echo    -- Real mode: OK. Saved to !FILENAME!_real.txt
    ) else (
        echo    -- Real mode: ERROR.
    )

    :: 2. INTEGER MODE
    type "%%f" | "%EXE_PATH%" -i > "%OUTPUT_DIR%\!FILENAME!_integer.txt" 2>&1
    if !errorlevel! equ 0 (
        echo    -- Integer mode: OK. Saved to !FILENAME!_integer.txt
    ) else (
        echo    -- Integer mode: ERROR.
    )

    echo.
)
```

Набір тестових даних

Тестування проводилося на наборі вхідних даних, що покривають різні сценарії роботи симплекс-методу. Кожен тест зберігався у окремому текстовому файлі у форматі:

<коефіцієнти цільової функції c>

<рядки матриці обмежень A та вільні члени b>

Опис тестових задач:

1. **Problem 1** (Стандартна ЗЛП):

3 4

2 3 7

2 1 5

2. **Problem 2** ("Integer Safe"):

5 3

1 0 5

0 1 4

1 1 8

Всі коефіцієнти підібрані так, що результати ділення є цілими числами на всіх ітераціях.

3. **Problem 3** (Стандартна ЗЛП з від'ємними коефіцієнтами):

3 4

1 2 14

3 -1 9

1 -1 2

4. **Problem 4** (Unbounded – необмежена задача):

2 1

-1 1 5

Цільова функція необмежена зверху.

5. **Problem 5** (Задача з ризиком зациклення):

10 -57 -9 -24

0.5 -5.5 -2.5 9 0

0.5 -1.5 -0.5 1 0

1 0 0 0 1

Вхідні дані містять дробові коефіцієнти, що гарантовано призводять до порушення контракту при ініціалізації INTEGER_NUMBER.

6. **Problem 6** (Infeasible – недопустима задача):

1

1 -5

Початковий базисний план містить від'ємний вільний член (b < 0), що порушує інваріант допустимості.

7. **Problem 7** (Багатовимірна задача):

5 4 3 2 1

1 1 1 1 1 5

2 0 2 0 2 8

Задача з п'ятьма змінними для перевірки масштабованості.

Ці тести охоплюють основні класи еквівалентності: випадки успішного розв'язання, граничні умови (необмеженість, недопустимість), а також специфічні сценарії, де INTEGER_NUMBER не може коректно замінити REAL_NUMBER.

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)

Тест	Опис задачі	Результат (REAL)	Результат (INTEGER)	Висновок
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.)