Науковий вісник НЛТУ України
Scientific Bulletin of UNFU
https://doi.org/10.36930/40340211
Article received 26.02.2024
Article accepted 04.03.2024
UDC 004.[4'23+4'24+43+82]
Correspondence author: A. V. Tkachuk andrewtkachuk@yahoo.com
ISSN 1994-7836 (print)
ISSN 2519-2477 (online)

А. В. Ткачук
Національний технічний університет України "Київський політехнічний інститут ім. Ігоря Сікорського", м. Київ, Україна

АВТОМАТИЗАЦІЯ РЕФАКТОРИНГУ КОДУ ІЗ ВИКОРИСТАННЯМ БАЗИ ЗНАНЬ І ЛОГІЧНИХ ПРАВИЛ

Розглянуто завдання автоматизації процесу заміни початкового програмного коду без зміни поведінки програми (рефакторингу), яку запропоновано вирішити шляхом використання бази знань про початковий код та логічних правил для опису процесу перетворення коду. Окреслено актуальність завдання розроблення нових методів і підходів до рефакторингу з урахуванням обмежень наявних систем і ростом потреб безпосередніх користувачів в автоматизації процесів рефакторингу внаслідок зростання обсягу кодових баз. Наголошено на важливості збереження семантики початкового коду для успішного виконання дій рефакторингу. Запропоновано використовувати онтології та засоби роботи із семантикою (зокрема, такі, що стосуються мови OWL (англ. *Web Ontology Language*) для формалізованого подання початкового коду програмного продукту у базі знань. Запропоновано метод автоматизованого рефакторингу на підставі знань про початковий код, модель подання початкового коду на підставі онтології, метод оброблення моделі на підставі правил, що дають можливість спиратися на семантику початкового коду під час виконання дій рефакторингу. Запропонований метод автоматизованого рефакторингу полягає у його здійсненні з використанням двох перетворень і оброблення проміжного подання коду. Описана модель подання початкового коду складається з двох частин, яку будують на підставі онтології та аксіом, що додають до моделі після проведення синтаксичного аналізу початкового коду. Метод оброблення моделі на підставі правил використовує два типи правил для логічного виведення, що описують додаткові функціональні залежності між сутностями бази знань. Проаналізовано запропоновані методи та моделі, а також описано їх практичне застосування. Проведено ряд випробувань для перевірки теоретичних засад запропонованих моделі та методів на практиці. Встановлено, що програмний інструментарій рефакторингу доцільно реалізувати з використанням клієнт-серверної архітектури для спрощення абстрагування бази знань про код від конкретної кодової бази. Експериментально встановлено поліноміальну залежність необхідного об'єму оперативної пам'яті для виконання завдань від обсягу початкового коду, який обробляють. Продемонстровано пришвидшення до 36 % процесу рефакторингу з використанням розробленого програмного прототипу порівняно з наявними засобами.

Ключові слова: програмне забезпечення; початковий код; онтологія; модель; формалізоване подання знань.

Вступ / Introduction

Автоматизовані засоби рефакторингу дають можливість розробникам забезпечувати підтримку якості програмного коду. Вони стали необхідним елементом у сучасних середовищах розроблення програмного забезпечення, що спрощує підтримку та подальший розвиток кодових баз. Ефективне функціонування автоматизованих засобів рефакторингу значною мірою залежить від автоматизованого аналізу початкового коду. Цей аналіз надає необхідне уявлення про структуру та семантику коду, що дає змогу убезпечитись від помилок в процесі виконання рефакторингу. Наявні системи автоматизованого аналізу й рефакторингу є недостатньо гнучкими для врахування потреб безпосередніх користувачів та виконання відносно нетривіальних завдань рефакторингу [12]. Одним з найбільш важливих принципів рефакторингу є те, що він повинен здійснювати зміну початкового коду без зміни функціональності та поведінки коду. Проте наявні автоматизовані інструменти для рефакторингу, наскільки б точними вони не були, можуть порушувати цей принцип навіть через те, що рефакторинг вони здійснюють, покладаючись винятково на синтаксичний аналіз початкового коду. Такі засоби можуть здійснити зміни, які на перший погляд здаються безпечними, але докорінно змінюють семантику програм. Наприклад, такий інструмент може перемістити метод до базового класу в ієрархії спадкування, вважаючи, що це безпечна операція. Однак у певних сценаріях (де фігурує динамічне зв'язування) це призведе до помилкового виклику методу в іншому класі (з ієрархії успадкування), в такий спосіб неочікувано змінюючи поведінку програми.

Для зменшення кількості помилкових дій через обмеженість засобів рефакторингу та спрощення рутинних дій під час рефакторингу, запропоновано використовувати нові підходи і методи, які дають змогу користувачу формулювати високорівневі, декларативні команди до системи рефакторингу, що реалізується через використанням бази формалізованих знань про початковий код (сутності коду та їх властивості) [13]. Застосування такого "семантичного" підходу дає змогу ві-

Інформація про автора:
Ткачук Андрій Віталійович, аспірант, кафедра системного проектування.
Email: andrewtkachuk@yahoo.com; https://orcid.org/0000-0002-9127-6381

Цитування за ДСТУ: Ткачук А. В. Автоматизація рефакторингу коду із використанням бази знань і логічних правил. Науковий вісник НЛТУ України. 2024, т. 34, № 2. С. 87–93.
Citation APA: Tkachuk, A. V. (2024). Automated code refactoring using a knowledge base and logical rules. *Scientific Bulletin of UNFU*, 34(2), 87–93. https://doi.org/10.36930/40340211

дійти від специфічної реалізації дій рефакторингу під синтаксис конкретної мови програмування і спиратися на сенс, закладений у початковому коді. У даній публікації запропоновано використовувати онтології (наприклад, [2]) для подання початкового коду та здійснення з ним маніпуляцій через проміжне подання [7].

Об'єкт дослідження – процес автоматизованої модифікації початкового коду програмного продукту.

Предмет дослідження – метод автоматизованого рефакторингу початкового коду, що під час своєї роботи враховує не тільки синтаксис, а й семантику, і який базується на використанні формалізованих знань про код.

Мета роботи – розробити метод автоматизованого рефакторингу, що використовують базу знань і логічні правила.

Для досягнення зазначеної мети визначено такі основні *завдання дослідження*:

1) проаналізувати наявні досягнення у сфері виконання рефакторингу на підставі семантики початкового коду, що дасть змогу зробити висновки про доцільність і актуальність застосування таких підходів і врахувати зазначені у напрацюваннях обмеження підходу для покращення власних результатів;
2) розробити схему бази знань, яку будують на підставі онтології про об'єктно-орієнтовану мову програмування, та спосіб внесення фактів про конкретний початковий код до неї, що дасть змогу різнобічно і якомога більш повно описати всю семантику цього коду;
3) удосконалити загальноприйняту послідовність дій, яку виконують під час рефакторингу, додавши етапи внесення фактів про початковий коду у базу знань, виконання логічного виведення та оберненого перетворення сутностей бази знань у початковий код, що зробить можливим виконання рефакторингу на етапі логічного виведення шляхом видозміни семантичного подання початкового коду;
4) розробити метод застосування логічних правил для оброблення інформації, що міститься у базі знань, що дасть змогу виконати складний рефакторинг, описаний послідовністю дій у правилах, виконуючи прості операції (спрощення проходить саме через оброблення коду шляхом зміни подання цього коду, що зберігає семантику);
5) провести перевірку теоретичних засад на практиці шляхом виконання ряду випробувань, що забезпечить підтвердження доцільності використання запропонованих методів і моделі, а також дасть змогу кількісно оцінити економію часу.

Аналіз останніх досліджень та публікацій. У 2011 році було проведено дослідження щодо перетворення початкового коду в онтологію. Внаслідок цього було продемонстровано і підтверджено можливість подання коду через сутності запропонованої онтології. Отже, було доведено, що об'єктно-орієнтована мова програмування як і будь-яка інша предметна область може бути формалізована і подана у вигляді, який можна використовувати для подальшої машинного оброблення, у т.ч. і з використанням засобів логічного виведення в онтології [6].

У наступному році було опубліковано статтю [9], яка пропонує підхід до виявлення шаблонів проектування у початковому коді програм, використовуючи концепції семантичного вебу [3]. Основна ідея полягає в тому, щоб створити онтології, які описують як самі шаблони проектування, так і початковий код програм, і використовувати ці онтології для автоматизованого виявлення шаблонів.

Для опису шаблонів проектування створено онтологію [9], яка містить інформацію про структуру шаблону, його основні характеристики, учасників та зв'язки між ними. Це дає змогу системі розуміти сутність кожного шаблону та виявляти його в коді. Також створено онтологію для опису початкового коду програм. Ця онтологія містить інформацію про класи, методи, змінні та їхні взаємозв'язки. Це дає змогу системі аналізувати код програм і розуміти його структуру та функціональність. Використовуючи ці дві онтології, система проводить аналіз початкового коду програм для виявлення в ньому шаблонів проектування. Для цього вона порівнює структури та зв'язки в коді з описами шаблонів в онтології шаблонів. Якщо знайдено відповідність, система визначає, що в коді присутній певний шаблон.

У 2017 році було наведено систему CodeOntology [2], яка дає змогу здійснювати SPARQL запити до початкового коду, написаного мовою Java. Дослідники використовують онтологію для опису предметної області об'єктно-орієнтованої мови програмування і за допомогою парсера перетворюють початковий код конкретного проєкту у набори даних.

Інший дослідник П. Кесселі [7] пропонує використовувати семантичний підхід до рефакторингу – алгоритм синтезу, який працює разом із системою перевірки, для виконання рефакторингу, що базується на семантиці початкового коду. У роботі порівняно семантичний підхід із традиційними підходами на підставі синтаксису і доведено ефективність першого. На підставі роботи захищено дисертацію PhD.

У 2021 році група вчених [1] дослідила можливість подання початкового коду різнорідних проєктів (код написаний різними мовами програмування) в онтології, що дає можливість розуміти цілісну структуру проєкту за допомогою тільки одного засобу. Дослідження проводилось для мов Python та Java.

Наведені дослідження продовжувались і в наступних роках, що підтверджує важливість і актуальність досліджуваної теми.

Матеріали та методи дослідження. Під час проведення дослідження використовувались такі методи: спостереження, порівняння, аналіз, формалізація, абстрагування, моделювання, експеримент. Ці методи було обрано з урахуванням мети та поставлених завдань дослідження, а також на практичність їх використання при вирішенні проблем автоматизованого аналізу та рефакторингу початкового коду прикладних програм.

Результати дослідження та їх обговорення / Research results and their discussion

На підставі проведеного аналізу літературних джерел та ідеї щодо використання бази знань як основи для автоматизованого виконання рефакторингу було запропоновано наступні методи перетворення та модель подання початкового коду.

Метод автоматизованого рефакторингу на підставі знань про код. Метод, який запропоновано в цьому дослідженні, полягає у здійсненні рефакторингу з використанням двох перетворень і оброблення проміжного подання. Перше перетворення – перетворення сутностей початкового коду в сутності бази знань. Оброблення подання коду полягає у застосуванні визначених методів перевірки узгодженості і повноти, а також в автоматизованому доповненні бази знань, що представляє

початковий код. Така база знань спирається на засоби роботи із семантикою – OWL (англ. *Web Ontology Language*). Друге перетворення – обернене до першого і є перетворенням сутностей бази у синтаксичні конструкції початкового коду [14]. Детально процес виконання такого рефакторингу зображено на блок-схемі (рис. 1).

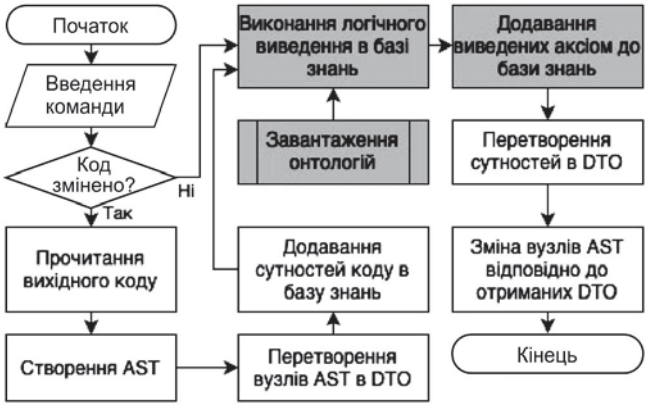

Рис. 1. Блок-схема методу автоматизованого рефакторингу / Block diagram of the automated refactoring method

Кроки запропонованого методу:

1. Користувач вводить команду для виконання рефакторингу, яка може бути запитом інформації про початковий код, зміну початкового коду або ініціалізацію автоматизованого рефакторингу на підставі попередніх знань про дії рефакторингу.
2. Якщо початковий код змінився або не був прочитаний раніше, його зчитують і обробляють засобами конкретної мови програмування. Якщо код не змінився, виконання переходить до логічного виведення в базі знань.
3. Зі зчитаного коду будують абстрактне синтаксичне дерево AST (англ. *Abstract Syntax Tree*), і для кожного вузла створено відповідний об'єкт для передачі даних до бази знань.
4. Створені об'єкти додають в базу знань.
5. Одночасно з цим у базу знань завантажуються онтології, що описують як код, так і предметну область.
6. Після завершення операцій з внесення даних в базу знань запускають логічне виведення на підставі всіх аксіом і здійснюють перевірку на узгодженість. Результатом логічного виведення є набір нових аксіом.
7. Отримані аксіоми додають в базу знань, створюючи повне подання початкового коду з можливими модифікаціями, які наведено аксіомами.
8. Виконується обернене перетворення елементів бази знань в допоміжні об'єкти для передачі даних з бази знань до AST.
9. Відповідно до отриманих об'єктів здійснюють заміну вузлів AST.

Модель початкового коду, що будують на підставі онтології. Оскільки для пропонованих методів фундаментальною властивістю є збереження семантики, то база знань має базуватись на моделі початкового коду, що зберігає семантику. Для виконання цієї умови було обрано набір засобів для роботи із семантикою OWL. Основою моделі є онтологія про об'єктно-орієнтовану мову програмування [2].

У пропонованих методах база знань представляє собою модель початкового коду, що складається з набору TBox аксіом, які описують синтаксис мови програмування (класи, атрибути, відношення) та базовий набір аксіом для класифікації і виведення нових знань про екземпляри. Цей набір завантажується в базу знань з онтології об'єктно-орієнтованої мови програмування. Друга частина бази знань – набір ABox аксіом, які генерують під час синтаксичного аналізу конкретного початкового коду. Внаслідок цього отримуємо базу знань, що містить повний перелік класів, властивостей, відношень і екземплярів, що описують конкретний початковий код. Під час роботи системи логічного виведення всі факти про код, що не були явно вказані, додають в базу знань, а всі суперечності виявляють і повідомляють. Це відповідає вимогам автоматизованого рефакторингу, де код змінюють відповідно до знань про такий рефакторинг у використаній онтології.

Варто зауважити, що така онтологія може містити не тільки знання про конкретний початковий код, а й про усталені загальноприйняті шаблони в програмуванні, що дасть можливість не тільки виявляти порушення цих шаблонів, але одразу їх виправляти, при цьому не змінюючи функціональність означеного коду.

Метод оброблення моделі на підставі правил. Метод оброблення є доповненням до методу автоматизованого рефакторингу. Хоч методи і не залежні один від одного і можуть використовуватись окремо (або як частина інших методів чи підходів), їхнє одночасне застосування дає можливість отримати якісно нові результати. Метод оброблення використовує правила логічного виведення і застосовує набір правил до фактів про початковий код, які містяться в базі знань.

Метод містить два типи логічних правил:

1. Правила: це набори правил, які об'єднують для виконання однієї дії рефакторингу. Вони описують екземпляри, їх властивості і функціональні залежності між ними з точки зору кроків, необхідних для досягнення поставленої цілі конкретного рефакторингу. Ці правила розміщують в базі знань і застосовують до моделі початкового коду під час процесу логічного виведення, без додаткового втручання користувача.
2. Запити: ці логічні правила відповідають можливостям системи автоматизованого рефакторингу. Вони дають можливість знаходити екземпляри, що відповідають певним критеріям, шукати властивості та інше. Такі запити можуть бути введені користувачем або транслюватися системою рефакторингу в правила перед виконанням в базі знань.

Для перевірки теоретичних засад запропонованих підходів на практиці було розроблено програмний прототип для автоматизованого рефакторингу із використанням запропонованих методів і моделі. Для розроблення було використано клієнт-серверну архітектуру, що дало змогу відокремити (абстрагувати) базу знань від клієнтів-споживачів функціональності рефакторингу і використовувати додаткові потужності для проведення обчислень.

У ході проведення експерименту для оцінювання можливості використання методами системних ресурсів було виміряно об'єм оперативної пам'яті, що потрібний для виконання завдань: виконання наборів окремих дій рефакторингу в коді певного обсягу; виконання однієї дії рефакторингу в значному за обсягом початковому коді. Графік залежності витрат об'єму пам'яті від кількості одиничних завдань рефакторингу зображено на рис. 2,*a*. Графік залежності витрат об'єму пам'яті від обсягу початкового коду зображено на рис. 2,*b*.

У ході проведення експерименту для оцінювання можливості використання методами системних ресурсів було також виміряно час, що потрібний методу для виконання завдань: виконання наборів окремих дій рефакторингу в коді певного обсягу; виконання однієї дії рефакторингу в значному за обсягом початковому коді.

Графік залежності витрат часу від кількості одиничних завдань рефакторингу зображено на рис. 3,*a*. Графік залежності витрат часу від обсягу початкового коду зображено на рис. 3,*b*.

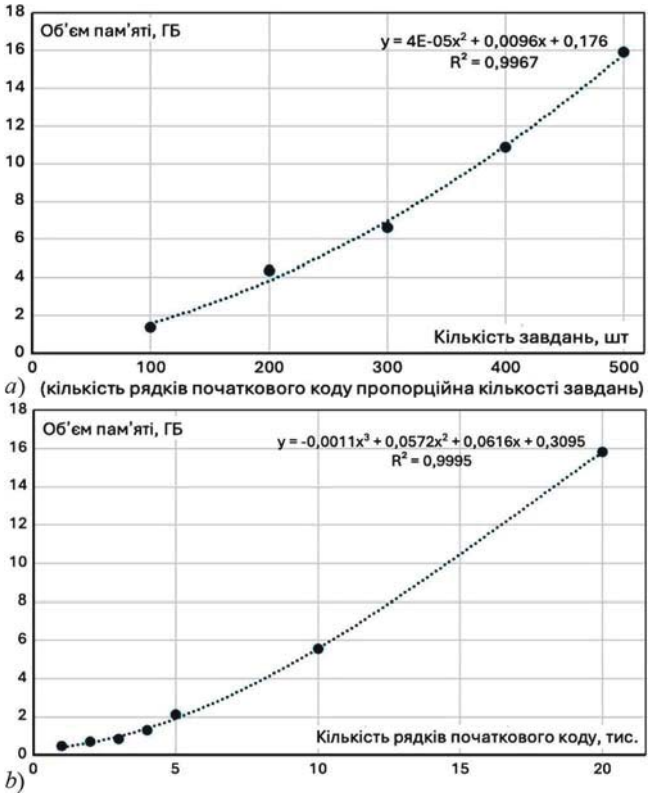

Рис. 2. Графік залежності об'єму оперативної пам'яті від / Plot depicting the dependency of the amount of RAM on: *a*) кількості завдань рефакторингу / the number of tasks for refactoring; *b*) розміру початкового коду / the size of the source code

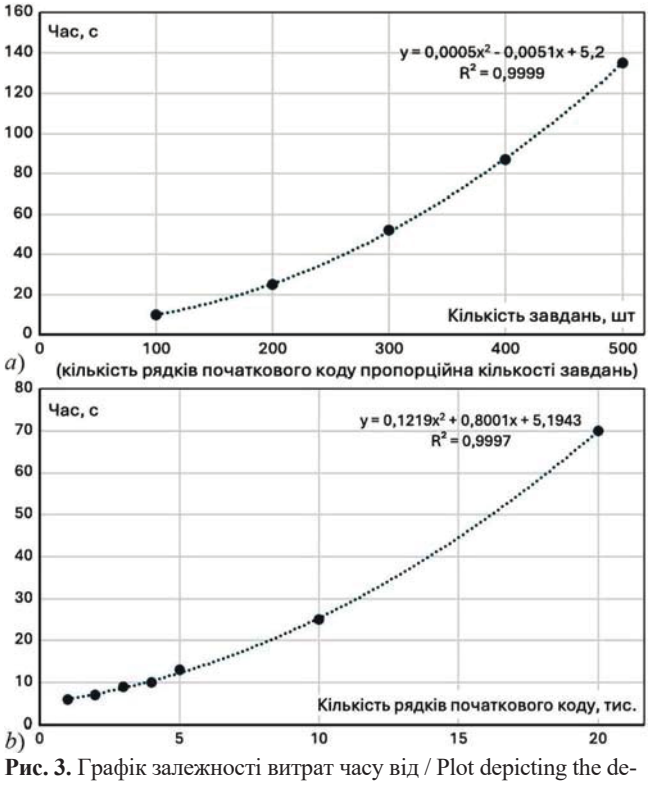

Рис. 3. Графік залежності витрат часу від / Plot depicting the dependency of time cost on: *a*) кількості завдань рефакторингу / the number of tasks for refactoring; *b*) розміру початкового коду / the size of the source code

У ході цих практичних експериментів було встановлено, що процеси логічного виведення у базі знань із значним обсягом фактів є ресурсномістким процесом – залежність необхідного обсягу оперативної пам'яті для виконання завдань від кількості завдань чи обсягу початкового коду, який обробляють, описують поліномами другого чи третього степеня відповідно. Щодо залежності часу роботи методу від кількості завдань чи обсягу початкового коду, який обробляють, описують поліномом другого степеня. Це означає, що процес вимагатиме значних ресурсів під час оброблення великих кодових баз (такі обсяги пам'яті, необхідної для роботи методу, є співставними із наявними рішеннями).

Для перевірки зручності у використанні підходів, трьом групам експертів було запропоновано здійснити кілька дій рефакторингу із використанням "класичного" (наявного та звичного їм) підходу до рефакторингу та із застосуванням розробленого прототипу.

Під час оброблення результатів експерименту було здійснено обчислення середніх абсолютних дисперсій для двох способів здійснення рефакторингу – класичним методом і з застосуванням запропонованих підходів. Отримані результати показали, що різниця в часі при застосуванні класичного підходу до рефакторингу у різних групах сильно варіюється (~122,23 хв), а при здійсненні рефакторингу із використанням пропонованих підходів затрачений час відрізняється менше (~48,89 хв), що можна пояснити певним спрощенням процесу, а також певною уніфікацією дій (рис. 4).

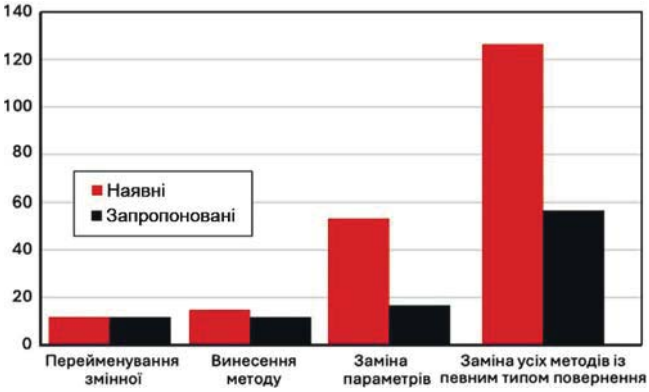

Рис. 4. Середні значення витрат часу для кожного із випадків рефакторингу / Average values of the time spent for each of the refactoring cases

Експеримент показав, що пришвидшення процесу рефакторингу з використанням прототипу у групах із різними вміннями і досвідом становить приблизно до 36 %. Це означає, що пропоновані методи однаково добре підходять як для розробників із малим досвідом роботи, так і для досвідчених.

Серед переваг запропонованого методу автоматизованого рефакторингу порівняно із наявними відзначено, що новий метод має рівноцінні можливості для всіх мов програмування. Це досягається завдяки реалізації правил і можливостей через базу знань, що не залежить від конкретної технології чи мови програмування. Також важливо відзначити, що усі правила, що додають в базу знань, є агностичними щодо мови програмування, тому їх потрібно додавати тільки один раз.

Запропоновані методи дають можливість одразу описати шаблони проектування чи дії, які потрібно виконати системою логічного виводу, що дає змогу застосовувати дії рефакторингу безпосередньо після додавання початкового коду в базу знань і без втручання розробника. Ці переваги також забезпечують стійкість

перетворення початкового коду, що гарантує успішне компілювання або інтерпретацію видозміненого початкового коду за допомогою програмного прототипу.

Запропоновані підходи відрізняється від наявних тим, що використовують засоби для роботи із семантикою для створення подання початкового коду і виконання рефакторингу шляхом модифікації цього подання. Розроблені методи та модель є універсальними і можуть використовуватись для будь-якої мови програмування, що підтримує об'єктно-орієнтовану парадигму.

Обговорення результатів дослідження. У роботі [8] запропоновано новий метод автоматизації семантичних завдань рефакторингу коду, що обумовлено спостереженням про потребу такого типу рефакторингу для модифікації подання даних абстрактного типу. Описаний метод приймає на вхід початкову реалізацію АТД (абстрактного типу даних), нове подання даних і так званий інваріант відносного подання (що пов'язує старі та нові подання даних) і автоматично генерує нову реалізацію АТД, яка семантично еквівалентна оригінальній версії. Запропонований у роботі метод базується на методі індуктивного синтезу керованого контр прикладами (CEGIS). Запропонований авторами підхід зводить базову задачу реляційного синтезу до набору спрощених задач програмування, використовує техніки символьного мислення і поняття часткової еквівалентності. Здійснена авторами оцінка показує, що метод може правильно виконати завдання рефакторингу у 97 % випадків.

Автори у свої роботи [5] зазначають, що рефакторинг програмного забезпечення може бути складним і затратним за часом процесом. Хоч такі інструменти рефакторингу допомагають розробникам створювати більш зрозумілий і підтримуваний код, більшість з них характеризуються довгими циклами зворотного зв'язку. Автори вважають, що можна досягнути зменшення часу зворотного зв'язку, акцентуючи увагу на концепції рефакторингу в реальному часі (англ. *Live Refactoring*) та її основних принципах: рекомендацій в реальному часі та постійній візуалізації "кандидатів" на рефакторинг, а також негайному відображенні результатів від застосування рефакторингу до коду. Для оцінювання запропонованого підходу автори провели емпіричний експеримент, результати якого показали, що підхід покращує декілька особливостей якості коду, і його було добре прийнято, зрозуміло і використано учасниками експерименту.

У роботі [11] презентується напів-автоматизований інструмент рефакторингу та вимірювання (англ. *Refactoring and Measurement Tool*, RMT), який виконує рефакторинг на підставі шаблонів у початковому коді, написаному мовою Java. RMT виявляє та застосовує шаблони проектування, використовуючи методи з літератури, та оцінює переваги застосування таких шаблонів через призму якісних атрибутів за допомогою метрик програмного забезпечення. Авторами було проведено п'ять тестів з Java-проектами.

Автори роботи [15] зазначають, що ефективне подання початкового коду є важливим для різних завдань інженерії програмного забезпечення. Більшість підходів до подання початкового коду все ще використовують AST і не використовують семантичні графи, такі як CFG (англ. *Control Flow Graph*, граф потоку керування) і PDG (англ. *Program Dependency Graph*, граф залежностей програми). У роботі зазначено, що одним з ефективних методів подання початкового коду є отримання шляхів з AST і використання навчальної моделі для розуміння властивостей програми. Ідея авторів роботи полягає в розширенні підходу на підставі шляхів code2vec так, щоб внести семантичні графи CFG і PDG з AST, що значною мірою не досліджувалося в інженерії програмного забезпечення. Порівняно з code2vec, запропонований авторами підхід покращує F1-оцінку на 11 % для повного набору даних і до 100 % для окремих проектів. Автори стверджують, що семантичні ознаки зі шляхів CFG і PDG дуже покращують результативність завдань інженерії програмного забезпечення.

У роботі [10] зазначається, що багато наявних систем рефакторингу використовують оригінальний початковий код як вхідні дані для операцій рефакторингу, чи то через еволюційні обчислення, чи через інші евристичні оператори. Однак зазначені оператори трансформації коду не завжди добре корелюють із гранулярністю початкового коду, на якому вони працюють. У цій роботи запропоновано статичне попереднє оброблення коду для його створення з більш однорідною гранулярністю, яка відкриває можливість процесу рефакторингу для відповідних компонент програми. У кожному розглянутому авторами випадку, застосування процедури попереднього оброблення перед виконанням основних дій рефакторингу, дає змогу інструменту вирішувати дефекти програмного забезпечення, які раніше були недосяжними для обраного інструменту рефакторингу. Отриманий авторами результат важливий, оскільки він застосовний до більшості методів вдосконалення програмного забезпечення на підставі пошуку та вирішує фундаментальну проблему відповідності гранулярності подання до гранулярності операторів.

У роботі [8] автори пропонують використовувати специфічне подання даних для подання початкового коду, що співзвучне із пропонованим поданням початкового коду у даному дослідженні. Однак, запропонована модель коду містить не тільки абстрактні типи даних (як це зроблено у роботі [10]), а весь початковий код з урахуванням його функціонального призначенням.

Автори дослідження [5] пропонують скоротити час зворотного зв'язку інструменту рефакторингу шляхом здійснення рефакторингу у режимі реального часу, що підкреслює важливість застосування таких засобів у щоденній роботі. У даному дослідженні поширеність і доцільність використання засобів рефакторингу досягається можливістю виправити усі запрограмовані правилами антишаблони автоматично за один прохід, що дає змогу враховувати повний задум розробника і не спрацьовує тоді, коли ще ідея не втілена повністю.

Доволі схожим за ідеєю є дослідження [11] щодо виконання рефакторингу на підставі шаблонів. Проте у ньому автори застосовують літературні методи, а в цьому дослідженні запропоновано використовувати опис дій рефакторингу і виправлення антишаблонів за допомогою логічних правил.

Автори дослідження [15] провели експеримент і стверджують, що семантичні ознаки покращують результативність виконання завдань програмної інженерії, що свідчить про те, що результати, отримані у даній роботи, є значущими.

У роботі [10] автори зазначають, що виконання попереднього оброблення початкового коду дає змогу до-

сягти якісно нових результатів. Це підкреслює значущість виконання рефакторингу із поданням початкового коду, а не оригінальним кодом, що власне і запропоновано у даному дослідженні.

Отже, за результатами виконаної роботи можна сформулювати такі наукову новизну та практичну значущість результатів дослідження.

Наукова новизна отриманих результатів дослідження – вперше розроблено модель бази знань для подання початкового коду, який будують на підставі онтології і використовує засоби OWL для подання семантики початкового коду, розроблено метод оброблення запропонованої моделі на підставі логічних правил для формалізації кроків рефакторингу і можливості їх виконання саме на рівні подання, а не на рівні синтаксису початкового коду, удосконалено метод автоматизації рефакторингу початкового коду, який відрізняється від наявних тим, що використовує засоби для роботи із семантикою і базу знань із фактами про початковий код для виконання дій рефакторингу.

Практична значущість результатів дослідження – застосовано для розроблення програмного прототипу для здійснення рефакторингу початкового коду, написаного мовою програмування Swift, їх також можна використати для розроблення засобів рефакторингу для інших мов програмування (оскільки розроблені методи та модель є універсальними), для пошуку семантичних дублікатів, для автоматизованого написання документації.

Висновки / Conclusions

Розроблено методи автоматизованого рефакторингу, які використовують базу знань і логічні правила, що дасть змогу виконати складний рефакторинг, описаний послідовністю дій у правилах, виконуючи прості операції (спрощення яких проходить саме через оброблення коду шляхом зміни подання цього коду, що зберігає семантику). За результатами дослідження можна зробити такі основні висновки.

1. Проаналізовано публікації останніх років, описані в них досягнення у сфері виконання рефакторингу на підставі семантики. Зроблено висновок про доцільність розроблення таких засобів.
2. Розроблено метод автоматизації рефакторингу на підставі знань, метод оброблення моделі подання коду на підставі правил, а також саму модель початкового коду, що будують на підставі онтології.
3. Запропоновано використовувати системи логічного виведення для оброблення подання початкового коду, що виконане з використанням семантичних засобів, для здійснення рефакторингу, описаного правилами логічного виведення.
4. З'ясовано, що правильне відображення стану кодової бази аксіомами бази знань дасть змогу виявити неявні або невідомі відношення та аксіоми і додати їх до бази знань, водночас роблячи її більш повною, додавати правила до бази знань (наприклад, шаблони, що описують антипатерни, та підходи до їх усунення), щоб видаляти, змінювати або генерувати код, який буде компілюватися або інтерпретуватися.
5. Наголошено на конкурентній перевазі запропонованих підходів у частині можливості їх перевикористання для будь-яких мов програмування. Експериментально встановлено, що залежність витрат системних ресурсів і часу відносно характеристик вхідних даних (початкового коду і завдань рефакторингу) є поліноміальною. Проведено дослідження ефективності застосування запропонованих підходів для здійснення рефакторингу серед різних груп розробників і показано, що витрата часу зменшується на 36 %.
6. Доведено ефективність застосування семантичного підходу для здійснення автоматизованого рефакторингу. Зауважено можливість подальшого розвитку, як-от інтеграція запропонованих методів із раніше розробленими для отримання якісно нових результатів.

References

1. Aguiar, C. Z., Zanetti, F., & Souza, V. E. (2021). Source Code Interoperability based on Ontology. *VII Brazilian Symposium on Information Systems* (pp. 1–8). ACM. https://doi.org/10.1145/3466933.3466951
2. Atzeni, M., & Atzori, M. (2017). CodeOntology: RDF-ization of Source Code. *The Semantic Web – ISWC*, (pp. 20–28). https://doi.org/10.1007/978-3-319-68204-4_2
3. Berners-Lee, T., Hendler, J., & Lassila, O. (2001) The Semantic Web: A New Form of Web Content That is Meaningful to Computers Will Unleash a Revolution of New Possibilities. Linking the World's Information: Essays on Tim Berners-Lee's Invention of the World Wide Web (1st ed.). Association for Computing Machinery, New York, NY, USA, 91–103. https://doi.org/10.1145/3591366.3591376
4. Fernandes, E., Uchôa, A., Bibiano, A. C., & Garcia, A. (2019). On the Alternatives for Composing Batch Refactoring. 2019 IEEE/ACM 3rd *International Workshop on Refactoring* (IWoR), Montreal, QC, Canada, pp. 9-12. https://doi.org/10.1109/IWoR.2019.00009
5. Fernandes, S., Aguiar, A., & Restivo, A. (2023). LiveRef: a Tool for Live Refactoring Java Code. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (ASE '22). Association for Computing Machinery, New York, NY, USA, Article 161, 1–4. https://doi.org/10.1145/3551349.3559532
6. Ganapathy, G., & Sagayaraj, S. (2011). To Generate the Ontology from Java Source Code. *International Journal of Advanced Computer Science and Applications, 2*(2), 111–116. https://doi.org/10.14569/IJACSA.2011.020218
7. Kesseli, P. (2017). Semantic refactorings [PhD thesis]. University of Oxford. URL: https://ora.ox.ac.uk/objects/uuid:0c74954e-dc83-463f-bcd4-519d98c3dcca
8. Pailoor, S., Wang, Y., & Dillig, I. (2024). Semantic Code Refactoring for Abstract Data Types. Proc. ACM Program. Lang. 8, POPL, Article 28, 32 pages. https://doi.org/10.1145/3632870
9. Paydar, S., & Kahani, M. (2012). A Semantic Web based approach for design pattern detection from source code. *2012 2nd International eConference on Computer and Knowledge Engineering* (*ICCKE*), Mashhad, Iran, pp. 289–294. https://doi.org/10.1109/ICCKE.2012.6395394
10. Reiter, P., Espinoza, A. M., Doupé, A., Wang, R., Weimer. W., & Forrest, S. (2022). Improving source-code representations to enhance search-based software repair. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO '22). Association for Computing Machinery, New York, NY, USA, 1336–1344. https://doi.org/10.1145/3512290.3528864
11. Riemer, Y., Pedroso Nogueira, P. M., Matos, S. N., & Bronoski Borges, H. (2023). RMT: A Semi-Automated Tool for Refactoring Design Patterns. In: *Proceedings of the XXXVII Brazilian Symposium on Software Engineering* (SBES '23). Association for Computing Machinery, New York, NY, USA, 77–82. https://doi.org/10.1145/3613372.3613416
12. Tkachuk, A., & Bulakh, B. (2022). Research of possibilities of default refactoring actions in Swift language. *Technology Audit and Production Reserves, 5*(2(67)), 6–10. https://doi.org/10.15587/2706-5448.2022.266061
13. Tkachuk, A., & Bulakh, B. (2022). Usage of formalized knowledge about source code for refactoring actions in Swift. *Technology Audit and Production Reserves, 6*(2(68)), 6–10. https://doi.org/10.15587/2706-5448.2022.267160
14. Tkachuk, A., & Bulakh, B. (2023). Describing the Knowledge About the Source Code Using an Ontology. *Infocommunication*

and Computer Technology, *1*(5), 123–133. https://doi.org/10.36994/2788-5518-2023-01-05-14
15. Vagavolu, D., Swarna, K. C., & Chimalakonda, S. (2022). A mocktail of source code representations. In Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE '21). IEEE Press, 1296–1300. https://doi.org/10.1109/ASE51524.2021.9678551

A. V. Tkachuk

National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine

AUTOMATED CODE REFACTORING USING A KNOWLEDGE BASE AND LOGICAL RULES

The task of automating the process of replacing the source code without changing the program's behavior (i.e., refactoring) is considered. This is proposed to be achieved by using a knowledge base about the source code and logical rules to describe the code transformation process. The relevance of developing new methods and approaches to refactoring (considering the limitations of existing systems and the increasing needs of end-users in automating refactoring processes due to the growth of code bases) is outlined. The importance of preserving the semantics of the original source code for the successful execution of refactoring is emphasized. It is proposed to use ontologies and tools for semantic processing (including those provided by the Web Ontology Language (OWL)) for the formal representation of the source code of a software product in the knowledge base. A method of automated refactoring based on knowledge of the source code, a model for representing the source code based on an ontology, and a method for processing the model based on rules that relying on the semantics of the source code during refactoring is proposed. The method of automated refactoring consists of two transformations and processing of an intermediate source code representation. The described model for representing the source code consists of two parts, which are constructed based on the ontology and axioms added to the model after syntactic analysis of the source code. The method for processing the model based on rules uses two types of rules for logical inference, which describe additional functional dependencies between entities in the knowledge base. The proposed methods and models are analyzed, and their practical application is discussed. A series of tests was conducted to verify the theoretical foundations of the proposed models and methods in practice. It is stated that it is expedient to implement the refactoring software toolkit using a client-server architecture to simplify the abstraction of the knowledge base about the code from the specific code base. It is experimentally established that the required amount of operating memory for task execution depends on the volume of the processed source code polynomially. A speedup of up to 36 % in the refactoring process using the developed software prototype compared to existing tools is demonstrated.

Keywords: software; source code; ontology; model; formalized knowledge representation.