ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)
Наука та прогрес транспорту, 2026, № 1 (113)

УДК 004.415.2:004.89

Д. О. КАРПОВСЬКИЙ[1*]

[1*]Каф. «Комп'ютерні науки», Український державний університет науки і технологій, ННІ ДІІТ, вул. Лазаряна, 2, Дніпро, Україна, 49010, тел. +38 (056) 733 19 61, ел. пошта karpovskyi.d@gmail.com, ORCID 0009-0002-0020-5148

Методи та засоби рефакторингу онтологій

Мета. Робота спрямована на дослідження еволюції концепції рефакторингу – від інструмента оптимізації програмного коду до потужного засобу вдосконалення структур даних, алгоритмів та бізнес-процесів. Основною метою дослідження є вивчення історичного процесу становлення рефакторингу програмного забезпечення та онтологій, можливостей застосування методів покращення до онтологій як специфічного виду програмних структур, що відіграють ключову роль у знання-орієнтованих системах, семантичному вебі та кіберфізичних системах нового покоління. **Методика.** Методологічну основу дослідження становлять загальнонаукові та спеціальні методи аналізу й узагальнення. Застосовано систематичний огляд наукових публікацій у галузі рефакторингу програмного забезпечення та онтологічної інженерії з використанням повнотекстових і реферативних наукометричних баз даних. У роботі використано методи порівняльного аналізу для зіставлення різних підходів до рефакторингу, структурно-функціональний аналіз для дослідження змін у програмних та онтологічних моделях, а також методи класифікації для групування існуючих технік рефакторингу за рівнем абстракції та сферою застосування. **Результати.** У межах дослідження проведено ґрунтовний аналіз наукових публікацій, присвячених тематиці рефакторингу та онтологічної інженерії, із залученням повнотекстових і реферативних баз даних. Розглянуто історію становлення підходів до рефакторингу, зокрема трансформації схем даних, концептуального рефакторингу, модифікації обмежень цілісності та еволюції інструментів для оцінки якості змін. Особливу увагу приділено застосуванню рефакторингу до онтологій, включно з аналізом його впливу на модульність, когерентність та повторне використання знань. **Наукова новизна** роботи полягає в систематичному дослідженні існуючих методів рефакторингу та їх класифікації за типами. Здійснено комплексний аналіз сильних і слабких сторін кожного з підходів, що дозволяє обґрунтовано обирати оптимальні стратегії вдосконалення програмного коду. Запропонований підхід сприяє глибшому розумінню механізмів рефакторингу в контексті різних сфер застосування. **Практична значимість** Представлені результати можуть бути використані під час проєктування, супроводу та еволюції знання-орієнтованих систем, де важливу роль відіграє структурна узгодженість і підтримка семантики даних. Окреслено перспективи подальших досліджень, що включають автоматизацію процесів рефакторингу онтологій за допомогою методів машинного навчання, розширення системи метрик оцінювання та адаптацію запропонованих рішень до предметних галузей, які характеризуються високою динамікою змін.

Ключові слова: інформаційні технології; онтологія; рефакторинг; бази знань; методи рефакторингу; програмне забезпечення; трансформація; якість

Вступ

Наразі створено велику кількість онтологій, які описують цілий ряд предметних областей, винайдено нові методи та розроблені застосунки, які дозволяють шукати потрібні онтології та обробляти їх. Можливо найбільш використовуваний застосунок для пошуку онтологій – це Swoogle [18], веб орієнтований інструмент, який дозволяє знайти бажану онтологію серед десятків тисяч існуючих.

Попри наявність сучасних технологій пошуку, що забезпечують широкий спектр інструментів для виявлення онтологій, які відповідають заданим критеріям, процес ідентифікації відповідної онтології все ще становить певну складність [20]. Навіть якщо пощастило, та бажана онтологія була знайдена, то існує велика вірогідність зіштовхнутись із новим викликом – як адаптувати онтологію так, щоб її можна було ефективно інтегрувати в конкретну бізнес модель [14]. Насправді, коли знайдено онтологію, то її необхідно реструктурувати, щоб зробити більш придатною для використання (за допомогою використання існуючих методів рефакторингу онтологій), та адаптувати до тієї концепції та бізнес термінології, яку використовує кінцевий користувач [15].

doi: https://doi.org/10.15802/stp2026/356061

Операції еволюціонування відігравали і відіграють важливу роль в розробці програмного забезпечення [22]. Деякі із цих операцій спрямовані на зміну програми, або концептуальної схеми та можуть використовуватись для додавання, видалення, або зміни певних функціональних можливостей системи без зміни її поведінки [30]. В контексті даних наукових досліджень такі операції називають «операції рефакторингу» [22]

Дослідники даної предметної області намагались використовувати операції рефакторингу не тільки в програмному забезпеченні, а і в елементах із більш високим рівнем абстракції: базах даних, UML моделях, Object Constraint Language (OCL) моделях і в набагато меншій мірі в онтологіях [5].

Логічно постає питання, чому до онтологій методи рефакторингу менше застосовувались на практиці. Двома основними недоліками застосування рефакторингу до онтологій була відсутність однозначного та узгодженого каталогу операцій рефакторингу, які мають практичний сенс [15]. Такі загально узгоджені операції присутні в інших сферах, таких як рефакторинг програмного забезпечення та баз даних [4].

Наслідком цього стало те, що дослідники намагались застосувати вже існуючи операції рефакторингу [16] до онтологій. Основними концепціями, які із цього вийшли стали «реструктуризація програми» та «трансформація схеми» [3].

Мета

Метою даної статті є комплексне дослідження еволюції підходів до рефакторингу в програмній інженерії та онтологічній інженерії як взаємопов’язаних напрямів розвитку сучасних знаннє-орієнтованих інформаційних систем. У роботі поставлено завдання проаналізувати історичні передумови виникнення рефакторингу, його трансформацію від локальних змін програмного коду до концептуального рівня, що охоплює структури даних, архітектурні рішення та онтологічні моделі. Особливу увагу приділено виявленню можливостей і обмежень застосування класичних та сучасних методів рефакторингу до онтологій як специфічного виду програмних структур, що забезпечують формалізацію знань у різноманітних системах. Додатковою метою дослідження є формування теоретичного підґрунтя для подальшої автоматизації процесів рефакторингу онтологій із використанням методів штучного інтелекту та машинного навчання, а також визначення критеріїв оцінювання ефективності таких змін з погляду якості, масштабованості та повторного використання знань.

Методика

Методологічну основу дослідження становить поєднання загальнонаукових і спеціалізованих методів аналізу, що забезпечує комплексний розгляд процесів рефакторингу програмного забезпечення та онтологій. У роботі застосовано систематичний аналіз наукових публікацій з питань рефакторингу, онтологічної інженерії та знаннє-орієнтованих систем із використанням повнотекстових і реферативних наукометричних баз даних, а також історико-аналітичний і порівняльний методи для дослідження еволюції підходів та зіставлення класичних і сучасних технік рефакторингу.

Для аналізу змін в онтологічних моделях використано структурно-функціональний аналіз і методи класифікації, що дозволило оцінити вплив рефакторингу на модульність, зв’язність, зчеплення та когерентність онтологій. Ефективність підходів оцінювалася на основі якісних і кількісних показників структурної узгодженості, підтримки семантики та можливостей повторного використання знань, що створює методологічне підґрунтя для подальшої автоматизації процесів рефакторингу онтологій із використанням методів штучного інтелекту.

Аналіз сучасних досліджень і публікацій.

Становлення методів рефакторингу. Останніми десятиліттями схожі за своїми цілями операції були досліджені різними науковцями та детерміновані різними назвами: операції рефакторингу, операції реструктуризації, операції трансформації програм, операцій трансформації моделей [9]. Розглянемо в хронологічному порядку які техніки заховані за попередньо згаданими термінами, та які із них є найбільш ефективними для реструктуризації онтологій.

doi: https://doi.org/10.15802/stp2026/356061

Операції реструктуризації програм. Реструктуризація програм – методика яка була винайдена у 80х роках 20 століття [12]. Призначення даної методики – реструктуризація програм без зміни їх поведінки для кінцевого користувача [24]. Дана техніка відрізняється від загальноприйнятого у сьогоденні підходу рефакторингу тим, що вона має справу лише із необ'єктно-орієнтованою архітектурою програм.

В 1993 році дослідники Грізволд та Ноткін презентували у своїй науковій роботі певну кількість реструктуризаційних методів [24] для покращення якості програмного коду та підтримки програми в майбутньому. Дані методи надавали можливість змінювати назви змінних, змінювати вирази змінними і навпаки змінні виразами, замінювати деяку послідовність викликів команд окремим методом, який включав в себе даний набір команд. В даній роботі також було визначено яким чином можна автоматизувати виконання даних операцій [23].

Також в 1998 році цими ж авторами було видано роботу, в якій було визначено підхід (фреймворк), який підтримував користувача у визначенні місць для застосування змін та використанні конкретних методів реструктуризації для великих програм.

Існують додаткові методи реструктуризації програм, але вони не можуть бути застосовані до онтологій через певні обмеження. У роботі Бергстейна 1991 року [8] визначено методи реструктуризації, але обов'язковою умовою є те, що всі неабстрактні класи двох схем мають однакові назви та атрибути, включаючи успадковані.

Операції трансформації схем. Операції трансформації схем активно вивчаються вже протягом останніх 20 років [26]. Дана методика оперує схемами як такими. На вхід подається початкова схема, потім до неї застосовуються операції трансформації і на виході ми маємо нову покращену схему. Такі трансформації можуть бути класифіковані за принципом впливу на інформацію, яку схема містить в собі:

– перетворення зі збереженням – виконання операцій зберігає інформацію, яку містить в собі початкова схема [25]

– перетворення зі змінами – виконання операцій модифікує інформацію, яку містить в собі початкова схема [6]. Надалі такі операції можна класифікувати таким чином:

– перетворення із розширенням – результуюча схема містить додаткову інформацію у порівнянні з початковою;

– перетворення із скороченням – результуюча схема містить лише необхідні дані;

– непорівняні перетворення – жодна із попередніх категорій невірна.

Операції перетворення схем, які належать до першої групи, до них можна віднести «Перейменування елементів», «Виділення підкласу або суперкласу», «Злиття або поділ класів/таблиць», «Зміна типів властивостей», «Видалення надлишкових елементів», можна розглядати як рефакторинг операцій [3], оскільки вони зберігають семантику модифікованої схеми, тобто інформацію, яку вона містить. Використовуючи термін «операції перетворення схем» ми маємо на увазі операції перетворення зі збереженням інформації [26].

Зазначити, чи визначають дві концептуальні схеми однакову інформацію (тобто вони еквівалентні) дуже важко. Кілька авторів вивчали цю тему та представили різні пропозиції щодо еквівалентності на основі: наборів даних, логіки, контекстуальної еквівалентності, замінної еквівалентності тощо [25].

Таким чином, операція перетворення схеми може перетворити схему S1 в іншу схему S2 і скасувати зміни шляхом перетворення схеми S2 на схему S1 [26]. Батіні у своїй роботі 1992 року визначив набір операцій перетворення схеми зі збереженням інформації, які покращували читабельність і стислість концептуальних схем [6]. До цих операцій відноситься:

– видалення зайвих циклів зв'язків: ця операція видаляє зв'язки, що складаються з типів лінкування, які вже визначені іншими типами відносин;

– додавання або видалення похідних атрибутів/підмножин: ця операція додає або видаляє похідні атрибути або класи;

– усунення наслідуваних сутностей в ієрархіях узагальнення: ця операція об'єднує сутність з її підтипами; використання методу дійсне лише тоді, коли підтипи порожні;

– усунення підвішених елементів: сутність поглинає іншу сутність, пов'язану з нею відносинами типу;

– створення узагальнення: ця операція створює узагальнення та спеціалізує зв'язки між сутностями зі спільними атрибутами;

doi: https://doi.org/10.15802/stp2026/356061

– створення узагальнення: ця операція створює зв'язки узагальнення та спеціалізації між сутностями зі спільними атрибута.

Ейк в своїй роботі 1991 року представив декілька операцій трансформації схеми, які дозволяють [20]:

– перемістити атрибути між типами сутностей концептуальної схеми, пов'язаними типами зв'язків і відношеннями узагальнення або спеціалізації;

– розділити клас навпіл і зв'язати два отримані класи за допомогою типу відношення;

– створити новий підтип або супертип існуючого типу сутності;

– перемістити кінців типів зв'язків через таксономію типів сутностей.

Також Ейком було складено на той час найбільш практичну схему збереження інформації під час використання операцій рефакторингу [20]. Також автором було досліджено, як використовувати згадані вище операції задля покращення якості концептуальної схеми. Вісім із дев'яти представлених операцій зосереджені на типах зв'язків реструктуризації, і лише один фокусується на типах сутностей реструктуризації. Автор згадує в статті, що представлений перелік операцій є неповним і було б корисно доповнити його новими операціями.

Рефакторинг ПЗ та його еволюція. Концепція рефакторингу народилася на початку 1980-х років, коли Опдайк у роботі 1992 року [30] визначив рефакторинг як «перетворення програми, яке реорганізує програму без зміни її поведінки».

До Опдайка кілька авторів представили ідею створення набору операцій модифікації для об'єктно-орієнтованих програм, які зберігають їх поведінку. Існують певні алгоритми та евристики, які забезпечують хорошу організацію класів [12].

Такі приклади включає робота Бергштейна 1991 року [8], де він запропонував набір операцій перетворення класу, які зберігають поведінку об'єктно-орієнтованої схеми з певними структурними обмеженнями. Також такі приклади є в роботі Казаєса 1989 року [12], де він представив набір методів для реструктуризації успадкування діаграми класів без втрати функціональності.

У своїй дисертації 1992 року Опдайк визначив 23 типи примітивних методів рефакторингу та показав, як їх компонувати, щоб створити більш складні операції рефакторингу. Для кожної операції рефакторингу автор визначив передумову, яка гарантує, що поведінка програми не зміниться після виконання операції. Опдайк визначив операції рефакторингу програмного забезпечення в термінах C++ [30].

Пізніше Робертс в 1994 році трошки видозмінив визначення Опдайком поняття рефакторингу, щоб мати можливість визначити нові методи рефакторингу, які не зберігають поведінку програми, але можуть бути корисними для дизайнерів [31]. Робертс також розширив визначення рефакторингу пост умовами, щоб спростити створення складних операцій рефакторингу шляхом упорядкування більш простих і зменшити кількість аналізу, необхідного для визначення того, які складні ланцюжки рефакторингу можна застосувати.

Проте вважається, що найкращим визначенням рефакторингу програмного забезпечення є те, яке надав Фаулер в 1999 році [22]:

– рефакторинг (іменник): зміна внутрішньої структури програмного забезпечення, щоб зробити її легшою для розуміння та дешевшою для модифікації без зміни її спостережуваної поведінки;

– рефакторинг (дієслово): реструктуризувати програмне забезпечення шляхом застосування серії рефакторингів без зміни його спостережуваної поведінки.

Відповідно до Філіпса та Румпе (2001) [19], визначення зовнішньої або спостережуваної поведінки даної програми залежить від її функціональних вимог. Наприклад, в інформаційних системах реального часу час виконання вважається функціональною вимогою. Таким чином, у цьому випадку багато типових операцій рефакторингу можуть бути незастосовними, оскільки їхні зміни можуть негативно вплинути на час виконання і, отже, порушити функціональні вимоги, а отже, змінити спостережувану поведінку.

Фаулер (1999) у своїй книзі визначає 72 типи рефакторингу та коротко підсумовує кожну з них, пояснюючи для кожної операції, за яких умов вона корисна, етапи її виконання та перевірки, які слід виконати після її виконання, щоб гарантувати збереження поведінки програми [22]. Фаулер також визначив ознаки поганого коду (далі ОПК) які є умовами, що позначають

doi: https://doi.org/10.15802/stp2026/356061

певні структури в коді, які передбачають (і іноді вимагають) можливість рефакторингу. З кожним ОПК Фаулер пов'язує набір операцій рефакторингу, які можна застосувати для покращення програми.

Хоча рефакторинг був розроблений для застосування вручну, деякі інструменти були розроблені для його автоматичного виконання, такі як Refactoring Browser, RefactorIT5, Together ControlCenter, тощо [31].

Незважаючи на код ОПК, знати, коли і де застосовувати операції рефакторингу, все ще є великою проблемою. Виявлення того, які операції рефакторингу можна ефективно застосувати, і де знаходиться те, що називається «виявленням можливостей рефакторингу» [7]. При виявленні можливостей рефакторингу слід відповісти на такі запитання:

– які операції рефакторингу можна ефективно застосувати: необхідно отримати операції рефакторингу, виконання яких покращить програму;

– де їх можна застосувати: в яких частинах програми можна ефективно застосувати попередні операції рефакторингу;

– у якому порядку їх слід застосовувати: операції рефакторингу мають бути послідовними, якщо кілька операцій рефакторингу можуть бути ефективно застосовані одночасно. Зробити висновок, у якому порядку їх слід застосовувати, нелегко, оскільки це залежатиме від фактору якості, який ми хочемо покращити, і застосування однієї операції може означати, що інша операція більше не може бути застосована, і тому її слід відкинути;

– які операції рефакторингу не слід застосовувати: оскільки операції рефакторингу є односпрямованими, є операції, які скасовують зміни, внесені іншими операціями. Це слід враховувати, щоб уникнути циклів у виконанні операцій рефакторингу [29].

Щоб повністю автоматизувати процес рефакторингу, необхідно:

– визначити, які операції можна застосувати і до якої частини схеми;

– виконати їх автоматично.

Виявлення можливостей рефакторингу є недетермінованим процесом, і для визначення найкращої операції рефакторингу для кожного випадку потрібно багато семантичної інформації. Отже, це виявлення можливе лише в деяких випадках, і навіть у таких випадках його автоматизація не є тривіальною [27].

З усіх цих причин процес рефакторингу не може бути повністю автоматизований і потребує втручання дизайнера/програміста для його успішного застосування. Найдосконаліші інструменти виявляють деякі можливості рефакторингу та використовують їх, щоб запропонувати рефакторинг, але, як правило, програміст на останньому кроці повинен вручну вибрати, які операції рефакторингу застосувати та обрати порядок їх застосування [29].

Рефакторинг обмежень цілісності. Важливою частиною концептуальної схеми є її обмеження цілісності. Обмеження цілісності визначають умови, яким мають задовольняти екземпляри схеми, щоб бути коректними. Обмеження цілісності були найбільш забутим елементом у рефакторингу концептуальних схем. Корреа і Вернер першими запропонували обмеження цілісності рефакторингу в 2004 році [16]. Зокрема, їх робота стосується обмежень цілісності програм, написаних на мові OCL. Автори адаптували кілька класичних операцій рефакторингу програмного забезпечення до OCL і представили набір OCL ознак поганого коду, які потенційно можуть бути в програмі. Ознаки поганого коду OCL еквівалентні ОПК Фаулера: правила, які означають, що певне обмеження OCL можна покращити за допомогою рефакторингу.

Ознаки поганого коду OCL, представлені в цій статті, це:

– магічний літерал, який визначає, коли обмеження цілісності використовує літерал у джерелі обмеження;

– «I» ланцюг, який визначає, коли обмеження складається з двох або більше додаткових обмежень, пов'язаних оператором «and»;

– тривалий шлях – виявляє, коли вираз OCL проходить через велику кількість асоціацій.

– Корреа і Вернер також визначили чотири операції, які мають справу з вищезгаданими ознаками поганого коду OCL [16]:

– додати змінну до виразу: додає оголошення змінної до виразу OCL;

– замінити вираз змінною: частина виразу OCL замінюється змінною, яка пояснює його вміст;

doi: https://doi.org/10.15802/stp2026/356061

– вираз «Розділення І»: розділяє обмеження цілісності, що складається з двох або більше обмежень, з'єднаних між собою;

– додати визначення операції з виразу та замінити вираз на вираз виклику операції: ці операції стосуються і мають ту саму функцію, що й перша та друга операції, наведені вище.

Як згадувалося вище, лише частина рефакторингу програмного забезпечення адаптується до OCL, щоб покращити якість обмежень OCL. Список ОПК і операцій OCL є неповним, оскільки існують інші операції рефакторингу програмного забезпечення, адаптація яких до OCL може бути дуже корисною, наприклад операції «Замінити алгоритм» і «Видалити подвійний негатив». Перший замінює фрагмент вихідного коду іншим кодом із тим самим значенням; останнє усуває подвійні негативи, які використовуються у вихідному фрагменті, тим самим значно покращуючи його читабельність [22].

Рефакторинг та його якість. Операції рефакторингу програмного забезпечення реструктуризують програми з метою покращення їх якості. Якість програмного забезпечення відносна і залежить від точки зору, з якої розглядається програма. Наприклад, для однієї людини якісною програмою є та, яка виконує свої завдання максимально швидко, а для іншої якісну програму легко зрозуміти та змінити [13].

Операції рефакторингу не впливають однаково на всі показники якості. Операція рефакторингу, яка покращує спільне використання та читабельність коду, може погіршити часову ефективність; або операція, яка покращує час виконання, може погіршити читабельність [37]. Як наслідок, існує значна залежність між факторами якості, які ми хочемо покращити, операціями рефакторингу, які ми маємо застосувати, і операціями, яких ми маємо уникати, щоб покращити бажані фактори якості [31].

Зв'язок між операціями рефакторингу та факторами якості не вивчався, доки рефакторинг не досяг певної зрілості. Тим не менш, сьогодні якість є однією з найпоширеніших тем для при дослідженні рефакторингу [28].

Саймон та ін. були одними з перших дослідників, які враховували показники якості в процесі рефакторингу (Саймон, Штейнбуккнер, 2001). Вони визначили структуру, яка використовує об'єктно-орієнтовані показники для виявлення певних можливостей рефакторингу для операцій «Move attribute», «Move method», «Extract class» і «Inline class». Зокрема, вони використовують метрики згуртованості на основі відстані, щоб визначити, коли можна застосувати рефакторинг [36].

Дю Буа в 2004 році [10] запропонував техніку переписування постумов операцій рефакторингу, щоб автоматично зробити висновок про те, як виконання операції рефакторингу вплине на якість програмного забезпечення з точки зору бажаних показників.

Дю Буа та Демейєр в свої роботі адаптували цю техніку для створення набору стратегій рефакторингу, які слід враховувати в процесі виявлення можливостей рефакторингу. Ці стратегії були створені для покращення згуртованості та зв'язку системи, яка підлягає рефакторингу. Ідея Дюбуа багатообіцяюча, але вона стосується кількох операцій рефакторингу (зокрема п'яти) і визначає лише шість стратегій, які не охоплюють усі можливі випадки [11].

Роботи інших авторів зосереджені на тому, як виявити можливості рефакторингу та виконати пов'язані з ними операції з урахуванням нефункціональних вимог системи, що рефакторингується.

У цьому контексті Мілопус та Ю представили структуру, в якій розробник використовував графи Schema integration graph (SIG) для ідентифікації операцій рефакторингу, які можна застосувати до програми, щоб задовольнити нефункціональні вимоги виражений у SIG. SIG – це графіки, які показують залежності між цілями, підзавданнями, ресурсами тощо [13].

Автори також представили чотириетапний алгоритм для ідентифікації та виконання операцій рефакторингу. Ця техніка повністю ручна. Крім того, його важко застосувати через дві проблеми: 1) незрозуміло, які операції рефакторингу можуть покращити певний аспект програмного забезпечення, і 2) незрозуміло, скільки операцій необхідно для досягнення заданої цілі.

Крім того, в загальному випадку збіжність процесу не гарантується, оскільки можуть існувати операції, які покращують і погіршують протилежні характеристики. У таких випадках застосування операції рефакторингу може задовольнити ціль g1, але порушити іншу g2, а застосування іншої операції рефакторингу для досягнення мети g2 може погіршитися

doi: https://doi.org/10.15802/stp2026/356061

і, отже, порушити умову g1 (яка раніше була задоволена). У такому випадку алгоритм не сходиться.

Рефакторинг онтологій. Дотепер рефакторинг застосовувався до широкого діапазону джерел і артефактів: програмного забезпечення, моделей UML, варіантів використання тощо [35]. Пропонується визначення рефакторинга, яке, можливо, найкраще визначає його значення незалежно від артефакту, який потрібно рефакторингувати: рефакторинг – це процес переписування письмового матеріалу для покращення його читабельності чи структури з явною метою збереження його значення чи поведінки [15].

Всі артефакти, які були перероблені до теперішнього часу слідують граматиці і тому можуть бути виражені як текстове представлення, що робить вищенаведене визначення застосовним у всіх випадках. Однак кожен артефакт, як правило, має своє власне визначення рефакторингу, яке чітко визначає значення і поведінку в його контексті.

Наприклад, рефакторинг програмного забезпечення визначається як процес реструктуризації програмного забезпечення, із метою зробити його більш зрозумілим і легше підтримуваним без зміни його поведінки. Оскільки мета всіх програмних систем полягає в тому, щоб запропонувати заздалегідь визначену поведінку – тобто завжди повертати однакові виходи для певного набору входів – відповідність між двома рефакторинговими визначеннями є чіткою [22].

У контексті бази даних Амблер перевизначив рефакторинг [1] як процес зміни схеми бази даних, що покращує її дизайн, зберігаючи при цьому як її поведінкову, так і інформаційну семантику. Це визначення не дуже точне, але, як і попереднє, воно визначає якісні фактори, які необхідно покращити (дизайн), а також визначає поведінку бази даних, що характеризується як інформаційна семантика (виражена та сама інформаційна база) та поведінкова семантика (тригери, збережені процедури тощо).

Онтології не є ні програмами, ні базами даних. Тому їх рефакторингове визначення слід уточнити, як і в вищезгаданих випадках [2]. Рівень абстракції онтології вище, ніж у схеми бази даних або програми. Тому всі якісні фактори, пов’язані з конкретною реалізацією концептуалізації або представленням доменної популяції, не мають відношення до якості онтології, оскільки її цілі по суті є семантичними. Таким чином, можемо визначити онтологічний рефакторинг як: процес реструктуризації онтологій для поліпшення їх читабельності або структури, зберігаючи при цьому відповідні знання [33].

Онтології призначені для задоволення множини вимог [32]. Рефакторинг онтології має зберігати конкретні знання, якщо ці знання, необхідні для задоволення її вимог. Іншими словами, рефакторована онтологія повинна бути здатна вивести ті ж знання, що і оригінальна онтологія [34]. Тому метою онтологічного рефакторингу є не зміна асоційованої концептуалізації (концептуальної зміни), а спосіб представлення асоційованої концептуалізації (репрезентативної зміни).

Одним з найскладніших завдань при виконанні рефакторингових операцій є зміна вихідного коду, який відноситься до рефакторингових елементів [22]. Наприклад, якщо ми виконуємо рефакторинг операції «Change method», щоб змінити назву методу GetPersons на GetPeople, всі посилання на метод GetPersons у вихідному коді повинні бути ідентифіковані і замінені посиланнями на метод GetPeople. Ця проблема також виникає в онтологічному рефакторингу, навіть якщо онтологія не визначає поведінкову інформацію, оскільки онтології можуть містити загальні обмеження цілісності, наприклад, деякі правила SWRL – Semantic Web Rule Language. це мова правил, яка використовується для створення логічних правил у рамках семантичного вебу. Вона дозволяє описувати правила, які працюють разом із онтологіями, написаними, наприклад, на OWL (Web Ontology Language). Ці обмеження можуть стосуватися будь-якого поняття онтології. Тому після зміни концепції всі обмеження цілісності, які відносяться до цього поняття, повинні бути змінені для підтримки синтаксичної узгодженості онтології [16].

Іншою проблемою в онтологічному рефакторингу є той факт, що тести не можуть бути виконані, для того, щоб перевірити, що поведінка онтології не була змінена [17]. Тести такого роду необхідні в програмному забезпеченні та рефакторингу баз даних, але не можуть бути застосовані до онтологій, оскільки, як згадува-

doi: https://doi.org/10.15802/stp2026/356061

лося вище, онтології не можуть бути виконані. Однак перевірка певного роду може проводитися на різних онтологічних інстанціях для перевірки, після кожного виконання рефакторингової операції, чи були будь-які екземпляри втрачені внаслідок зміних [31]. Для виконання цього тесту операція рефакторингу повинна змінювати не тільки структуру онтології, але і її інформаційну базу (екземпляри онтології) [20].

Операції рефакторингу онтологій. Операції рефакторингу онтологій поділяються на два основних типи: структурний рефакторинг та рефакторинг обмежень цілісності [15].

Структурні рефакторингові операції: вони можуть рефакторувати структурні елементи онтології (поняття, типи відносин, узагальнюючі відносини та екземпляри та обмеження, вбудовані в мову онтології).

Рефакторинг обмежень цілісності: вони можуть рефакторувати динамічні елементи онтології, такі як правила виведення, написані в SWRL або OCL та його загальні обмеження цілісності [16].

Перша група рефакторингових операцій може бути застосована до будь-якої онтології, а друга група актуальна тільки в онтологіях, які містять загальні обмеження цілісності або операції [38].

Структурні рефакторингові операції – це ті, які змінюють структурні поняття онтології, тобто її класи, типи зв’язків, їх екземпляри та обмеження цілісності, які вбудовані в мову онтології, такі як відносини успадкування, які визначають, що поняття є підтипом іншого поняття, або обмеження типу відносин.

Рефакторингові операції структурного зв’язку класифікуються в межах трьох основних категорій відповідно до їх призначення [22]:

– переміщення ознак між поняттями: операції, які змінюють контекст, в якому визначаються властивості онтології (типи відносин і обмеження цілісності);

– організація даних: операції, які змінюють онтологічну структуру, щоб полегшити роботу з її даними;

– робота з узагальненням: операції, що використовуються для зміни таксономії або переміщення онтологічних властивостей (типів відносин і обмежень цілісності) через таксономію онтології.

Операції із даної категорії можуть бути використані для реструктуризації вмісту загальних обмежень цілісності. «Обмеженнями цілісності» називаються такі загальні обмеження цілісності, які не можуть бути представлені безпосередньо мовою онтології, і для їх вираження потрібно використовувати іншу мову, таку як мова OCL у випадку онтологій UML. Ці операції марні, коли онтологія, яка рефакторується, не допускає загальних обмежень цілісності, як у випадку онтології, яка використовує тільки мову OWL [16].

Рефакторингові операції обмеження цілісності класифікуються в межах двох основних категорій відповідно до їх призначення [21]:

– складання обмежень цілісності: операцій, які покращують якість обмежень цілісності;

– спрощення обмежень цілісності: операції, які спрощують умови та вирази, що використовуються в обмеженнях цілісності.

Рис. 1. Класифікація методів рефакторингу онтологій
Fig. 1. Classification of ontology refactoring methods

doi: https://doi.org/10.15802/stp2026/356061

Результати

У результаті проведеного дослідження здійснено систематичний аналіз наукових праць, присвячених рефакторингу програмного забезпечення, концептуальних моделей та онтологій. Встановлено, що еволюція підходів до рефакторингу характеризується поступовим переходом від локальних змін програмного коду до більш абстрактних рівнів, зокрема схем даних, UML-моделей та онтологічних структур. Показано, що більшість існуючих підходів до рефакторингу онтологій мають фрагментарний характер і зосереджуються на окремих операціях або тактиках без формування цілісної методології.

У ході аналізу виявлено основні проблеми застосування рефакторингу до онтологій, серед яких відсутність узгодженого визначення поняття «рефакторинг онтології», нестача уніфікованих каталогів рефакторингових операцій та обмеженість інструментальної підтримки. Водночас встановлено, що застосування рефакторингових підходів до онтологій позитивно впливає на їх модульність, когерентність та можливості повторного використання знань за умови збереження семантичної цілісності.

На основі узагальнення результатів проаналізованих досліджень визначено перспективні напрями подальшого розвитку, зокрема необхідність формування системного підходу до рефакторингу онтологій, розроблення критеріїв оцінювання якості змін та створення методологічного підґрунтя для автоматизації відповідних процесів.

Наукова новизна і практична значимість

Наукова новизна роботи полягає в систематичному аналізі та узагальненні підходів до рефакторингу програмного забезпечення з подальшим перенесенням ідей та методів рефакторингу на рівень онтологій як специфічного класу програмних і знаннєвих структур. Уперше рефакторинг онтологій розглядається не лише як технічний процес реструктуризації, а як інструмент підтримки якості знань, що охоплює структурні, семантичні та концептуальні аспекти. Запропоновано класифікацію методів рефакторингу за рівнем абстракції та характером впливу на онтологічні моделі, що дозволяє систематизувати наявні підходи та визначити їх сильні й слабкі сторони. Отримані результати сприяють поглибленню теоретичного розуміння механізмів рефакторингу в контексті знаннє-орієнтованих систем, семантичного вебу та кіберфізичних систем.

Практична значимість роботи полягає в можливості використання отриманих результатів під час проєктування, супроводу та еволюції онтологій і знаннє-орієнтованих інформаційних систем. Запропоновані підходи можуть бути застосовані для підвищення модульності, когерентності та повторного використання онтологічних моделей, а також для зменшення зчеплення між концептами без порушення семантичної цілісності. Результати дослідження можуть слугувати методологічною основою для розроблення інструментів автоматизованого рефакторингу онтологій із використанням методів машинного навчання та штучного інтелекту, а також для адаптації процесів рефакторингу до предметних галузей, що характеризуються високою динамікою змін.

Висновки

Протягом останніх двадцяти років рефакторинг зарекомендував себе як потужний і ефективний інструмент покращення програмного забезпечення. Його застосування не обмежується лише кодом – концепції рефакторингу успішно були адаптовані до інших типів концептуальних артефактів, таких як бази даних та UML-моделі. Проте у випадку з онтологіями подібна практика все ще перебуває на стадії становлення. Попри численні спроби, повноцінного впровадження рефакторингу до онтологій досі не досягнуто, що вказує на потребу в подальшому дослідженні та систематизації.

Основні виклики пов’язані з відсутністю чіткого визначення самого поняття «рефакторинг онтології» та нестачею узгодженого й системного підходу. Також бракує уніфікованого каталогу рефакторингових операцій, подібного до того, який існує в інженерії програмного забезпечення. Це значно ускладнює процес практичного застосування рефакторингу до онто-

doi: https://doi.org/10.15802/stp2026/356061

логій, особливо в контексті досягнення конкретних цілей кінцевого користувача.

У межах дослідження було розглянуто еволюцію терміну «рефакторинг» і проаналізовано його застосування до різних типів інформаційних структур. Особливу увагу було приділено існуючим підходам до рефакторингу онтологій, аналізу ефективності окремих методів та критеріям якості. У ході аналізу виявлено, що більшість наукових праць зосереджуються на описі окремих методів або тактик рефакторингу, однак не пропонують комплексних інструментів чи практичних рекомендацій щодо їх інтеграції у загальну стратегію роботи з онтологіями.

Можна зробити висновок про необхідність подальшого розвитку цієї сфери, створення повноцінного методологічного підґрунтя та інструментарію, що дозволить ефективно адаптувати рефакторинг до задач онтологічного моделювання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ambler S. *Agile database techniques: effective strategies for the agile software developer*. Hoboken : Wiley, 2003. 480 p.
2. Ambler S. *Agile modeling: effective practices for extreme programming and the unified process*. New York : John Wiley & Sons, 2002. 400 p.
3. Ambler S. W., Sadalage P. J. Refactoring databases: Evolutionary database design. Boston : Addison-Wesley Professional, 2006. 384 p.
4. Assenova P., Johannesson P. Improving quality in conceptual modelling by the use of schema transformations. *Conceptual modeling – ER'96 : 15th international conference on conceptual modeling*. Berlin : Springer, 1996. P. 277–291. DOI: https://doi.org/10.1007/BFb0019929
5. Astels D. Refactoring with UML. *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP2002)*. Alghero, Italy, 2002. P. 67–70.
6. Batini C., Ceri S., Navathe S. B. *Conceptual database design: an entity-relationship approach.* Redwood City : Benjamin/Cummings, 1992. 470 p.
7. Beck K. *Extreme programming explained.* Boston : Addison-Wesley, 1999.
8. Bergstein P. L. Object-preserving class transformations. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91). New York : ACM Press, 1991. P. 299–313. DOI: https://doi.org/10.1145/117954.117977
9. Bottoni P., Parisi-Presicce F., Taentzer G. Coordinated distributed diagram transformation for software evolution. *Electronic notes in theoretical computer science*. 2003. Vol. 72, Iss. 4. P. 59–70. DOI: https://doi.org/10.1016/s1571-0661(04)80626-1
10. Du Bois B. Opportunities and challenges in deriving metric impacts from refactoring postconditions. *Proceedings of WOOR'04*. 2004.
11. Du Bois B., Demeyer S., Verelst J. Refactoring – improving coupling and cohesion of existing code. *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*. Los Alamitos, CA : IEEE Computer Society, 2004. P. 144–151. DOI: https://doi.org/10.1109/wcre.2004.33
12. Casais E. Reorganizing an object system. *Object-oriented development.* Genève : Centre universitaire d'informatique, 1989. P. 161–189.
13. Chung L., Nixon B. A, Yu E., Mylopoulos J. *Non-Functional Requirements in Software Engineering* : monograph. Boston : Kluwer Academic Publishers, 2000. 439 p.
14. Conesa Caralt J. Ontology-Driven Information Systems: Pruning and Refactoring of Ontologies. *Doctoral Symposium of the 7th International Conference on the Unified Modeling Language (UML 2004)*. Lisbon, 2004.
15. Conesa Caralt J. *Pruning and refactoring ontologies in the development of conceptual schemas of information systems* : doctoral dissertation. Barcelona : Technical University of Catalonia, 2008.
16. Correa A., Werner C. Applying refactoring techniques to UML/OCL models. *Proceedings of the International Conference on the Unified Modeling Language (UML 2004). Lecture Notes in Computer Science*. Vol. 3273. Springer, Berlin, Heidelberg, 2004. P. 173–187. DOI: https://doi.org/10.1007/978-3-540-30187-5_13
17. Critchlow M., Dodd K., Chou J., van der Hoek A. Refactoring product line architectures. *Proceedings of the Workshop on Refactoring: Achievements, Challenges, Effects (REFACE'03)*. 2003. P. 23–26.
18. Ding L., Finin T., Joshi A., Pan R., Cost R. S., Peng Y., Reddivari P., Doshi V., Sachs J. Swoogle: A search and metadata engine for the semantic web. *Proceedings of the 13th ACM Conference on Information and*

doi: https://doi.org/10.15802/stp2026/356061

Knowledge Management (CIKM'04). (Washington, 8-13 November, 2004). Washington, 2004. P. 652–659. DOI: https://doi.org/10.1145/1031171.1031289

19. Eckhardt J., Vogelsang A., Méndez Fernández D. Are "Non-functional" Requirements Really Non-functional? An Investigation of Non-functional Requirements in Practice. *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. New York : ACM, 2016. P. 832–842. DOI: https://doi.org/10.1145/2884781.2884788
20. Eick S. G., Steffen J. L., Sumner E. E. SeeSoft – a tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering*. 1992. Vol. 18, Iss. 11. P. 957–968. DOI: https://doi.org/10.1109/32.177365
21. Ernst M. D., Cockrell J., Griswold W. G., Notkin D. Dynamically discovering likely program invariants to support program evolution. *IEEE transactions on software engineering*. 2001. Vol. 27, Iss. 2. P. 99–123. DOI: https://doi.org/10.1109/32.908957
22. Fowler M., Beck K., Brant J., Opdyke W., Roberts D. *Refactoring: Improving the Design of Existing Code.* Boston : Addison‑Wesley, 1999. 431 p.
23. Gamma E., Helm R., Johnson R., Vlissides J. *Design patterns: elements of reusable object-oriented software*. Boston : Addison-Wesley, 1995. 395 p.
24. Griswold W. G., Notkin D. Automated assistance for program restructuring. *ACM transactions on software engineering and methodology*. 1993. Vol. 2, Iss. 3. P. 228–269. DOI: https://doi.org/10.1145/152388.152389
25. Halpin T. *Information modeling and relational databases: from conceptual analysis to logical design*. San Francisco : Morgan Kaufmann, 2001. 763 p.
26. Halpin T. A., Proper H. A. Database schema transformation and optimization. *Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modeling (OOER'95)*. (Gold Coast, 12-15 December, 1995). Gold Coast, 1995. P. 191–203.
27. Kataoka Y., Ernst M. D., Griswold W. G., Notkin D. Automated support for program refactoring using invariants. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*. (Florence, 07-09 November 2001). Florence, 2001. P. 736–743. DOI: https://doi.org/10.1109/icsm.2001.972794
28. Mens K., Tourwé T. Delving source code with formal concept analysis. *Computer languages, systems & structures*. 2005. Vol. 31, Iss. 3-4. P. 183–197. DOI: https://doi.org/10.1016/j.cl.2004.11.004
29. Murphy-Hill E., Parnin C., Black A. P. How we refactor, and how we know it. *IEEE transactions on software engineering*. 2012. Vol. 38, Iss. 1. P. 5–18. DOI: https://doi.org/10.1109/tse.2011.41
30. Opdyke W. F. *Refactoring object-oriented frameworks* : PhD dissertation. Urbana-Champaign : University of Illinois at Urbana-Champaign, 1992.
31. Roberts D., Brant J., Johnson R. A refactoring tool for Smalltalk. *Theory and practice of object systems*. 1997. Vol. 3, Iss. 4. P. 253–263.
32. Shynkarenko V., Zhuchyi L. Ontological harmonization of railway transport information systems. *Proceedings of the 5th International Conference on Computational Linguistics and Intelligent Systems (COLINS-2021)* (Kharkiv, 22-23 April, 2021). Kharkiv, 2021. Vol. 2870. P. 541–554.
33. Shynkarenko V., Zhuchyi L. Semantic Checking of Different Type Information Sources About Permitted Speeds in Railway Transport. *CEUR Workshop Proceedings. 6th International Conference on Computational Linguistics and Intelligent Systems (COLINS 2022)* (Gliwice, 12-13 May, 2022). Gliwice, 2022. Vol. 3171. P. 711–723.
34. Shynkarenko V., Zhuchyi L., Ivanov O. Conceptualization of the tabular representation of knowledge. *2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT)*. (Lviv, 22-25 September, 2021). Lviv, 2021. P. 248–251. DOI: https://doi.org/10.1109/csit52700.2021.9648761
35. Shynkarenko V., Zhuchyi L., Ivanov O. Ontology-based semantic checking of data in railway infrastructure information systems. *Foundations of computing and decision sciences*. 2022. Vol. 47, Iss. 3. P. 291–319. DOI: https://doi.org/10.2478/fcds-2022-0016
36. Simon F., Steinbrückner F., Lewerentz C. Metrics based refactoring. *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001)*. (Lisbon, 14-16 March, 2001). Lisbon, 2001. P. 30–38. DOI: https://doi.org/10.1109/.2001.914965
37. Van Eetvelde N., Janssens D. A hierarchical program representation for refactoring. *Electronic notes in theoretical computer science*. 2004. Vol. 82, Iss. 7. P. 91–104. DOI: https://doi.org/10.1016/s1571-0661(04)80749-7
38. Van Gorp P., Stenten H., Mens T., Demeyer S. Towards automating source-consistent UML refactoring. «*UML» 2003 – The Unified Modeling Language. Modeling Languages and Applications. Lecture Notes in Computer Science*. 2003. Vol. 2863. P. 144–158. DOI: https://doi.org/10.1007/978-3-540-45221-8_15

doi: https://doi.org/10.15802/stp2026/356061

D. O. KARPOVSKYI[1*]

[1*]Dep. «Computer Science», Ukrainian State University of Science and Technologies, SEI DIIT, Lazaryana St., 2, Dnipro, Ukraine, 49010, phone +38 (056) 733 19 61, e-mail karpovskyi.d@gmail.com, ORCID 0009-0002-0020-5148

Methods and Tools for Refactoring Ontologies

Purpose. This work is aimed at investigating the evolution of the refactoring concept – from a tool for optimizing program code to a powerful means of improving data structures, algorithms, and business processes. The main purpose of the study is to examine the historical development of software and ontology refactoring, as well as the possibilities of applying improvement methods to ontologies as a specific type of software structure that plays a key role in knowledge-oriented systems, the Semantic Web, and next-generation cyber-physical systems. **Methodology.** The methodological basis of the study consists of general scientific and specialized methods of analysis and synthesis. A systematic review of scientific publications in the fields of software refactoring and ontology engineering was conducted using full-text and abstract scientometric databases. Comparative analysis methods were applied to compare different refactoring approaches, structural and functional analysis was used to study changes in software and ontology models, and classification methods were employed to group existing refactoring techniques by level of abstraction and application domain. **Finding.** Within the scope of the study, a comprehensive analysis of scientific publications devoted to refactoring and ontology engineering was performed using full-text and abstract databases. The evolution of refactoring approaches was examined, including data schema transformations, conceptual refactoring, modification of integrity constraints, and the development of tools for assessing the quality of changes. Particular attention was paid to the application of refactoring to ontologies, including an analysis of its impact on modularity, coherence, and knowledge reuse. **Originality.** The scientific novelty of the work lies in the systematic study of existing refactoring methods and their classification by type. A comprehensive analysis of the strengths and weaknesses of each approach was carried out, enabling a well-founded selection of optimal strategies for software improvement. The proposed approach contributes to a deeper understanding of refactoring mechanisms in the context of various application domains. **Practical value.** The presented results can be used in the design, maintenance, and evolution of knowledge-oriented systems, where structural consistency and semantic support play an important role. Prospects for further research are outlined, including the automation of ontology refactoring processes using machine learning methods, the expansion of evaluation metric systems, and the adaptation of the proposed solutions to application domains characterized by a high degree of change dynamics.

Keywords: information technology; ontology; refactoring; knowledge bases; refactoring methods; quality metrics; software; schema transformation; refactoring quality

REFERENCES

1. Ambler, S. (2002). *Agile modeling: effective practices for extreme programming and the unified process*. New York: John Wiley & Sons. (in English)
2. Ambler, S. (2003). *Agile database techniques: effective strategies for the agile software developer*. Hoboken: Wiley. (in English)
3. Ambler, S. W., & Sadalage, P. J. (2006). Refactoring databases: Evolutionary database design. Boston : Addison-Wesley Professional. (in English)
4. Assenova, P., & Johannesson, P. (1996). Improving quality in conceptual modelling by the use of schema transformations. *Conceptual modeling – ER'96 : 15th international conference on conceptual modeling*. (pp. 277-291). Berlin: Springer. DOI: https://doi.org/10.1007/BFb0019929 (in English)
5. Astels, D. (2002). Refactoring with UML. In *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP2002)* (pp. 67-70). Alghero, Sardinia, Italy. (in English)
6. Batini, C., Ceri, S., & Navathe, S. B. (1992). *Conceptual database design: an entity-relationship approach*. Redwood City: Benjamin/Cummings. (in English)
7. Beck, K. (1999). *Extreme programming explained*. Boston: Addison-Wesley. (in English)
8. Bergstein, P. L. (1991). Object-preserving class transformations. *In Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)* (pp. 299-313). New York, NY: ACM Press. DOI: https://doi.org/10.1145/117954.117977 (in English)

doi: https://doi.org/10.15802/stp2026/356061

9. Bottoni, P., Parisi-Presicce, F., & Taentzer, G. (2003). Coordinated Distributed Diagram Transformation for Software Evolution. *Electronic Notes in Theoretical Computer Science, 72*(4), 59-70. DOI: https://doi.org/10.1016/s1571-0661(04)80626-1 (in English)
10. Du Bois, B. (2004). Opportunities and challenges in deriving metric impacts from refactoring postconditions. *Proceedings of WOOR'04*. (in English)
11. Du Bois, B., Demeyer, S., & Verelst, J. (2004). Refactoring – improving coupling and cohesion of existing code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)* (pp. 144-151). IEEE. DOI: https://doi.org/10.1109/wcre.2004.33 (in English)
12. Casais, E. (1989). Reorganizing an object system. *Object-oriented development*. (pp. 161-189). Genève: Centre universitaire d'informatique (in English)
13. Chung, L., Nixon, B. A., Yu, E., & Mylopoulos, J. (2000). *Non-functional requirements in software engineering*: monograph. Boston: Kluwer Academic Publishers. (in English)
14. Conesa Caralt, J. (2004). Ontology-driven information systems: Pruning and refactoring of ontologies. In *Doctoral Symposium of the 7th International Conference on the Unified Modeling Language (UML 2004),* Lisbon, Portugal. (in English)
15. Conesa Caralt, J. (2008). *Pruning and refactoring ontologies in the development of conceptual schemas of information systems* (Doctoral dissertation). Technical University of Catalonia, Barcelona, Spain. (in English)
16. Correa, A., & Werner, C. (2004). Applying refactoring techniques to UML/OCL models. In *Proceedings of the International Conference on the Unified Modeling Language (UML 2004) Lecture Notes in Computer Science*. (Vol. 3273, pp. 173-187). Springer, Berlin, Heidelberg. DOI: https://doi.org/10.1007/978-3-540-30187-5_13 (in English)
17. Critchlow, M., Dodd, K., Chou, J., & van der Hoek, A. (2003). Refactoring product line architectures. In *Proceedings of the Workshop on Refactoring: Achievements, Challenges, Effects (REFACE'03)* (pp. 23-26). (in English)
18. Ding, L., Finin, T., Joshi, A., Pan, R., Cost, R. S., Peng, Y., Reddivari, P., Doshi, V., & Sachs, J. (2004, November). Swoogle: A search and metadata engine for the semantic web. In *Proceedings of the 13th ACM Conference on Information and Knowledge Management (CIKM '04)* (pp. 652-659). Washington, USA. DOI: https://doi.org/10.1145/1031171.1031289 (in English)
19. Eckhardt, J., Vogelsang, A., & Méndez Fernández, D. (2016). Are "non-functional" requirements really non-functional? An investigation of non-functional requirements in practice. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)* (pp. 832-842). New York, USA. DOI: https://doi.org/10.1145/2884781.2884788 (in English)
20. Eick, S. G., Steffen, J. L., & Sumner, E. E. (1992). SeeSoft – A tool for visualizing line-oriented software statistics. *IEEE Transactions on Software Engineering, 18*(11), 957-968. DOI: https://doi.org/10.1109/32.177365 (in English)
21. Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering, 27*(2), 99-123. DOI: https://doi.org/10.1109/32.908957 (in English)
22. Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code.* Boston: Addison-Wesley. (in English)
23. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Boston: Addison-Wesley. (in English)
24. Griswold, W. G., & Notkin, D. (1993). Automated assistance for program restructuring. *ACM transactions on software engineering and methodology, 2*(3), 228-269. DOI: https://doi.org/10.1145/152388.152389 (in English)
25. Halpin, T. (2001). *Information modeling and relational databases: from conceptual analysis to logical design.* San Francisco: Morgan Kaufmann. (in English)
26. Halpin, T. A., & Proper, H. A. (1995, December). Database schema transformation and optimization. In *Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modeling (OOER'95)* (pp. 191-203). Gold Coast, Australia. (in English)
27. Kataoka, Y., Ernst, M. D., Griswold, W. G., & Notkin, D. (2001, November). Automated support for program refactoring using invariants. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)* (pp. 736-743). Florence, Italy. DOI: https://doi.org/10.1109/icsm.2001.972794 (in English)
28. Mens, K., & Tourwé, T. (2005). Delving source code with formal concept analysis. *Computer Languages, Systems & Structures, 31*(3-4), 183-197. DOI: https://doi.org/10.1016/j.cl.2004.11.004 (in English)

doi: https://doi.org/10.15802/stp2026/356061

29. Murphy-Hill, E., Parnin, C., & Black, A. P. (2012). How we refactor, and how we know it. *IEEE transactions on software engineering*, *38*(1), 5-18. DOI: https://doi.org/10.1109/tse.2011.41 (in English)
30. Opdyke, W. F. (1992). *Refactoring object-oriented frameworks* (Doctoral dissertation). Urbana-Champaign: University of Illinois at Urbana-Champaign (in English)
31. Roberts, D., Brant, J., & Johnson, R. (1997). A refactoring tool for Smalltalk. *Theory and practice of object systems*, *3*(4), 253-263. (in English)
32. Shynkarenko, V., & Zhuchyi, L. (2021, April). Ontological harmonization of railway transport information systems. In *Proceedings of the 5th International Conference on Computational Linguistics and Intelligent Systems (COLINS-2021)* (Vol. 2870, pp. 541-554). Kharkiv, Ukraine. (in English)
33. Shynkarenko, V., & Zhuchyi, L. (2022, May). Semantic Checking of Different Type Information Sources About Permitted Speeds in Railway Transport. *CEUR Workshop Proceedings. 6th International Conference on Computational Linguistics and Intelligent Systems (COLINS 2022)* (Vol. 3171, pp. 711-723). Gliwice, Poland. (in English)
34. Shynkarenko, V., Zhuchyi, L., & Ivanov, O. (2021, September). Conceptualization of the tabular representation of knowledge. In *2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT)* (pp. 248-251). Lviv, Ukraine. DOI: https://doi.org/10.1109/csit52700.2021.9648761 (in English)
35. Shynkarenko, V., Zhuchyi, L., & Ivanov, O. (2022). Ontology-based semantic checking of data in railway infrastructure information systems. *Foundations of computing and decision sciences*, *47*(3), 291-319. DOI: https://doi.org/10.2478/fcds-2022-0016 (in English)
36. Simon, F., Steinbrückner, F., & Lewerentz, C. (2001, March). Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR 2001)* (pp. 30-38). Lisbon, Portugal. DOI: https://doi.org/10.1109/.2001.914965 (in English)
37. Van Eetvelde, N., & Janssens, D. (2004). A hierarchical program representation for refactoring. *Electronic notes in theoretical computer science*, *82*(7), 91-104. DOI: https://doi.org/10.1016/s1571-0661(04)80749-7 (in English)
38. Van Gorp, P., Stenten, H., Mens, T., & Demeyer, S. (2003). Towards automating source-consistent UML refactoring. «*UML» 2003 – The Unified Modeling Language. Modeling Languages and Applications*. *Lecture Notes in Computer Science*, *2863*, 144-158. https://doi.org/10.1007/978-3-540-45221-8_15 (in English)

Надійшла до редколегії: 12.11.2025
Рекомендовано до публікації: 22.12.2025
Дата публікації: 27.03.2026

doi: https://doi.org/10.15802/stp2026/356061