Міністерство освіти і науки
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Кваліфікаційна наукова
праця на умовах рукопису

ТКАЧУК АНДРІЙ ВІТАЛІЙОВИЧ

УДК 004.4’23:004.4’24:004.43:004.82

ДИСЕРТАЦІЯ

МЕТОД АВТОМАТИЗАЦІЇ РЕФАКТОРИНГУ ПРОГРАМНОГО КОДУ З ВИКОРИСТАННЯМ БАЗИ ЗНАНЬ

122 – Комп’ютерні науки
12 – Інформаційні технології

Подається на здобуття наукового ступеня доктора філософії

Дисертація містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело
_____ А. В. Ткачук

Науковий керівник: к.т.н. Б. В. Булах

Київ – 2024

АНОТАЦІЯ

Ткачук. А. В. Методи автоматизації рефакторингу програмного коду з використанням бази знань. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії за спеціальністю 122 «Комп'ютерні науки». – Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», Київ, 2024.

Метою дисертаційного дослідження є розробка методу аналізу початкового коду та його подальшого автоматизованого рефакторингу, який спирається на використання бази знань про цей код.

Об'єктом дисертаційного дослідження є процеси збору, представлення і обробки інформації про початковий код програмних продуктів.

Предметом дисертаційного дослідження виступають методи аналізу, представлення та обробки інформації для рефакторингу початкового коду з урахуванням його семантики, які базуються на використанні формалізованих знань про код.

У ході роботи над дисертацією було використано такі методи дослідження: спостереження, порівняння, аналіз, формалізація, абстрагування, моделювання, експеримент. Зазначені методи були обрані зважаючи на мету та завдання дослідження, а також на практичність їх використання при вирішенні проблем автоматизованого аналізу та рефакторингу початкового коду прикладних програм.

У дисертації *вперше запропоновано* метод автоматизації рефакторингу програмного коду, який *відрізняється від існуючих тим*, що здійснює перетворення коду на рівні сутностей бази знань, створеної на основі запропонованої моделі формалізації знань про синтаксис і семантику коду, для автоматичного виявлення і виправлення розповсюджених антишаблонів програмування.

Вперше запропоновано метод модифікації проміжного представлення програмного коду в процесі рефакторингу, що уможливлює виправлення

нетривіальних антишаблонів, який *відрізняється від існуючих* тим, що застосовує механізми логічного виведення до знань про код, сформульованих з використанням описових логік і правил логічного виведення.

Удосконалено модель формалізації знань про програмний код для його рефакторингу на основі онтології об'єктно-орієнтованої мови програмування, що *відрізняється від існуючих тим*, що представляє не лише базові відомості про синтаксис, отримані від синтаксичного аналізатора, а й семантику складних конструкцій, шаблонів та антишаблонів з можливою прив'язкою до функцій програмного продукту.

Розроблений метод дозволяє *перевикористовувати схему бази знань* із накопиченими знаннями про шаблони та антишаблони під час аналізу та рефакторингу початкового коду різними мовами програмування, що дозволяє зменшити час та витрати на розробку систем рефакторингу, розробляючи лише частину системи, що пов'язана із перетворення початкового коду в сутності бази знань і навпаки.

Запропонований підхід до формалізації знань про початковий код дозволяє *отримувати метадані* про зміст початкового коду, *порівнювати* наповнення репозиторіїв (*шукати семантичні дублікати*), створювати *документацію* про програмний продукт з допомогою вже існуючих засобів для роботи із семантичними даними, *описувати шаблони і антишаблони* для проведення аналізу і рефакторингу без втручання розробника.

Запропоновано *застосування розробленого методу для побудови програмного прототипу* системи аналізу і рефакторингу початкового коду для мови програмування Swift з описом онтології мовою OWL-DL, а правил логічного виводу та запитів до онтології – мовою S(Q)WRL, що дало змогу значно пришвидшити процес здійснення рефакторингу. Таким чином досягається суттєве зменшення витрат на розробку та підтримку програмних продуктів.

Дисертація складається із семи розділів.

У **першому розділі** розглянуто автоматизовані системи рефакторингу. Дано визначення системи автоматизованого рефакторингу, порівняно такі системи за побудовою і принципом роботи, розглянуто їх переваги і недоліки. Розглянуто можливість застосування і використання семантичних підходів до вирішення задач рефакторингу. Сформульовано висновок про те, що ефективне функціонування інструментів автоматизованого рефакторингу значною мірою залежить від автоматизованого аналізу початкового коду. Аналіз дає необхідне уявлення про структуру та семантику коду, гарантуючи безпечність і послідовність рефакторингу. Зазначено, що включення семантичного розуміння в статичний аналіз коду та рефакторинг забезпечує глибоке й точне розуміння поведінки коду, що призводить до більш точних і надійних результатів.

У **другому розділі** детально розглянуто поняття онтології та її застосування в якості основи бази знань. Дано визначення описових (дескриптивних) логік та мови OWL на їх основі, а також описано версії та профілі мови, її основні поняття та особливості, концентруючись на можливостях представлення семантики і способах логічного виведення засобами OWL. Дано визначення системи логічного виведення, розглянуто і порівняно алгоритми, що застосовуються такими системами. Розглянуто можливості доповнення виразності OWL у вигляді правил SWRL та запитів SQWRL. Було зроблено висновок, що коли необхідно проаналізувати велику кодову базу або знайти шаблони в ній, онтологія, описана в OWL, та пов'язані з нею можливості логічного виведення можуть бути надзвичайно цінними. Використання системи логічного виведення (англ. reasoner) дозволяє створити логічно узгоджені та максимально інформативні онтології, тим самим максимізуючи їх корисність у таких застосуваннях як відкриття знань, інтеграція даних та семантичний пошук.

У **третьому розділі** досліджено можливості вбудованого рефакторингу у мові Swift. Розглянуто можливості додавання нових дій рефакторингу до мови програмування, а також оцінено можливість масштабування. Проведено оцінку отриманих результатів. У результаті експериментів було встановлено, що для

додавання лише одної відносно простої дії рефакторингу необхідно написати близько 300 рядків коду. І немає причин вважати, що в інших випадках потрібно буде написати суттєво меншу кількість рядків коду. Таким чином, існує актуальна потреба в удосконаленні та автоматизації процесу розширеного рефакторингу, що дозволила б розробнику уникнути написання чималої кількості «інфраструктурного» коду, більше зосередившись на кінцевій меті рефакторингу.

У **четвертому розділі** розглянуто застосування формалізованих знань про початковий код для здійснення рефакторингу у мові Swift. Досліджено можливість подолання існуючих перепон при додаванні нових дій рефакторингу із застосуванням формалізованих знань. Запропоновано формальне представлення початкового коду у вигляді окремої бази знань. У результаті проведеного дослідження встановлено, що використання запропонованого методу дозволяє виконувати складні завдання рефакторингу за допомогою простої вербальної формалізованої команди.

У **п'ятому розділі** доведено, що онтології можна використовувати, щоб ефективно зберігати знання зокрема про початковий код. Розглянуто кілька підходів до перетворення кодової бази в базу знань про код як сукупність фактів про код, що спираються на онтологію.

У **шостому розділі** проведено розширення можливостей здійснення аналізу та рефакторингу за допомогою описових правил. До пропонованої моделі додано набір правил, що описують певні дії рефакторингу, і здійснено перевірку їх застосування для доведення ефективності.

У **сьомому розділі** описано ряд проведених експериментів із застосуванням запропонованого методу та моделі для перевірки справедливості їх теоретичних очікуваних властивостей та характеристик. Здійснено аналіз статистичних даних, отриманих шляхом опитування експертів щодо швидкодії та взаємодії і роботи із запропонованими методом і моделлю. Проведено перевірку роботи методу і моделі на різних наборах вихідних даних, а також якісне порівняння із існуючими програмними рішеннями для рефакторингу.

Ці розділи разом підкреслюють практичне застосування та ефективність використання онтологій та формалізованих знань у розробці програмного забезпечення, зокрема в контексті рефакторингу та аналізу коду. Дослідження демонструє потенціал цього підходу у підвищенні ефективності процесів розробки програмного забезпечення.

У дисертації запропоновано метод аналізу та рефакторингу початкового коду прикладних програм, який через використання бази знань про початковий код на основі онтології, дозволяє абстрагуватись від конкретної мови програмування в проекті і здійснювати складні маніпуляції із початковим кодом шляхом маніпуляції його семантичним представленням. Основна ідея методу полягає у роботі із представленням коду на семантичному рівні, що відрізняється від існуючих способів використанням онтології та засобів OWL/S(Q)WRL для роботи із семантикою. У ході експериментальних досліджень було доведено можливість відносно просто описувати правила рефакторингу в онтології та здійснювати аналіз і рефакторинг різної складності, а також показано зменшення витрат часу на виконання таких завдань на 36% у порівнянні із іншими підходами аналізу і рефакторингу.

Ключові слова: програмне забезпечення, рефакторинг, синтаксичний аналіз коду, інженерія програмного забезпечення, інверсія залежностей, комп’ютерне моделювання, модель представлення знань, база знань, класифікатор, онтологія, описові логіки, мова OWL, семантичні правила SWRL, продукційні системи, механізми логічного виведення.

Список публікацій здобувача за темою дисертації:

Статті у наукових фахових виданнях України

[1] 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.

[2] Tkachuk, A., & Bulakh, B. (2022). Usage of formalized knowledge about source code for refactoring actions in Swift. *Technology Audit and Production Reserves, 2(68)*(6), 6–10.

[3] Tkachuk, A., & Bulakh, B. (2023). Describing the knowledge about the source code using an ontology. *Infocommunication and Computer Technologies, 5*(1), 123-133.

ABSTRACT

Tkachuk A. Method of source code refactoring automation using a knowledge base. – A qualifying scientific work as a manuscript.

Thesis for the degree of Doctor of Philosophy in specialty 122 “Computer Science”. – National Technical University of Ukraine “Igor Sikorsky Kyiv Polytechnic Institute”, Kyiv, 2024.

The aim of the dissertation research is the development of a method for analyzing source code and further automated refactoring, based on the utilization of knowledge about that code.

The object of the dissertation research is the processes of analysis, representation, and processing of information about the source code of software products.

The subject of the dissertation research is the methods of analysis, representation, and processing of information for refactoring of source code, considering its semantics, which is based on the utilization of formalized knowledge about the code.

In the course of working on the dissertation, the following research methods were used: observation, comparison, analysis, formalization, abstraction, modeling, experiment. These methods were chosen considering the goal and tasks of the research, as well as the practicality of their use in solving problems of automatic analysis and refactoring of the source code of application programs.

The dissertation introduces, *for the first time*, a method for automating code refactoring, which *differs from existing ones* in that it performs transformations on the level of entities of knowledge base, which is based on proposed model for formalizing knowledge about the syntax and semantics of source code, for the automatic detection and correction of common programming anti-patterns, considering the semantics of the code.

A method for modifying the intermediate representation of the code during refactoring is *proposed for the first time*, enabling the correction of non-trivial anti-

patterns, which *differs from existing ones* in that it applies logical inference mechanisms to the knowledge about the code formulated using descriptive logics and rules of logical inference.

The model for formalizing knowledge about the source code for its refactoring based on the ontology of the object-oriented programming language *is improved*. This model *differs from existing ones* in that it represents not only basic information about syntax obtained from the syntax analyzer, but also the semantics of complex constructions, patterns, and anti-patterns with possible linkage to the functions of the software product.

The developed method enables *the reuse of the knowledge base schema* with accumulated knowledge about patterns and anti-patterns during the analysis and refactoring of the source code in different programming languages, which reduces the time and costs of developing refactoring systems by only developing the part of the system related to converting the source code into knowledge base entities and vice versa.

The proposed approach to formalizing knowledge about the source code allows for *obtaining metadata* about the content of the source code, *comparing* repository contents (*searching for semantic duplicates*), creating *documentation* about the software product using existing tools for working with semantic data, and *describing patterns and anti-patterns* for analysis and refactoring without developer intervention.

The developed *method is proposed to be applied* for building a software prototype of a system for the analysis and refactoring of source code for the Swift programming language utilizing: ontology which is described using the OWL-DL language, the rules for logical inference and querying the ontology which are expressed using the S(Q)WRL language, which provided the ability to reduce the time required to conduct the refactoring. This approach achieves a significant reduction in costs for the development and maintenance of software products.

The dissertation consists of seven chapters.

In the first chapter, automated refactoring systems are discussed. A definition of an automated refactoring system is provided, comparing such systems in terms of structure and operating principles, and examining their advantages and disadvantages. The possibility of applying and using semantic approaches to solve refactoring tasks is considered. It is concluded that the effective functioning of automatic refactoring tools largely depends on the automatic analysis of the source code. This analysis provides the necessary understanding of the code's structure and semantics, ensuring the safety and consistency of refactoring. It was noted that incorporating semantic understanding into static code analysis and refactoring provides a deep and accurate understanding of code behavior, leading to more precise and reliable results.

In the second chapter, the concept of ontology and its application as a knowledge base foundation is thoroughly examined. A definition of description logics OWL (Web Ontology Language) on their basis is provided, along with descriptions of its versions and profiles, its main concepts, and features, focusing on the capabilities for representing semantics and the ways of logical reasoning with OWL tools. A definition of a logical reasoning system is given, and algorithms used by such systems are discussed and compared. The possibilities of enhancing OWL's expressiveness in the form of SWRL (Semantic Web Rule Language) rules and SQWRL (Semantic Query-Enhanced Web Rule Language) queries are considered. The conclusion is drawn that when it's necessary to analyze a large code base or find patterns within it, an ontology described in OWL, along with associated logical reasoning capabilities, can be extremely valuable. The use of a logical reasoning system (reasoners) allows for the creation of logically coherent and maximally informative ontologies, thereby maximizing their usefulness in applications such as knowledge discovery, data integration, and semantic search.

In the third chapter, the possibilities of built-in refactoring in the Swift programming language are explored. The chapter examines the potential for adding new refactoring actions to the programming language, as well as the feasibility of scaling these additions. An evaluation of the obtained results is conducted. As a result

of the experiments, it was found that to add just one relatively simple refactoring action, it was necessary to write about 300 lines of code. And there is no reason to believe that significantly fewer lines of code would be required in other cases. Thus, there is a current need for the improvement and automation of the extended refactoring process, which would allow developers to avoid writing a substantial amount of "infrastructure" code, focusing more on the ultimate goal of refactoring. This indicates a gap in the efficiency of the refactoring process in Swift, emphasizing the necessity for more streamlined and automated solutions.

In the fourth chapter, the application of formalized knowledge about source code for refactoring in the Swift programming language is considered. The chapter investigates the possibility of overcoming existing barriers in adding new refactoring actions through the application of formalized knowledge. A formal representation of the source code in the form of a separate knowledge base is proposed. As a result of the conducted research, it was established that the use of the proposed method allows for the execution of complex refactoring tasks using simple verbalized formalized commands. This approach significantly streamlines the refactoring process, enabling more efficient and effective manipulation of the source code, and potentially reducing the need for extensive coding to implement new refactoring actions.

In the fifth chapter, it was proven that ontologies can be effectively used to store knowledge, particularly about source code. Several approaches to transforming a codebase into a knowledge base about code (as a collection of facts about the code base), which comprise on an ontology, have been considered.

In the sixth chapter, the capabilities of conducting analysis and refactoring using descriptive rules were expanded. A set of rules describing certain refactoring actions was added to the proposed model, and their application was tested to prove effectiveness.

The seventh chapter involved a series of experiments using the proposed method and model to verify the accuracy of their theoretical expected properties and characteristics. An analysis of statistical data was conducted, obtained through surveys

of experts regarding the performance and interaction with the proposed method and model. The functionality of the method and model was tested on different data sets, and a qualitative comparison with existing software products for refactoring was made.

These chapters collectively highlight the practical application and effectiveness of using ontologies and formalized knowledge in software development, particularly in the context of refactoring and code analysis. The research demonstrates the potential of this approach in enhancing the efficiency and accuracy of software development processes.

In the dissertation, a method for analyzing and refactoring the source code of application programs is proposed, which, through the use of knowledge base about the source code (supported by an ontology), allows abstraction from the specific programming language used in the project and enables complex manipulations of the source code by manipulating its semantic representation. The main idea of the method is to work with the representation of code at a semantic level, which differs from existing methods by using ontology and OWL/S(Q)WRL tools for working with semantics. During experimental research, it was proven that it is possible to relatively easy describe refactoring rules in ontology and perform analysis and refactoring of various complexities. Moreover, it was shown that the time needed to perform such tasks is reduced by 36% compared to other approaches to analysis and refactoring.

Key words: software, refactoring, syntactic code analysis, software engineering, dependency inversion, computer modeling, knowledge representation model, knowledge base, classifier, ontology, descriptive logics, OWL language, SWRL semantic rules, production systems, logical inference mechanisms.

List of research papers and publications:

Articles in scientific professional publications of Ukraine

[1] 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.

[2] Tkachuk, A., & Bulakh, B. (2022). Usage of formalized knowledge about source code for refactoring actions in Swift. *Technology Audit and Production Reserves, 2(68)*(6), 6–10.

[3] Tkachuk, A., & Bulakh, B. (2023). Describing the knowledge about the source code using an ontology. *Infocommunication and Computer Technologies, 5*(1), 123-133.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – База даних

БЗ – База знань

ШІ – Штучний інтелект

API – Application Programming Interface

AST – Abstract Syntax Tree

BDD – Binary Diagram of Decisions

DAML – DARPA Agent Markup Language

DL – Description Logic

DTO – Data Transfer Object

FOL – First-Order Logic

GA – Genetic Algorithms

IDE – Integrated Development Environment

IRI – Internationalized Resource Identifier

JSON – JavaScript Object Notation

OIL – Ontology Interface Layer

OWL – Web Ontology Language

RDF – Resource Description Framework

RDFS – Resource Description Framework Schema

REST – Representation State Transfer

RSR – Recommendation System for Refactoring

SPARQL – SPARQL Protocol and RDF Query Language

SWRL – Semantic Web Rule Language

SQWRL – Semantic Query-Enhanced Web Rule Language

UML – Unified Modeling Language

UUID – Universally Unique Identifier

ВСТУП

Актуальність теми. Автоматизований аналіз та рефакторинг початкового коду сприяє створенню надійних програмних продуктів, забезпечуючи дотримання визначеного рівня якості в межах встановлених часових рамок і обмежень ресурсів. Однак, існуючі просунуті інструменти аналізу та рефакторингу часто обмежуються лише найпопулярнішими мовами програмування. Це пов'язано з тим, що розробка та підтримка таких засобів вимагає значних зусиль і ресурсів, що може уповільнювати їх еволюцію та інноваційний розвиток.

Застосування зазначених інструментів у комерційних проектах, за даними аналізу (Pinto & Kamei, 2013), не є широко поширеним. Це обумовлено потребою у додаткових ресурсах для їх налаштування та використання, а також через те, що вони не завжди виправдовують очікувані результати.

Розробка нових систем та методів рефакторингу є актуальною з огляду на динамічний характер технологічного розвитку в області програмної інженерії, де постійно виникають нові вимоги та виклики. Сучасні програмні системи характеризуються високою складністю та взаємозалежністю компонентів, що зумовлює необхідність в їхньому систематичному оновленні та покращенні з метою підтримання високих стандартів продуктивності, безпеки та масштабованості.

Розвиток нових методів рефакторингу спрямований на поліпшення характеристик коду (важливо зазначити, що стандарти ISO 25010, ISO 9126 зауважують властивості систем, але деякі з таких властивостей можуть стосуватися власне початкового коду): зниження його складності, виправлення помилок, забезпечення кращої сумісності та розширюваності. Це, у свою чергу, сприяє зменшенню витрат часу та ресурсів на підтримку та розвиток програмних продуктів в межах процесу інженерії програмного забезпечення.

Варто зауважити, що хоч рефакторинг сам по собі (за визначенням) не забезпечує якість, однак правильне і доцільне застосування методів

рефакторингу може допомогти із покращенням таких характеристик якості, як тестоздатність, ремонтопридатність, багаторазове використання, аналізованість та інші.

Крім того, нові методи рефакторингу можуть включати підтримку останніх технологічних інновацій, що забезпечують автоматизацію та підвищення точності процесів рефакторингу. Вони також можуть бути адаптовані до специфічних мов програмування або парадигм, що забезпечує більш ефективну підтримку сучасних програмних розробок.

Таким чином, постійний розвиток методів та систем рефакторингу є ключовим для задоволення поточних та майбутніх потреб у сфері програмної інженерії, що сприяє підвищенню ефективності та якості розробки програмного забезпечення.

Мета і завдання дослідження. ***Метою*** дослідження є розробка методу аналізу та автоматизованого рефакторингу початкового коду прикладного програмного забезпечення, який спирається на базу формалізованих знань про цей код, використану мову програмування та предметну область застосування конкретного програмного продукту.

Для досягнення поставленої мети необхідно розв'язати наступні ***завдання***:

- проаналізувати існуючі методи аналізу коду та здійснення його рефакторингу, зокрема такі, що залучають формалізовані знання про код; зробити висновок про доцільність застосування семантичних підходів під час здійснення рефакторингу;
- провести аналіз засобів і способів представлення семантичної інформації, роботи із нею, обґрунтувати можливість використання таких засобів для побудови формалізованого представлення знань про початковий код;
- дослідити можливості додавання та параметризації нових дій рефакторингу до інструментарію розробки в рамках об’єктно-орієнтованої парадигми на прикладі мови програмування Swift;

- провести дослідження способів формалізації знань про початковий код і можливостей їх представлення і використання у процесі рефакторингу початкового коду;
- запропонувати метод автоматизованого рефакторингу початкового коду таким чином, щоб його основою якого була б база формалізованих знань про код, а також інструменти оперування цими знаннями;
- запропонувати метод модифікації представлення коду для виправлення нетривіальних антишаблонів з можливістю представляти формалізовані знання про початковий код довільною мовою програмування для подальшого аналізу та рефакторингу і вдосконалити модель формалізації знань про початковий код для його представлення;
- розробити прототип програмного інструментарію, що використовує запропоновані методи, та на практиці порівняти його з існуючими рішеннями для рефакторингу, оцінити його потреби в обчислювальних ресурсах.

Об’єкт дослідження. Об'єктом дисертаційного дослідження є процеси збору, представлення і обробки інформації про початковий код програмних продуктів.

Предмет дослідження. Предметом дисертаційного дослідження виступають методи аналізу, представлення та обробки інформації для рефакторингу початкового коду з урахуванням його семантики, які базуються на використанні формалізованих знань про код.

Методи дослідження. Спостереження, порівняння, аналіз, формалізація, абстрагування, моделювання, експеримент. Зазначені методи були обрані зважаючи на мету та завдання дослідження, а також на практичність їх використання при вирішенні проблем автоматизованого аналізу та рефакторингу початкового коду прикладних програм.

Наукова новизна отриманих результатів.

1. *Вперше запропоновано* метод автоматизації рефакторингу програмного коду, який *відрізняється від існуючих тим*, що здійснює перетворення коду на рівні сутностей бази знань, створеної на основі запропонованої моделі формалізації знань про синтаксис і семантику коду, для автоматичного виявлення і виправлення розповсюджених антишаблонів програмування.
2. *Вперше запропоновано* метод модифікації проміжного представлення програмного коду в процесі рефакторингу, що уможливлює виправлення нетривіальних антишаблонів, який *відрізняється від існуючих* тим, що застосовує механізми логічного виведення до знань про код, сформульованих з використанням описових логік і правил логічного виведення.
3. *Удосконалено* модель формалізації знань про програмний код для його рефакторингу на основі онтології об’єктно-орієнтованої мови програмування, що *відрізняється від існуючих тим*, що представляє не лише базові відомості про синтаксис, отримані від синтаксичного аналізатора, а й семантику складних конструкцій, шаблонів та антишаблонів з можливою прив'язкою до функцій програмного продукту.

Зв’язок роботи з науковими програмами, планами, темами. Дослідження з автоматизації процесів написання, аналізу та підтримки коду прикладного програмного забезпечення з використанням семантичного підходу виконувались в рамках тематичного плану науково-дослідних робіт кафедри системного проектування, зокрема в рамках ініціативної теми «Впровадження сервіс-орієнтованого підходу до реалізації процесів діджиталізації суспільства (тема СП – 01/2023)» (№ держреєстрації 0123U101333), що виконується згідно Тематичного плану виконання ініціативних кафедральних науково-дослідних робіт Навчально-наукового інституту прикладного системного аналізу КПІ ім. Ігоря Сікорського. На основі отриманих результатів дослідження було

вдосконалено комп’ютерний практикум для предмету «Об’єктно-орієнтоване програмування».

Практичне значення отриманих результатів. Розроблений метод дозволяє *перевикористовувати схему бази знань* із накопиченими знаннями про шаблони та антишаблони під час аналізу та рефакторингу початкового коду різними мовами програмування, що дає змогу зменшити час та витрати на розробку систем рефакторингу, розробляючи лише частину системи, що пов'язана із перетворення початкового коду в базу знань і навпаки.

Запропонований метод до формалізації знань про початковий код дозволяє *отримувати метадані* про зміст початкового коду, *порівнювати* наповнення репозиторіїв (*шукати семантичні дублікати*), створювати *документацію* про програмний продукт з допомогою вже існуючих засобів для роботи із семантичними даними, *описувати шаблони і антишаблони* для проведення аналізу і рефакторингу без втручання розробника.

Запропоновано *застосування розробленого методу для побудови програмного прототипу* системи аналізу і рефакторингу початкового коду для мови програмування Swift з описом онтології мовою OWL-DL, а правил логічного виводу та запитів до онтології – мовою S(Q)WRL, що дало змогу значно пришвидшити процес здійснення рефакторингу. Таким чином досягається суттєве зменшення витрат на розробку та підтримку програмних продуктів.

Особистий внесок здобувача. Усі основні результати дисертаційного дослідження, представлені до захисту, одержані автором особисто. У публікаціях у співавторстві здобувачеві належать: дослідження існуючих методів, проектування системи рефакторингу, розробка методу рефакторингу на основі формалізованих знань, представлених в онтології з використанням правил, експериментальне підтвердження ефективності запропонованого методу.

Апробація матеріалів дисертації. Основні положення та отримані наукові результати, що викладені в даній дисертації, пройшли апробацію через їх

презентацію на Міжнародній науковій конференції «Інформаційне суспільство: технологічні, економічні та технічні аспекти становлення (випуск 86)» у березні 2024 року.

Публікації.

Статті у наукових фахових виданнях України

[1] 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. Здобувач провів дослідження існуючих можливостей рефакторингу, що здійснюється із використанням інструментарію розробки мови програмування Swift, і здатність до їх параметризації, здійснив розробку одного сценарію рефакторингу шляхом написання інфраструктурного коду (містить власне логіку здійснення відповідної дії рефакторингу, а також код для представлення дії рефакторингу як частини контекстного меню), здійснив його перевірку і обробку отриманих результатів.

[2] Tkachuk, A., & Bulakh, B. (2022). Usage of formalized knowledge about source code for refactoring actions in Swift. *Technology Audit and Production Reserves, 2(68)*(6), 6–10. Здобувачем проведено аналіз існуючих пропозицій з вдосконалення процесу рефакторингу з використанням семантичної інформації. Здобувач запропонував метод виконання рефакторингу із використанням відокремленої бази знань, виконав формалізований опис методу, здійснив розробку прототипу та його тестування.

[3] Tkachuk, A., & Bulakh, B. (2023). Describing the knowledge about the source code using an ontology. *Infocommunication and Computer Technologies, 5*(1), 123-133. Здобувач здійснив формальний опис методу рефакторингу з використанням бази знань, що будується на основі онтології, запропонував використання правил для отримання якісно нових результатів, здійснив розробку програмного прототипу. Здобувач провів експерименти і описав отримані результати.

Структура дисертації. Дисертація складається з анотації, вступу, семи розділів, висновків, списку використаних джерел. Загальний обсяг дисертації становить 166 сторінок.

РОЗДІЛ 1. АВТОМАТИЗОВАНІ СИСТЕМИ РЕФАКТОРИНГУ

1.1 Поняття системи автоматизованого рефакторингу

Інструменти автоматизованого рефакторингу — це програмні утиліти, розроблені, щоб допомогти розробникам реструктурувати існуючий комп'ютерний код, не змінюючи його зовнішню поведінку. Ці інструменти спрямовані на покращення нефункціональних атрибутів програмного забезпечення, роблячи його більш зрозумілим, зменшуючи його складність і підвищуючи придатність до обслуговування. (Fowler, 1999)

Переваги автоматизованого рефакторингу:

- ефективність: ручний рефакторинг може зайняти багато часу. Автоматичні інструменти можуть швидко аналізувати та змінювати великі кодові бази;
- узгодженість: автоматизовані інструменти можуть однаково застосовувати ті самі зміни до кодової бази, забезпечуючи узгодженість;
- безпека: багато інструментів автоматизованого рефакторингу мають вбудовані процедури перевірки, щоб переконатися, що зміни не створюють помилок. (Mens & Tourwé, 2004)

Приклади інструментів автоматизованого рефакторингу:

- вбудовані інструменти IDE: інтегровані середовища розробки (IDE), такі як IntelliJ IDEA, Eclipse та Visual Studio, мають вбудовані можливості рефакторингу. Розробники можуть автоматично перейменовувати змінні, виносити методи або змінювати назви методів кількома клацаннями;
- браузери для рефакторингу: такі інструменти, як Smalltalk Refactoring Browser, надають потужні можливості для трансформації коду;
- спеціалізовані інструменти: деякі інструменти спеціалізуються на певних мовах або фреймворках. Наприклад, jscodeshift — це набір інструментів

для запуску великомасштабної модуляції коду JavaScript. (Murphy-Hill, Parnin, & Black, How we refactor, and how we know it, 2012)

Інструменти автоматизованого рефакторингу зазвичай включають або покладаються на автоматизований аналіз початкового коду. Перш ніж застосовувати рефакторинг, інструмент повинен «зрозуміти» структуру та семантику коду. Це розуміння досягається шляхом аналізу початкового коду.

Автоматичний аналіз початкового коду, який іноді називають статичним аналізом коду, перевіряє початковий код без його фактичного виконання. Мета полягає в тому, щоб визначити різні аспекти коду, такі як його структура, залежності та потенційні проблеми. Ця інформація є важливою для інструментів рефакторингу, щоб безпечно здійснювати зміни без створення нових дефектів. (Dig, Manzoor, Johnson, & Nguyen, 2008)

Фаулер (Fowler, 1999) наголошує на необхідності наявності набору тестів, які можна запустити, щоб переконатися, що рефакторинг не змінив спостережувану поведінку коду. Хоча він не заглиблюється в механізми автоматизованого аналізу початкового коду у своїй статті, важливість «розуміння» коду перед рефакторингом є очевидною без явних посилань на це в тексті.

Види аналізу початкового коду, що використовується при рефакторингу:

- аналіз синтаксису: інструменти рефакторингу аналізують початковий код для створення абстрактного синтаксичного дерева (AST). AST представляє структуру коду та формує основу для багатьох операцій рефакторингу;
- семантичний аналіз: окрім «розуміння» структури (синтаксису), інструмент також має «розуміти» значення (семантику) коду. Сюди входять такі речі, як інформація про тип змінних, області їх видимості і рівні доступу для методів;
- аналіз залежностей: перш ніж вносити зміни, інструмент має визначити залежності. Це гарантує, що зміна в одному місці не вплине негативно на інше місце в коді;

- розпізнавання шаблонів: деякі рефакторинги покладаються на розпізнавання конкретних шаблонів коду або антишаблонів. Після розпізнавання ці шаблони можна перетворити на більш бажані структури. (Johnson, Song, Murphy-Hill, & Bowdidge, 2013)

Для кращого розуміння властивостей і можливостей статичного аналізу розглянемо методи, які він використовує:

1. Аналіз на основі шаблонів: ця техніка зіставляє шаблони в кодовій базі з відомим набором проблемних шаблонів коду. Коли збіг знайдено, його позначають як потенційну проблему. Інструменти, які використовують цей метод, зазвичай мають базу даних відомих шаблонів помилок, які вони шукають.
2. Аналіз на основі потоку даних: цей метод переглядає потік виконання і потік даних програми, щоб зрозуміти, як дані переміщуються через нього. Це особливо корисно для пошуку вразливостей, пов’язаних із потоком даних, наприклад SQL-ін’єкцій або міжсайтового скриптингу (XSS), шляхом аналізу того, як дані переходять від ненадійних вхідних даних до конфіденційних операцій без належної перевірки чи очищення.
3. Аналіз на основі показників: цей метод зосереджений на вимірюванні різних показників, пов’язаних із кодом. Ці показники можуть стосуватися складності коду (наприклад, цикломатичної складності), розміру коду, щільності коментарів та інших атрибутів. Метрики можуть надати уявлення про «ремонтопридатність» і потенційні зони ризику в кодовій базі.
4. Семантичний аналіз: цей підхід розглядає значення і призначення коду. Створюючи абстрактне синтаксичне дерево (AST) або інше проміжне представлення, інструменти можуть оцінювати семантику конструкцій коду, визначаючи проблеми, які можуть бути пропущені методом зіставленням шаблонів.

5. Символьне виконання: у цій техніці змінні замінюються символьними значеннями, а потоки виконання коду оцінюються для визначення всіх можливих результатів. Цей метод може бути потужним у виявленні крайових випадків, хоча він може потребувати великого обсягу обчислень.
6. Аналіз забруднення: це особливий тип аналізу потоку, який відстежує контрольовані користувачем вхідні дані (або «зіпсовані» дані) через програму, щоб переконатися, що вони не досягають конфіденційних операцій без належної перевірки чи очищення.
7. Перевірка коду вручну: незважаючи на те, що автоматизовані інструменти можуть виявити багато проблем, доволі важко замінити людське судження. Перевірка коду вручну включає розробників, які перевіряють код, часто у спільній роботі, щоб виявити проблеми, які інструменти можуть пропустити.
8. Перегляд конфігурації: перевіряє файли конфігурації та параметри, пов’язані з кодовою базою, щоб переконатися, що вони відповідають найкращим практикам безпеки. Неправильні конфігурації можуть призвести до вразливості так само, як і проблеми на основі коду.
9. Аналіз схеми бази даних: якщо програма використовує базу даних, аналіз її схеми може надати уявлення про потенційні вразливості, як-от слабке шифрування, відсутність цілісності зв’язків або незахищені дозволи.
10. Аналіз сторонніх компонентів: фокусується на виявленні відомих вразливостей у сторонніх бібліотеках або компонентах, які використовує програма.
11. Перевірка синтаксису та стилю. Хоча перевірка відповідності стилю кодування та синтаксису не пов’язана безпосередньо з уразливістю, вона може покращити читабельність і зручність обслуговування, що може опосередковано зменшити кількість помилок.

Для ефективного статичного аналізу багато інструментів і платформ поєднують кілька методологій, щоб забезпечити повне охоплення. Варто також

зазначити, що статичний аналіз слід доповнювати динамічним аналізом та іншими методологіями тестування для багатосторонньої підтримки безпеки програмної системи.

1.2 Типи автоматизованих систем рефакторингу

1.2.1 Рефакторинг на основі правил

Рефакторинг на основі правил — це структурований метод покращення коду, який використовує попередньо визначені правила або шаблони для визначення областей коду, які можна змінити. Ці правила сформульовані для виявлення певних структур або code smells в коді. Після розпізнавання антишаблону або code smell система може автоматизованого застосувати перетворення цих структур у більш оптимальні форми.

Термін «запах коду» або code smell позначає шаблони або риси в коді, які вказують на потенційну основну проблему. Code smells не обов'язково вказують на помилковий код, але вказують на області, де може знадобитися рефакторинг для кращої ясності, дизайну чи зручності обслуговування.

Для кожного ідентифікованого «запаху коду» часто існує відповідне правило перетворення, яке пропонує кроки для його виправлення (Fowler, 1999).

Сучасні засоби рефакторингу зазвичай мають можливість автоматизованого виявлення запахів коду. Вони досягають цього шляхом аналізу початкового коду, як правило, за допомогою таких методів, як використання абстрактного синтаксичного дерева (AST) та інших методологій статичного аналізу. (Kerievsky, 2004)

Деякі приклади поширених «запахів коду» включають «Великий клас» – завеликий або переобтяжений клас, який свідчить про те, що класи намагаються виконувати занадто багато функцій і, можливо, стали перевантаженими. Типовим рефакторингом для цього «запаху» є поділ класу на кілька менших, більш спеціалізованих класів. Іншим прикладом є «довгий метод», де методи займають

надто багато рядків коду та намагаються впоратися з багатьма завданнями; рефакторинг полягає в сегментації методу на кілька коротких методів. (Mens & Tourwé, 2004)

Методи, що використовуються: зіставлення шаблонів, обхід AST, перетворення на основі шаблонів.

1.2.2 Рефакторинг на основі пошуку

Рефакторинг на основі пошуку використовує алгоритми оптимізації для вивчення різних варіантів рефакторингу та вибору тих, які покращують конкретні показники програмного забезпечення або досягають певних цілей. Цей підхід розглядає рефакторинг як проблему оптимізації, де метою є знайти найкращу комбінацію дій рефакторингу, яка максимізує (або мінімізує) один або більше критеріїв.

Основи такого рефакторингу:

- цільова функція: успіх рефакторингу на основі пошуку залежить від визначення хорошої цільової функції. Ця функція кількісно визначає якість потенційного рішення рефакторингу. Часто розглядаються кілька цілей, наприклад покращення читабельності коду, зменшення складності та підвищення модульності;
- простір пошуку: він складається з усіх можливих рішень рефакторингу. Враховуючи, що існує кілька способів рефакторингу кодової бази та кілька порядків, у яких можна застосовувати рефакторинг, цей простір є величезним і складним; (Boukadida, Kessentini, Bechikh, & Ben Said, 2015)
- алгоритми оптимізації: для навігації в просторі пошуку та пошуку оптимальних або близьких до оптимальних рішень рефакторингу можна використовувати різні алгоритми. Ці алгоритми працюють шляхом ітеративного дослідження, оцінки та вдосконалення потенційних рішень на основі визначеної цільової функції (функцій). (Ouni, Kessentini, Sahraoui, & Hamdi, 2013)

Алгоритми, що часто використовуються:

- генетичні алгоритми (ГА): GA надихаються процесом природного відбору. Вони представляють потенційні рішення рефакторингу як «індивіди» та розвивають їх з часом, використовуючи такі операції, як мутація, кросинговер і відбір для дослідження простору пошуку;
- симуляція відпалу: це ймовірнісний метод оптимізації, натхненний процесом відпалу в металургії. Це передбачає дослідження простору пошуку шляхом прийняття неоптимальних рішень час від часу, щоб уникнути локальних оптимумів;
- сходження на пагорб: прямий ітеративний метод оптимізації, коли алгоритм починається з довільного рішення та ітеративно вдосконалює його, вносячи невеликі зміни та оцінюючи їхній вплив. (Harman & Tratt, 2007)

1.2.3 Системи рекомендацій для рефакторингу

Рекомендаційні системи для рефакторингу (RSR) використовують методи зі сфери рекомендаційних систем, щоб запропонувати розробникам потенційні варіанти рефакторингу. Ці системи аналізують кодову базу, історичні дані, а іноді навіть поведінку розробників, щоб надати контекстно релевантні та корисні пропозиції рефакторингу.

Основні характеристики та аспекти:

- контекстна обізнаність: RSR часто враховують поточний стан коду, останні зміни та поточну увагу розробника, щоб надати своєчасні та відповідні контексту рекомендації;
- інтелектуальний аналіз історичних даних: аналізуючи репозиторії системи контролю версій, RSR можуть розпізнавати закономірності розвитку коду та пропонувати рефакторинг, який узгоджується з історією та траєкторією проекту;

- відгуки користувачів: деякі RSR дозволяють розробникам надавати відгуки щодо рекомендацій. Цей цикл зворотного зв’язку дозволяє системі вдосконалювати та покращувати свої пропозиції з часом;
- інтеграція із середовищами розробки: щоб бути ефективними та зручними для користувача, RSR часто інтегруються в інтегровані середовища розробки (IDE), що дозволяє розробникам застосовувати рефакторинг безпосередньо з рекомендацій. (Silva, Ramos, Valente, & Anquetil, 2016)

Типи рекомендацій щодо рефакторингу:

1. виявлення «запаху коду»: визначаючи code smells або антишаблони в коді, система може рекомендувати відповідні дії рефакторингу для їх вирішення; (Otaiby, Turki, & Touir, 2012)
2. рекомендації щодо шаблонів проектування: RSR можуть запропонувати запровадження або адаптацію шаблонів проектування на основі поточної структури та потреб коду; (Tsantalis & Chatzigeorgiou, 2009)
3. виявлення клонів коду: якщо виявлено повторюваний код (клони), система може порекомендувати об’єднати код за допомогою вилучення методів і створення класів.

Методи, що використовуються: машинне навчання, аналіз схожості коду, історичні дані проекту.

1.2.4 Трансформації, що зберігають поведінку

Дуже важливо, щоб рефакторинг не змінював поведінку програми. Деякі системи зосереджуються саме на цьому аспекті, гарантуючи, що набір пропонованих перетворень зберігає поведінку і не змінює семантику (Opdyke, Refactoring object-oriented frameworks. Ph.D. thesis, 1992).

Ключові аспекти трансформацій, що зберігають поведінку:

- власне, визначення рефакторингу: згідно визначення, — це процес реструктуризації існуючого комп’ютерного коду без зміни його зовнішньої

поведінки. Цей акцент на збереженні поведінки відрізняє рефакторинг від інших змін коду;

- тести як допоміжні механізми: одним із основних механізмів забезпечення збереження поведінки під час рефакторингу є наявність надійного набору тестів. До і після рефакторингу потрібно запустити тести, щоб переконатися, що поведінка системи залишається узгодженою;
- формальні методи: деякі просунуті підходи використовують формальні методи для забезпечення збереження поведінки. Це може включати математичні докази або перевірку моделі, щоб переконатися, що перетворення коду не викликають небажаних побічних ефектів. (Leino, 2010)

Приклади рефакторингу, що зберігає поведінку:

1. виокремлення методу: розбиття довгого методу на кілька менших. У той час як структура змінюється, загальна функціональність, надана оригінальним методом, залишається незмінною;
2. перейменування методу: зміна назви методу, щоб зробити його більш описовим. Це перетворення не впливає на логіку чи поведінку методу;
3. переміщення методу: перенесення методу з одного класу в інший, часто для покращення «зчепленості» (англ. cohesion) методів класу та/або послаблення зв’язності (англ. coupling) класів. Функціональність методу залишається незмінною. (Fowler, 1999)

1.2.5 Рефакторинг на основі сценаріїв або макросів

Рефакторинг на основі сценаріїв або макросів стосується використання сценаріїв або макросів для автоматизації та застосування послідовності операцій рефакторингу. Цей підхід дозволяє розробникам вказувати, зберігати та відтворювати користувацькі або повторювані перетворення (операції) рефакторингу, забезпечуючи узгоджені зміни у великих кодових базах або

кількох проектах. (Murphy-Hill, Parnin, & Black, How we refactor, and how we know it, 2012)

Ключові аспекти рефакторингу на основі сценаріїв або макросів:

- автоматизація та відтворюваність: основною перевагою використання сценаріїв або макросів для рефакторингу є можливість автоматизувати повторювані або складні операції, забезпечуючи послідовне застосування тих самих перетворень щоразу, коли виконується сценарій або макрос;
- здатність до налаштування (підлаштування): на відміну від стандартних інструментів рефакторингу, які пропонують попередньо визначені операції (наприклад, перейменування методу або вилучення класу), сценарії дозволяють розробникам визначати користувальницькі перетворення, адаптовані до конкретних потреб проекту або умов написання початкового коду;
- пакетні операції: рефакторинг за допомогою сценаріїв може бути особливо корисним для пакетних операцій, коли серію операцій рефакторингу необхідно застосувати до кількох файлів, класів або модулів;
- інтеграція із середовищами розробки: багато сучасних інтегрованих середовищ розробки (IDE) підтримують можливості створення сценаріїв або макросів, що дозволяє розробникам визначати та виконувати спеціальні операції рефакторингу безпосередньо в середовищі розробки. (Tokuda & Batory, 2001)

1.2.6 Багатоцільовий рефакторинг

Багатоцільовий рефакторинг — це підхід, коли під час процесу рефакторингу одночасно розглядаються кілька цілей або завдань. Традиційний рефакторинг зазвичай має єдину мету, таку як покращення читабельності або зменшення складності. Навпаки, багатоцільовий рефакторинг визнає, що якість програмного забезпечення багатогранна і що часто існують компроміси між різними атрибутами якості. Такий рефакторинг часто передбачає компроміси.

Багатоцільові підходи враховують кілька цілей одночасно, як-от максимізація читабельності при мінімізації розміру коду. (Seng & Burkhart, 2006)

Ключові аспекти багатоцільового рефакторингу:

- компроміси: коли розглядаються кілька цілей, вони іноді можуть конфліктувати одна з одною. Наприклад, оптимізація продуктивності може зменшити читабельність коду. Багатоцільовий рефакторинг шукає рішення, які врівноважують ці компроміси;
- оптимальність за Парето: загальна концепція, яка використовується в багатоцільовій оптимізації, оптимальність за Парето допомагає визначити рішення, у яких неможливо досягти покращення однієї цілі без погіршення іншої. Набір таких рішень називається фронтом Парето, який забезпечує різні оптимальні компроміси між цілями; (Harman & Tratt, 2007)
- складні простори пошуку: з кількома цілями простір пошуку (усі можливі рішення рефакторингу) стає набагато складнішим. Для ефективної навігації в цьому просторі часто використовуються розширені алгоритми пошуку та евристики; (Ouni, Kessentini, Sahraoui, & Hamdi, 2013)
- пріоритети цілей: у деяких випадках, хоча розглядаються кілька цілей, вони можуть бути визначені по-різному, залежно від потреб проекту або переваг зацікавлених сторін.

Усі описані підходи задовольняють різні потреби та контексти. Методи, що використовуються в системах рефакторингу, часто перетинаються та є взаємодоповнюючими, забезпечуючи як структурну, так і семантичну цілісність коду, що піддається рефакторингу.

1.3 Побудова систем автоматизованого рефакторингу

Автоматичні системи рефакторингу зазвичай будуються на основі кількох основних компонентів, які дозволяють аналізувати, «розуміти» та перетворювати початковий код:

- парсер: аналізатор читає початковий код і перетворює його в більш структурований формат – абстрактне синтаксичне дерево (AST). Ця деревоподібна структура представляє ієрархічну природу програмних конструкцій;
- абстрактне синтаксичне дерево (AST): як згадувалося, AST представляє структуру програми. Інструменти рефакторингу переглядають і змінюють це дерево, щоб здійснювати рефакторинг коду;
- таблиця символів і семантичний аналіз: хоча AST відображає структуру коду, таблиця символів та інші інструменти семантичного аналізу фіксують характеристики;
- механізм перетворення: цей компонент застосовує перетворення до AST на основі бажаного рефакторингу;
- генератор коду: після рефакторингу модифікований AST потрібно перетворити назад у початковий код. Це робиться за допомогою генератора коду;
- інтерфейс користувача: більшість інструментів рефакторингу мають інтерфейс, чи то інтерфейс командного рядка, чи графічний, за допомогою якого користувачі можуть вказувати бажані рефакторинги.

Існує велика кількість підходів до розробки систем для автоматизованого рефакторингу. Усі вони підпадають під узагальнену архітектуру, що описана вище. Системи на основі модифікації AST вимагають великих ресурсів для розробки і підтримки, а також працюють лише із однією мовою програмування. Саме для розв’язання цих проблем здійснюється пошук нових універсальних методів.

1.4 Переваги і недоліки автоматизованих систем рефакторингу

1.4.1 Переваги

1. **Ефективність**. В основі процесу розробки програмного забезпечення лежить потреба у швидкості та ефективності. Ручний рефакторинг, залежно від складності, може бути повільним і важким процесом. Автоматизовані інструменти рефакторингу є необхідною зручністю в цьому сценарії. Вони значно скорочують час, який витрачається на рефакторинг коду. Наприклад, перейменування методу, який використовується у великому проекті, може зайняти години вручну (включно із використанням засобів пошуку та заміни текстових фрагментів), але з автоматизованим інструментом це часто займає кілька секунд. Це прискорення допомагає не тільки швидко впроваджувати зміни, але й дозволяє розробникам експериментувати з різними варіантами реструктуризації без значних витрат часу.

   Крім того, у міру зростання розміру (кількість рядків коду) проектів програмного забезпечення взаємозалежність і складність коду також зростають. Робота з такими тонкощами вручну може не лише забрати багато часу, але й спричинити помилки. Автоматизовані інструменти рефакторингу вирішують ці складності за лаштунками, гарантуючи, що розробник може зосередитися на більш високорівневих аспектах. (Fowler, 1999)
2. **Консистентність**: послідовність у процесі написання коду є ключовою з різних причин. Це покращує читабельність коду, зменшує кількість помилок і гарантує, що члени команди можуть безперебійно працювати над проектом, не витрачаючи зайвого часу на розуміння різних стилів або шаблонів написання коду, використаних іншими розробниками. Автоматизовані інструменти рефакторингу – найкращі в цій області. Використовуючи попередньо визначені алгоритми та шаблони, ці інструменти гарантують послідовне виконання рефакторингу в усій кодовій базі.

Ця уніфікованість гарантує, що під час навігації по різних модулях або компонентах проекту розробники стикаються з узгодженим шаблоном написання коду. Така узгодженість різко зменшує когнітивне навантаження на розробників, дозволяючи їм зосередитися на вирішенні проблем, а не на розшифровці коду.

Крім того, у середовищах спільної роботи, де кілька розробників працюють над однією кодовою базою, підтримка узгодженої структури коду має першочергове значення. Автоматизовані інструменти гарантують, що незалежно від того, хто виконує рефакторинг, результат залишається однаковим. (Opdyke & Roberts, Extending objects to support multiple interfaces and access control, 1993)

3. Зменшення кількості помилок: однією з основних цілей будь-якого процесу розробки програмного забезпечення є зменшення кількості помилок, гарантуючи, що кінцевий продукт функціонує належним чином. Автоматизовані засоби рефакторингу відіграють важливу роль у цьому. Вони усувають елемент «людської помилки» з процесу рефакторингу, пропонуючи алгоритмічну точність. Це означає, що після того, як стратегія рефакторингу буде визначена, інструмент виконає її без помилок, таких як забуття деяких екземплярів, неправильне іменування або помилки при друці.

Наслідки такої мінімізації помилок є подвійними. По-перше, фаза тестування безпосередньо після рефакторингу проходить простіше, з меншою кількістю помилок, які виникають виключно через рефакторинг. По-друге, у довгостроковій перспективі підвищується загальна стійкість і надійність програмного забезпечення.

Однак важливо зазначити, що хоча ці інструменти мінімізують помилки в фактичному процесі рефакторингу, відповідальність за прийняття рішення про доцільність рефакторингу все ще лежить на

розробниках. (Murphy-Hill, Parnin, & Black, How we refactor, and how we know it, 2012)

4. Комплексний рефакторинг: у міру того, як проекти програмного забезпечення зростають у масштабах і складності, зростає і складність операцій рефакторингу, які необхідно застосувати. Деякі дії рефакторингу, такі як розбиття монолітного класу на кілька зв'язаних класів або реорганізація складної ієрархії успадкування, можуть бути складними, якщо виконувати їх самостійно. Автоматизовані інструменти рефакторингу мають перевагу в цих сценаріях. Вони розбивають складні рефакторинги на систематичні, покрокові операції, гарантуючи, що кожен крок підтримує цілісність коду.

   Завдяки наведеним можливостям, проведення значного (такого, що впливає на значну частину початкового коду) рефакторингу призводить до кращого, більш зручного в обслуговуванні коду в довгостроковій перспективі. (Dig, Manzoor, Johnson, & Nguyen, 2008)

5. Інтеграція з IDE: сучасні інтегровані середовища розробки (IDE) — це більше, ніж просто редактори коду; це комплексні інструменти, які допомагають протягом життєвого циклу розробки програмного забезпечення. Однією з значних переваг багатьох сучасних IDE є їх вбудовані можливості рефакторингу.

   Ця інтеграція означає, що розробникам не потрібно перемикати інструменти чи контексти під час рефакторингу — це стає невід'ємною частиною процесу кодування. Такі функції, як аналіз коду в реальному часі, пропозиції щодо потенційних рефакторингів і миттєвий попередній перегляд результатів рефакторингу, забезпечують плавний робочий процес розробки.

   Крім того, інструменти рефакторингу, вбудовані в IDE, гарантують, що вони завжди синхронізуються зі станом проекту, зменшуючи можливі невідповідності або проблеми з конфігурацією. (Mens & Tourwé, 2004)

1.4.2 Недоліки

1. Втрата семантики: один із основоположних принципів рефакторингу полягає в тому, що він не повинен змінювати зовнішню поведінку коду. Однак автоматизовані інструменти, незважаючи на свою точність, іноді дають збої в цьому відношенні. Вони можуть внести зміни, які на перший погляд здаються нешкідливими, але можуть призвести до зміни семантики. Наприклад, інструмент може перемістити метод до базового класу, припускаючи, що це нешкідлива операція. Однак у сценаріях, де діє динамічне зв'язування, це може призвести до помилкового виклику методу не в тому класі в ієрархії успадкування, таким чином змінюючи поведінку програми.

   Ще одна тонка область, де семантика може бути ненавмисно змінена, це багатопотокові середовища. Певні рефакторинги, якщо вони не виконуються з огляду на паралельне виконання операцій різними потоками, можуть спричинити умови гонки або взаємоблокування.

   Ці проблеми підкреслюють важливість розуміння базових алгоритмів інструментів і того, що на них не можна покладатися сліпо. (Xing & Stroulia, 2007)
2. Обмежене розуміння намірів «на високому рівні»: автоматизованим інструментам рефакторингу, незважаючи на їх прогрес, бракує людської інтуїції. Вони не здатні «розуміти» високорівневі цілі, закладені в конкретний код, ширше дивитись на ідеї в основі програмного забезпечення, як це може робити людина. Хоча вони можуть трансформувати код на основі попередньо визначених алгоритмів, вони не можуть зрозуміти намір розробника чи ширшу мету рефакторингу.

   Наприклад, якщо намір розробника полягає в тому, щоб спростити модуль для майбутньої розширюваності, інструмент може запропонувати рефакторинг, який зробить код лаконічним, але не обов'язково розширюваним.

Це підкреслює важливість людського судження в процесі рефакторингу. Інструменти можуть допомогти, але остаточні рішення мають відповідати ширшим цілям проекту. (Murphy-Hill & Black, Breaking the barriers to successful refactoring: Observations and tools for extract method, 2008)

3. Надмірна залежність розробників: хоча автоматизовані інструменти рефакторингу пропонують безліч переваг, існує прихована небезпека: розробники починають надмірно покладатися на них. Така надмірна довіра може призвести до відсутності критичного мислення про те, чому потрібні певні рефакторинги та які можуть бути їхні ширші наслідки.

   Наприклад, інструмент може запропонувати кілька рефакторингів, але не всі вони обов'язково будуть відповідати архітектурним цілям проекту. Сліпе прийняття всіх пропозицій може призвести до кодової бази, яка, незважаючи на технічно надійну, відрізняється від запланованих принципів дизайну.

   Надмірна довіра також може породити самовдоволення, коли розробники можуть пропустити ретельне тестування після рефакторингу, припускаючи, що зміни в інструменті завжди правильні. Це може призвести до невиявлених помилок або проблем з продуктивністю. (Beller, Gousios, & Zaidman, 2019)

4. Обмеження інструменту: жоден інструмент не є ідеальним, і автоматизовані інструменти рефакторингу не є винятком. Незважаючи на те, що вони можуть обробляти широкий спектр рефакторингів, завжди є крайові випадки або рефакторинги, що сильно залежать від контексту, які можуть виходити за межі можливостей інструменту.

   Деякі рефакторинги переплітаються з бізнес-логікою або покладаються на зовнішні системи, тому автоматизованим інструментам важко впоратися з ними без потенційних пасток.

Існує також проблема оновлення та обслуговування інструментів. Якщо інструмент рефакторингу не оновлюється регулярно, він може не підтримувати нові мовні функції чи передові практики, що з часом обмежить його корисність. (Kim, Cai, & Kim, 2011)

5. Накладні витрати на продуктивність: хоча автоматизовані засоби рефакторингу призначені для підвищення ефективності процесу розробки, вони не позбавлені накладних витрат. Деякі інструменти, особливо при роботі з великими кодовими базами або складними операціями рефакторингу, можуть потребувати значних обсягів ресурсів та часу.

   Це може проявлятися як відставання або затримки в IDE, збільшення споживання пам’яті або подовження часу виконання операцій рефакторингу.

   Ці накладні витрати на продуктивність, особливо якщо вони часті, можуть порушити роботу розробника, що призведе до розчарування та зниження продуктивності роботи самого розробника. (Goetz, et al., 2006)

1.5 Використання семантичних підходів у рефакторингу

Представлення та розуміння семантики в області обчислювальної техніки та розробки програмного забезпечення значно просунулися за останні роки. У той час як семантика за своєю суттю відноситься до значення або інтерпретації символу, виразу або фрази, в обчислювальній техніці вона часто стосується поведінки та виконання програм або значення чи вмісту оброблюваних даних. Розглянемо деякі сучасні підходи до роботи з семантикою або представлення її, що можуть бути використані для представлення семантики початкового коду під час здійснення статичного аналізу та виконання рефакторингу.

RDF (Resource Description Framework) — фреймворк опису ресурсів. RDF — це більше, ніж просто формат; це мова представлення структурованої інформації, яка описує дані у вигляді графу. Кожна частина даних у RDF

структурована як «трійка», яка складається з суб'єкта, предиката та об'єкта. (Berners-Lee, Hendler, & Lassila, 2001)

Щоб розширити можливості RDF, було представлено схему RDF (RDFS). RDFS дозволяє користувачам визначати класи та зв'язки між ними, роблячи дані більш значущими (з чітко визначеною структурою і вмістом) та прив'язаними до контексту. RDFS є способом додавання рівнів метаданих поверх RDF. (Berners-Lee, Hendler, & Lassila, 2001)

Коли існує потреба в більш виразному інструменті, важливо згадати про веб-орієнтовану мову опису онтологій (OWL). Це комплексна мова онтології (онтологія буде детальніше розглядатись у наступному розділі), яка дозволяє користувачам формулювати складну інформацію про концепції, зв'язки, властивості, обмеження тощо. За допомогою OWL можна визначати поняття в специфічному домені (предметній області), дозволяючи машинам не тільки «читати» дані, але також «розуміти» і міркувати про них. (Berners-Lee, Hendler, & Lassila, 2001)

Маючи визначену структуру даних, потрібні також і ефективні механізми здійснення запитів до структурованої інформації. SPARQL служить мостом для вирішення цієї потреби. SPARQL для RDF — це те ж, що SQL для традиційних баз даних. Це мова запитів, яка дозволяє користувачам отримувати дані, що зберігаються у форматі RDF, та обробляти їх. Незалежно від того, чи йдеться про фільтрування певної інформації, об'єднання даних із кількох джерел чи побудову нових графів даних, SPARQL надає інструментарій для ефективної обробки семантичних даних. (Prud'hommeaux & Seaborne, 2008)

Продовжуючи огляд засобів для роботи із семантикою, перейдемо тепер до області семантичного пошуку. На відміну від традиційних механізмів пошуку, які значною мірою покладаються на збіги ключових слів, семантичний пошук спрямований на розуміння – не лише слів, але наміру та контексту пошукового запиту. (Baeza-Yates & Ribeiro-Neto, 2011)

Подальшим розвитком семантики є програмна семантика. Ця предметна область спрямована на розуміння поведінки програм під час їх виконання. Існує кілька способів інтерпретації поведінки програми. Можна використовувати операційну семантику, яка забезпечує чітке покрокове пояснення операторів програми. Крім того, існує денотаційна семантика, яка використовує математичні функції для опису, або навіть аксіоматична семантика, яка працює на основі попередньо визначених аксіом або правил. Кожен підхід із власними унікальними властивостями покращує розуміння того, як програмне забезпечення поводиться та взаємодіє. (Nielson & Nielson, 1992)

Якщо розглядати сховища даних у контексті семантичних засобів, варто звернути увагу на семантичні бази даних. Виходячи за рамки жорстких структур традиційних баз даних, ці бази даних інкапсулюють значення або «семантику» даних. Тут вступають у дію такі технології, як RDF, які роблять представлення даних більш значущим і взаємопов’язаним. Це як перехід із світу ізольованих островів даних до більш згуртованого, взаємопов’язаного архіпелагу даних. (Hitzler & van Harmelen, 2010)

У своїй роботі Кесселі пропонує використовувати семантичний підхід до рефакторингу. Автором запропоновано 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 основі синтаксису і доведено його ефективність. (Kesseli, 2017)

Зрозуміло, що застосування семантичного підходу під час рефакторингу дозволяє абстрагуватись від конкретної реалізації і опиратись на значення (вміст) початкового коду в проекції застосування рефакторингу, що відноситься саме до рефакторингу вмісту, а не змісту. У цій роботі запропоновано використовувати онтології для представлення початкового коду і здійснення маніпуляцій з ним через проміжне представлення. Розглянемо онтології через призму OWL у наступному розділі.

Використання семантичного методу під час рефакторингу допомагає відмовитись від конкретних технічних деталей та зосередитись на змісті початкового коду, зокрема, коли йдеться про рефакторинг самого змісту.

Під час проведення дослідження було знайдено доволі небагато проектів, які використовують онтологію (чи інші семантичні представлення) як модель для представлення домену початкового коду об’єктно-орієнтованої мови програмування. Серед них утиліта для створення онтології на основі початкового коду програм Java (Ganapathy & Sagayaraj, 2011), утиліта для здійснення SPARQL запитів до початкового коду (Atzeni & Atzori, 2017), метод здійснення аналізу та рефакторингу початкового коду на основі семантичного представлення (Kesseli, 2017), а також представлення різнорідного початкового коду у вигляді онтології (Aguiar, Zanetti, & Souza, 2021).

Враховуючи проведений аналіз типів систем автоматизованого рефакторингу, їх побудову, а також переваги і недоліки, можемо сформулювати бажані властивості пропонованого методу, що відображено у Таблиця 1.

Таблиця 1. Бажані властивості пропонованого методу

Властивість	**Реалізація**	**Обґрунтування**
Використання правил	Рефакторинг на основі правил	Правило описує антишаблон для його успішного виявлення в початковому коді, а також спосіб усунення
Збереження поведінки	Трансформації, що зберігають поведінку і семантику	Зміна початкового коду, що не несе зміни поведінки чи семантики, але видаляє антишаблон – є прикладом успішного рефакторингу
Повторюваність дій	Рефакторинг на основі сценаріїв	Можливість повторювати одноманітні дії над різними кодовими базами

Властивість	Реалізація	Обґрунтування
Ефективність	Використання усталених підходів до реалізації системи автоматизованого рефакторингу	Використання запропонованого методу, що має властивості інших систем, а також пропонує новий підхід до покращення процесу рефакторингу позитивно впливатиме на динаміку розробки
База знань	OWL онтологія і модель початкового коду на її основі	Можливість відображати семантику та складні зв’язки початкового коду

ВИСНОВКИ ДО РОЗДІЛУ 1

Інструменти автоматизованого рефакторингу надають розробникам можливість систематично й ефективно покращувати якість коду. Вони стали невід'ємною частиною сучасних середовищ розробки програмного забезпечення, допомагаючи командам ефективніше підтримувати та розвивати кодові бази.

Ефективне функціонування інструментів автоматизованого рефакторингу значною мірою залежить від автоматизованого аналізу початкового коду. Аналіз дає необхідне уявлення про структуру та семантику коду, гарантуючи безпечність і послідовність рефакторингу.

Рефакторинг на основі правил забезпечує систематичне керівництво розробникам щодо вдосконалення структури та дизайну коду. Головне — розпізнавати проблемні шаблони (запахи коду) і знати, як їх вирішити за допомогою усталених методів рефакторингу.

Рефакторинг на основі пошуку розглядає процес вдосконалення коду як завдання оптимізації, використовуючи розширені алгоритми для пошуку найбільш вигідних перетворень. Сильна сторона підходу полягає в його здатності розглядати широкий діапазон потенційних рефакторингів і систематично визначати комбінації, які дають найкращі результати з точки зору заздалегідь визначених цілей.

Рекомендаційні системи для рефакторингу спрямовані на допомогу розробникам, надаючи інтелектуальні, контекстно-залежні пропозиції рефакторингу, які можуть підвищити якість коду та зручність обслуговування. Поєднуючи методи систем рекомендацій, статичного аналізу коду та інтелектуального аналізу даних, ці системи пропонують практичну інформацію, яка відповідає як безпосередньому, так і історичному контексту проекту програмного забезпечення.

Перетворення, що зберігають поведінку, мають вирішальне значення для підтримки довіри до процесу рефакторингу. Забезпечуючи незмінність

зовнішньої поведінки програмного забезпечення, розробники можуть з упевненістю вносити внутрішні зміни, щоб покращити дизайн системи, її читабельність і зручність обслуговування.

Рефакторинг на основі сценаріїв або макросів надає розробникам потужний спосіб автоматизувати, налаштовувати та послідовно застосовувати перетворення коду. Завдяки рефакторингу сценаріїв розробники можуть відповідати вимогам конкретного проекту, забезпечувати стандарти написання коду та швидко поширювати зміни у великих і складних кодових базах.

Багатоцільовий рефакторинг пропонує цілісний підхід до вдосконалення програмного забезпечення. Розглядаючи одночасно кілька атрибутів якості та цілей, розробники можуть досягти більш збалансованого та комплексного вдосконалення своїх кодових баз. Цей підхід визнає компроміси, властиві розробці програмного забезпечення, і намагається орієнтуватися в них у спосіб, який найкраще узгоджується з цілями та обмеженнями проекту.

Розширення механізмів статичного аналізу коду та рефакторингу за рахунок врахування семантики коду забезпечить більш повну та точну внутрішню модель представлення коду, що сприятиме точнішим та надійнішим результатам рефакторингу.

Сучасні підходи до роботи із семантикою відображають розуміння індустрією важливості збереження семантики під час рефакторингу. Оскільки галузь розробки програмного забезпечення продовжує розвиватися, ми можемо очікувати появи ще більш складних методів та інструмент.

РОЗДІЛ 2. ОНТОЛОГІЯ ТА ЇЇ ЗАСТОСУВАННЯ В БАЗАХ ЗНАНЬ

2.1 Мова OWL

2.1.1 Визначення OWL

Веб-орієнтовані мови опису онтологій (OWL) — це сімейство мов представлення знань або мов опису онтологій для створення баз знань. OWL призначена для використання програмами, яким потрібно самостійно автоматично інтерпретувати інформацію, а не просто вилучати та транслювати дані людям, які будуть їх інтерпретувати. Це стандартизована веб-мова, схвалена Консорціумом Всесвітньої павутини (W3C) для представлення складних знань про речі, групи речей і зв’язки між ними. (Berners-Lee, Hendler, & Lassila, 2001)

2.1.2 Поява OWL

Поява OWL є конвергенцією попередніх мов онтології та теорій логіки опису:

- логіка опису (DL): формалізми представлення на основі логіки, призначені для формального представлення концепцій онтології. Вони є теоретичною основою, на якій будується OWL; (Baader, Horrocks, & Sattler, Description logics as ontology languages for the semantic web, 2005)
- DAML і OIL: до створення OWL мова DARPA Agent Markup Language (DAML) і Ontology Inference Layer (OIL) були піонерами в розробці мов онтології. Злиття цих ініціатив породило DAML+OIL, яка згодом зіграла ключову роль у створенні OWL. (van Harmelen, Patel-Schneider, & Horrocks, 2001)

2.1.3 Версії OWL

Важливо розуміти версії та профілі OWL:

- OWL Lite: спрощений і менш виразний діалект, що спрощує обмеження та вимоги до класифікації;
- OWL DL: передбачає баланс між виразністю та обчислювальною спроможністю. Це гарантує, що всі проблеми «міркування» можуть бути вирішені алгоритмічно, хоча потенційно з високими обчислювальними витратами;
- OWL Full: найбільш виразний діалект, призначений для користувачів, які вимагають максимальної виразності, хоча й з ризиком обчислювальної невизначеності. (McGuinness & van Harmelen, 2004)

2.1.4 Профілі OWL 2

OWL 2, друга версія мови веб-онтології, була створена для усунення деяких обмежень і варіантів використання, не розглянутих оригінальним стандартом OWL. Щоб задовольнити потреби різних програм і вимоги до обчислень, OWL 2 було розділено на різні профілі. Ці профілі були розроблені, щоб запропонувати конкретні компроміси між експресивністю та обчислювальною складністю. Три найвідоміші профілі: OWL 2 EL, OWL 2 QL і OWL 2 RL. (Motik, Patel-Schneider, & Parsia, OWL 2 web ontology language structural specification and functional-style syntax (second edition), 2012)

1. OWL 2 EL (профіль EL++): OWL 2 EL базується на сімействі логік опису EL++. Його дизайн зосереджений на потребі в онтологіях, які можуть представляти величезну кількість ієрархії класів або властивостей. Ключові характеристики: дозволяє екзистенціальну квантифікацію, але не універсальну квантифікацію; дозволяє ієрархію та композицію ролей; обмежене використання обмежень потужності. Пропонує поліноміальне міркування (reasoning) в часі, що є важливим для онтологій із сотнями тисяч класів. (Baader, Brandt, & Lutz, Pushing the EL envelope, 2005)

2. OWL 2 QL (профіль мови запитів) створено для додатків, які потребують ефективної відповіді на запити для великих наборів даних. Його базова логіка є підмножиною сімейства логіки опису, відомої як DL-Lite. Ключові характеристики: зосереджується на прямих бінарних зв'язках і простих ієрархіях класів; дозволяє кваліфіковану та обмежену екзистенціальну квантифікацію; не підтримує обмеження потужності. Профіль розроблено для додатків, де в центрі уваги – доступ або інтеграція даних, керована онтологією. Гарантує складність L (LogSpace) по обсягу пам'яті для відповідей на запити, що робить можливою роботу з великими наборами даних. (Calvanese, Giacomo, Lembo, Lenzerini, & Rosati, 2007)
3. OWL 2 RL (профіль мови правил) розроблено для областей, які вимагають масштабованих міркувань без необхідності повної логічної виразності. Він обґрунтований фрагментом логіки опису DLP. Ключові характеристики: дозволяє використовувати обмежений набір конструкторів класів; надає обмежену підтримку для ієрархії властивостей і характеристик; може бути реалізовано за допомогою систем на основі правил. Підходить для додатків, де аргументація на основі правил є центральною. Може бути зіставлено з такими мовами правил, як SWRL (мова правил семантичної мережі). Розроблено для забезпечення поліноміальної складності процесу «міркування» для більшості стандартних висновків. (Motik, et al., 2009)

2.1.5 Основні поняття та особливості OWL

1. Класи: вони представляють категорії. Наприклад, в онтології програмування можна виділити класи «Функція», «Змінна» і «Клас».
2. Індивіди: це екземпляри класів. Використовуючи онтологію програмування як приклад, «calculateSum» є ідентифікатором екземпляру класу «Функція».
3. Властивості: визначають відносини між класами та між окремими індивідами. Два основних типи:

a. Властивості об’єкта: визначають зв’язки між індивідами. Наприклад, властивість «callsFunction» може пов'язувати одну «Функцію» з іншою «Функцією», описуючи той факт, що одна функція викликає іншу.
b. Властивості типу даних: пов’язують індивіда зі даними. Наприклад, властивість «hasLineNumber» може пов'язати «Функцію» з цілим числом, що представляє її номер рядка у початковому коді.

4. Характеристики властивостей:
    a. Функціональність: властивість може мати щонайбільше одне значення для даної особи. Наприклад, функція може мати властивість «returnsDataType», яка вказує тип, який вона повертає.
    b. Обернена функціональність: якщо два індивіди пов’язані властивістю з іншим індивідом, то два оригінальні індивіди мають бути ідентичними.
    c. Транзитивність: якщо властивість «callsFunction» пов’язує функції A з B і B з C через виклики, тоді функція A також опосередковано викликає функцію C (властивість буде пов’язувати функції A і C).
    d. Симетричність: якщо функція A викликає функцію B, це не обов’язково означає, що B викликає A, тобто властивість «callsFunction» не є симетричною. Таким чином, це може рідше використовуватися в контексті початкового коду. (McGuinness & van Harmelen, 2004)
5. Аксіоми: це твердження в онтології. Приклади включають аксіоми підкласу (клас «PublicFunction» є підкласом «Function») або аксіоми еквівалентності («StaticMethod» еквівалентний «StaticFunction»).
6. Підклас і еквівалентність: в OWL можна визначити, що один клас є підкласом іншого або що два класи є еквівалентними. Наприклад, «Метод» в об’єктно-орієнтованому програмуванні може бути еквівалентним «Функції» в процедурному програмуванні. (Gruber, 1993)

7. Міркування (reasoning): враховуючи аксіоми та факти в онтології, автоматизовані системи виводу можуть додавати нові факти. Наприклад, якщо всі «PublicMethods» доступні ззовні класу, а метод «getDetails» є «PublicMethod», система виводу може зробити висновок, що «getDetails» доступний ззовні свого класу.
8. Виразність: можна використовувати різні варіанти OWL залежно від бажаного рівня деталізації або складності.
9. Анотації: Дозволяють додавати метадані до різних елементів в онтології, такі як коментарі чи інші описи фрагмента коду або його призначення.
10. Простори імен: у контексті програмування простори імен можна використовувати для розрізнення різних бібліотек, модулів або пакетів в онтології. (Horrocks, Patel-Schneider, & van Harmelen, From SHIQ and RDF to OWL: the making of a Web Ontology Language, 2003)

2.1.6 Семантика та логічне виведення в OWL

Семантика в OWL:

В OWL семантика надає значення символам у онтологіях OWL, роблячи їх інтерпретованими. У контексті початкового коду це може представлятися як розуміння ролей та поведінки різних елементів коду.

В OWL все базується на її формальній семантиці. Необхідно сприймати це як основні граматичні правила, які визначають структуру та функціональність мови програмування. Семантика корениться в описовій логіці, типі логіки, що використовується для представлення онтологічних структур.

Суть семантики OWL можна вловити у її інтерпретації класів та властивостей. Клас в онтології програмування може представляти «Метод», і його семантика визначатиме набір екземплярів коду, які визнаються «Методами». Аналогічно, властивість може представляти відношення «має-параметр» між функціями та їх параметрами. (McGuinness & van Harmelen, 2004)

Логічне виведення в OWL:

Саме в логічному виведенні OWL дійсно проявляє свій потенціал. Використовуючи логічне виведення, можна отримати неявну інформацію з явно вказаних фактів.

Уявімо онтологію, де докладно описано відношення між різними функціями в кодовій базі (Рисунок 1).

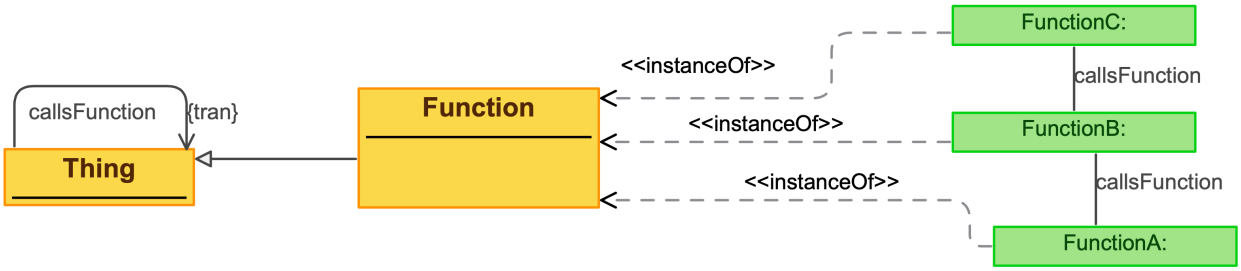

Рисунок 1. Приклад відношень між функціями

Якщо онтологія стверджує, що «FunctionA викликає FunctionB» та «FunctionB викликає FunctionC», тоді можна зробити логічний висновок, що «FunctionA опосередковано викликає FunctionC» (Рисунок 2). (Horrocks, Patel-Schneider, & van Harmelen, From SHIQ and RDF to OWL: the making of a Web Ontology Language, 2003)

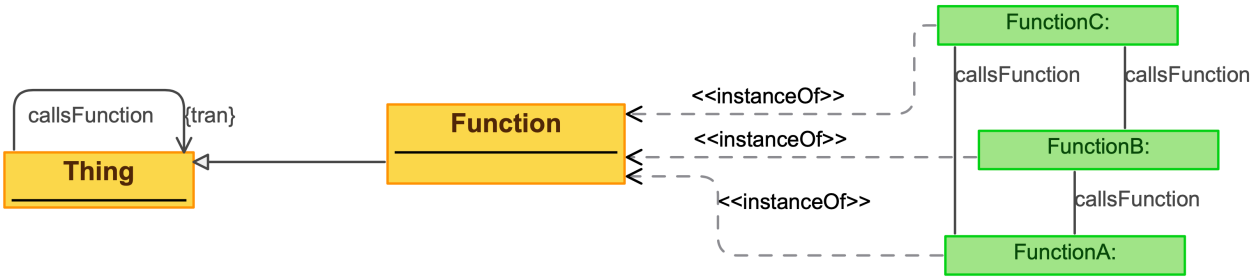

Рисунок 2. Відношення після виконання «міркування»

Логічні висновки також відіграють ключову роль у забезпеченні цілісності онтології, перевіряючи її на консистентність (цілісність, несуперечливість). Якщо в онтології є конфліктуючі висловлювання, такі як

«FunctionA приватна» та «FunctionA публічна», логічне виведення вказує на це як на неконсистентність.

Більше того, за допомогою логічного виведення можна організовувати або класифікувати сутності на основі їх відношень та визначень. Наприклад, у контексті початкового коду, клас «PublicFunction» визначений як функція, доступна поза своїм модулем (Рисунок 3).

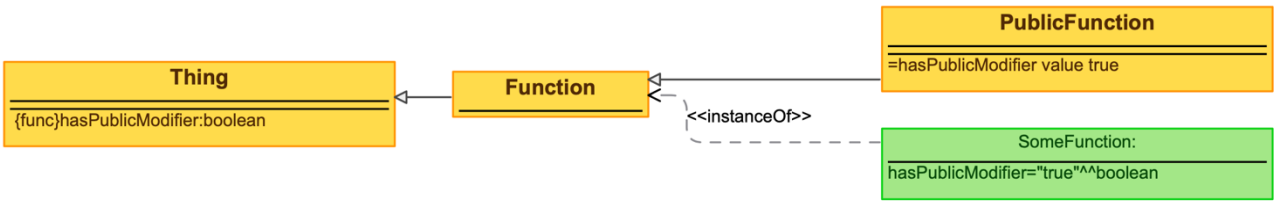

Рисунок 3. Початкова класифікація

За допомогою логічного виведення можна групувати всі функції під класом «PublicFunction» (Рисунок 4). (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2007)

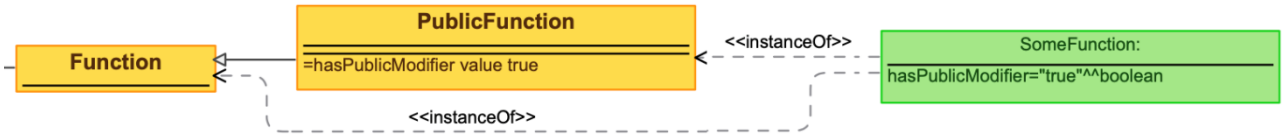

Рисунок 4. Здійснена класифікація після міркування (частину онтології пропущено)

Таким чином, логічне виведення відіграє одну з ключових ролей у «розумінні» початкового коду і встановлення неявно вказаних у ньому зв’язків.

2.1.7 Виклики та критика онтологій

OWL значно набула популярності у світі створення онтологій та є ключовим компонентом ініціативи семантичного вебу. Проте її шлях не був без перешкод та критичних зауважень.

Однією з основних критик OWL є її власна складність. Висока виразність OWL, особливо у її варіантах OWL DL та OWL Full, може відштовхувати потенційних користувачів. Хоча ця виразність пропонує багаті можливості моделювання, водночас вона може робити мову важкою для ефективного використання. Більше того, ця складність призводить до обчислювальних викликів. Зі збільшенням виразності логічне виведення в онтології може ставати ресурсоємкою задачею, призводячи до проблем з продуктивністю. (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2007)

Іншим питанням є масштабованість. Зі зростанням розміру онтологій час, необхідний для логічного виведення над ними за допомогою OWL, може збільшуватися. Це особливо помітно з великими онтологіями, що містять мільйони записів.

Крім того, інтеграція систем на основі OWL зі старими системами, які спочатку не були розроблені для інтеграції онтології або OWL, не завжди є однозначною (Smith, et al., 2005).

OWL, одночасно з її можливостями виразності, також має певні обмеження. Певні шаблони є складними або навіть неможливими для представлення в OWL. Наприклад, включення тимчасових або ймовірнісних знань не є тривіальними Крім того, існує змінність у продуктивності систем логічного виведення на основі конкретних конструкцій, використаних в онтології. Вони можуть різко уповільнити час роботи таких систем, ставлячи виклики ефективності. (Allemang & Hendler, 2011)

2.2 Системи логічного виведення

2.2.1 Поняття системи логічного виведення

Термін «reasoner» (система логічного виведення) в контексті OWL відноситься до програмного засобу або системи, яка може робити логічні висновки з набору фактів або аксіом в онтології.

Основні функції систем логічного виведення включають:

- перевірка на узгодженість: визначення того, чи суперечать один одному твердження в онтології;
- класифікація: виведення ієрархічних відносин, які не були явно вказані;
- реалізація: призначення індивідів їхнім найспецифічнішим класам на основі аксіом онтології;
- перевірка на виводимість: визначення, чи випливає конкретне твердження логічно з даної онтології, а разом і виявлення помилкових тверджень;
- підсумовування: перевірка, чи є одне поняття більш загальним, ніж інше. (Horrocks, et al., 2004)

2.2.2 Алгоритми, що використовуються в системах логічного виведення

Системи логічного виводу в OWL використовують різні алгоритми для виконання завдань, таких як перевірка на узгодженість, класифікація та реалізація. Вибір алгоритму та деталі його впровадження часто визначають ефективність, масштабованість та конкретні можливості системи. Розглянемо деякі ключові алгоритми та техніки, що використовуються в системах логічного виводу в OWL.

2.2.2.1 Алгоритм таблиці

Алгоритми таблиць – це основний метод у автоматизованому міркуванні, зокрема для модальних логік та логік опису (DL), які лежать в основі OWL. Основна ідея алгоритмів таблиць полягає в тому, щоб перевірити задовільність формули, намагаючись побудувати для неї модель (або таблицю).

Основи методу таблиці:

Метод таблиці починає свою роботу із заперечення формули для тестування і потім намагається побудувати модель для цього заперечення. Якщо модель для заперечення не може бути знайдена, тоді оригінальна формула є вірною.

Кроки алгоритму таблиці для логік опису:

1. **Ініціалізація**: починається робота з набору формул (зазвичай, понять з онтології). Кожна формула представляє твердження, яке потрібно задовільнити.
2. **Розширення**: застосовуються правила розширення до початкового набору, щоб згенерувати нові формули, фактично розширюючи набір. Цей процес може генерувати точки розгалуження, що призводить до кількох можливих наборів формул.
3. **Закриття**: якщо в будь-якій гілці таблиці виявляється суперечність (невідповідність), ця гілка закривається.
4. **Завершення**: якщо усі гілки закриваються, формула (або онтологія) є незадовільною. Якщо будь-яка гілка залишається відкритою, формула є задовільною, і відкрита гілка представляє модель.

Математична аргументація:

Суть правильності методу таблиць ґрунтується на двох принципах:

1. **Коректність**: якщо алгоритм таблиці знаходить модель, то формула дійсно є задовільною.
2. **Повнота**: якщо формула є задовільною, алгоритм таблиці знайде модель.

Ці принципи гарантують, що метод таблиці є і точним (він не стверджуватиме, що формула є задовільною, якщо це не так), і вичерпним (він не проґавить задовільну формулу). (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2007)

Правила розширення:

Правила розширення визначають, як будується таблиця. Для логік опису точні правила залежать від виразності логіки. Деякі базові правила для загальних конструкцій DL включають:

- **Кон'юнкція (⊓)**: Якщо $C \sqcap D$ є у наборі, додати C та D до набору.
- **Диз'юнкція (⊔)**: Якщо $C \sqcup D$ є у наборі, тоді розгалужити: додати C до одної гілки та D до іншої.

- **Екзистенціальний квантор (∃)**: Якщо $\exists R.C$ є у наборі та немає жодного індивіда у наборі, пов'язаного з R до певного поняття C, ввести нового індивіда та ствердити, що він пов'язаний з R і є екземпляром C.
- **Універсальний квантор (∀)**: Якщо $\forall R.C$ та $R(a, b)$ є у наборі, і b не відомо як екземпляр C, тоді додати C до b. (Blackburn, de Rijke, & Venema, 2001)

2.2.2.2 Алгоритм гіпертаблиці

Алгоритм гіпертаблиці є розширенням і оптимізацією стандартного методу таблиці для автоматизованого виведення в описових логіках (DLs).

Основна ідея:

Стандартний метод намагається побудувати модель даної онтології шляхом послідовного розширення її понять. Процес може включати недетерміновані вибори, які призводять до розгалужень в таблиці, де кожна гілка представляє різну потенційну модель. Процес виведення продовжується шляхом розширення цих гілок до тих пір, поки не знаходиться протиріччя або подальше розширення неможливе.

Алгоритм гіпертаблиці намагається зменшити ці точки розгалуження, групуючи певні етапи розширення. Саме звідси і походить термін «гіпер». (Baader, Calvanese, McGuinness, Nardi, & Patel-Schneider, 2007)

Основні характеристики гіпертаблиці:

1. групове розширення: на відміну від стандартного алгоритму, де кожне правило розширення застосовується окремо, алгоритм гіпертаблиці групує певні правила разом і застосовує їх за один крок.
2. оптимізоване блокування: щоб запобігти нескінченним моделям, попередні алгоритми використовують техніку, яка називається блокуванням. Алгоритм гіпертаблиці оптимізує цей процес, забезпечуючи більш ефективне визначення та обробку заблокованих гілок.
3. виявлення протиріч: протиріччя (конфлікти) можна виявляти більш ефективно, враховуючи груповий характер розширень. Якщо в одній гілці

виявляється протиріччя, це може швидко призвести до виявлення протиріч у пов'язаних гілках.

Алгоритм гіпертаблиці зберігає властивості коректності та повноти, оптимізуючи їх. Основний математичний аргумент полягає в тому, що групові розширення алгоритму гіпертаблиці семантично еквівалентні послідовності окремих розширень у стандартному методі таблиці. Тому будь-яка модель, знайдена алгоритмом гіпертаблиці, також є моделлю онтології, і навпаки. (Motik, Shearer, & Horrocks, Hypertableau reasoning for description logics, 2009)

2.2.2.4 Алгоритм наближення за поліноміальний час

Алгоритми наближення за поліноміальний час – це клас алгоритмів, що намагаються надати наближені рішення для NP-складних оптимізаційних проблем за поліноміальний час. Оскільки знаходження точних рішень для багатьох з цих проблем є обчислювально непрохідним, алгоритми наближення пропонують компроміс між якістю рішення та обчислювальною ефективністю.

Сутність NP-складних проблем:

Проблема оптимізації вважається NP-складною, якщо для будь-якого можливого алгоритму, що вирішує проблему, час, необхідний для вирішення проблеми, зростає не поліноміально (а, наприклад, експоненційно) з ростом розміру вхідних даних. Для багатьох практичних цілей такий експоненційний ріст обчислювального часу робить проблему непрохідною для вхідних даних великої розмірності.

Наближення за поліноміальний час:

Враховуючи виклик NP-важких проблем, метою стає знаходження наближеного рішення, а не точного. Конкретно мета алгоритмів наближення за поліноміальний час – знайти рішення, яке гарантовано знаходиться в межах оптимального рішення, і робити це за поліноміальний час. (Vazirani, 2003)

Математично, для проблеми мінімізації, алгоритм є алгоритмом α-наближення, якщо для кожного вхідного випадку *I*:

$$ALG(I) \leq \alpha \times OPT(I) \quad (1)$$

А для проблеми максимізації:

$$ALG(I) \leq \frac{1}{\alpha} \times OPT(I) \quad (2)$$

Де:

- $ALG(I)$ – рішення, отримане алгоритмом наближення для вводу *I*.
- $OPT(I)$ – оптимальне рішення для вводу I*I*.
- $\alpha \geq 1$ – фактор наближення.

Фактор наближення α забезпечує гарантію якості рішення: чим ближче α до 1, тим ближче рішення до оптимуму за критеріями (1) чи (2).

2.2.2.5 Двійкові діаграми рішень

Бінарна діаграма рішень (BDD) представляє булеву функцію. Булева функція *f* – це функція з $\{0,1\}^n \rightarrow \{0,1\}$, де *n* – це кількість булевих змінних. BDD – це спрямований ациклічний граф, який має:

- унікальний початковий (кореневий) вузол;
- кінцеві вузли, позначені 0 та 1;
- внутрішні вузли, асоційовані з булевими змінними;
- кожен внутрішній вузол має двох нащадків. (Bryant, 1986)

Існує кілька BDD, які можуть представляти ту саму булеву функцію. Щоб зробити BDD ефективними та канонічними, застосовуються два правила зниження:

- **об'єднання еквівалентних вузлів:** якщо два вузли представляють одну й ту ж функцію, їх об'єднують у єдиний вузол.
- **видалення надлишкових вузлів:** якщо обидва нащадка внутрішнього вузла є однаковими, вузол обходять.

BDD представляє рекурсивне декомпонування булевої функції на основі розширення Шеннона (3):

$$f(x_1, x_2, \dots, x_n) = x_1 \cdot f(1, x_2, \dots, x_n) + \overline{x_1} \cdot f(0, x_2, \dots, x_n) \quad (3)$$

2.2.2.6 Модульна декомпозиція

Модульна декомпозиція полягає в ідентифікації частин (або модулів) онтології, які можна «розуміти», підтримувати і обґрунтовувати незалежно від решти онтології. Модуль OWL для набору термінів (або сутностей) – це підмножина онтології, яка зберігає значення цих термінів у межах онтології. (Grau, Horrocks, Kazakov, & Sattler, 2007)

З математичної точки зору, маючи онтологію O і набір термінів S, модуль M для S відносно O такий, що для кожної онтології O', яка є дискретною від O (4) для кожної аксіоми α, яка включає лише терміни з S.

$$O \cup O' \vDash \alpha \Leftrightarrow M \cup O' \vDash \alpha \quad (4)$$

Це означає, що M зберігає значення S незалежно від того, як решта онтології може бути розширена, за умови, що вона не перетинається з O.

Існують різні методи і техніки для модульної декомпозиції в OWL, зокрема: синтаксичні методи, семантичні методи.

Переваги:

- **масштабованість**: великі онтології можуть бути важкими для програмного забезпечення та людей. Розділяючи їх на менші модулі, інструменти та люди можуть зосередитися на відповідних частинах, що веде до кращої продуктивності та розуміння.
- **повторне використання**: модулі можна повторно використовувати в різних онтологіях, що сприяє обміну та інтеграції онтологічних знань.
- **обслуговуваність**: зміни можна обмежити конкретними модулями, зменшуючи ризик ненавмисних глобальних наслідків.

2.2.3 Популярні системи логічного виведення

Існує декілька відомих систем логічного виведення для OWL. Вони відрізняються своїми можливостями, стратегіями оптимізації та конкретними профілями OWL, які вони підтримують. Ось деякі з популярних:

- Pellet: це одна з найвідоміших систем логічного виведення для OWL. Вона підтримує логічне виведення для OWL-DL на основі алгоритму таблиць та може використовуватися для завдань, таких як перевірка на узгодженість, класифікація та реалізація. (Sirin, Parsia, Grau, Kalyanpur, & Katz, 2007)
- HermiT: відомий своїми методами оптимізації та здатний працювати з онтологіями, які інші системи логічного виведення можуть вважати складними. (Glimm, Horrocks, Motik, Stoilos, & Wang, 2014)
- FaCT++: базується на алгоритмі таблиці та написаний на C++. Він є нащадком попереднього reasoner FaCT. (Tsarkov & Horrocks, 2006)

2.3 Доповнення до OWL

Декілька інструментів було розроблено для розширення та використання можливостей OWL. Ці інструменти, як правило, надають функціональність для редагування, обґрунтування, запитів та управління онтологіями OWL. Protégé є провідним редактором онтологій, тоді як Pellet, HermiT, FaCT++ є відомими системами логічного виводу. OWL API сприяє програмній маніпуляції онтологіями, а SPARQL дозволяє робити запити до наборів даних OWL. Інструменти, такі як ROBOT, OntoBee, OBO-Edit, Cellfie, DL Query Tab, OWLGrEd та VOWL, додатково покращують розробку, управління та візуалізацію онтологій.

2.3.1 SWRL

SWRL (Semantic Web Rule Language) – це мова правил. Вона надає можливість визначати правила, додатково до визначення онтології у OWL. Об'єднуючи OWL та SWRL, можна робити висновки про екземпляри онтології та отримувати нову інформацію на основі існуючих даних. (Horrocks, та ін., 2004)

1. **Основи**:
    - Правила SWRL складаються з антецеденту (умови) та консеквенту (висновку, якщо умови виконуються).

- Як антецедент, так і консеквент складаються з набору атомів. Атом може бути предикатом класу, предикатом властивості, предикатом «такий же як», предикатом «відрізняється від» або вбудованим предикатом. (Antoniou & Van Harmelen, 2009)

2. **Атоми в SWRL**:
   - **Класовий атом**: Class(C, x) – означає, що індивід x є типу C.
   - **Атом властивості**: P(x, y) стверджує, що існує властивість P між x та y.
   - **Атоми «такий же як» та «відрізняється від»**: sameAs(x, y) і differentFrom(x, y) – використовуються для вказівки на те, що два екземпляри є однаковими або різними.
   - **Атом діапазону даних**: DataRange(D, x) – вказує, що значення даних x належить діапазону даних D.
   - **Вбудований атом**: використовується для вираження умов, що стосуються вбудованих операцій, таких як математичні обчислення, операції з рядками тощо.
3. **Синтаксис**:
   - SWRL використовує синтаксис: «Antecedent → Consequent».
   - Наприклад: **Person(?x) ^ hasAge(?x, ?y) ^ swrlb:greaterThan(?y, 18) -> Adult(?x)** Це правило можна читати так: «Якщо ?x є особою і ?x має вік ?y, який більший за 18, то ?x є дорослим».
4. **Вбудовані функції**:
   - SWRL включає набір вбудованих функцій, переважно для операцій зі простими типами даних. До них належать математичні, рядкові, дати та інші.
   - Наприклад, вбудовані функції: **swrlb:add**, **swrlb:multiply**, **swrlb:stringConcatenate** тощо.
5. **Виразність та обмеження**:

- Хоча SWRL розширює виразність OWL, вона дотримується гарантій прийнятності, тобто будь-який висновок, який можна зробити на основі набору правил та фактів, можна обчислити за скінченний час.
- SWRL не може виразити правила, такі як: «Для кожного класу існує властивість», тому що це порушило б гарантії прийнятності.
- SWRL має обмеження монотонності, що не дозволяє здійснювати видалення аксіом чи індивідів з онтології.
- SWRL, так як і OWL, підтримує обмеження «відкритого світу».

6. **Інтеграція з системами логічного виводу онтології**:
   - Існує кілька систем логічного виводу онтології (наприклад, Pellet, HermiT), які підтримують роботу з правилами SWRL.
   - Використовуючи ці системи, можна робити висновки на основі існуючих фактів, поєднаних з правилами SWRL.

2.3.2 SQWRL

SQWRL (Semantic Query-Enhanced Web Rule Language) – це розширення мови правил SWRL. Воно вводить можливість формулювання запитів до онтологій OWL, будуючись на основі правил SWRL. (O'Connor & Das, 2009)

1. **Мета**: у той час як SWRL в основному використовується для визначення правил над онтологіями, SQWRL зосереджується на запитах до онтологій OWL, використовуючи синтаксис, схожий на правила.
2. **Базова структура**: запит SQWRL має структуру, схожу на правила SWRL. Він складається з антецеденту (умови) і консеквенту (висновку). Антецедент вказує умови для зіставлення з онтологією, а консеквент визначає, що повертати як результат.
3. **Синтаксис**:
   - Як і SWRL, синтаксис SQWRL є: «Антецедент -> Консеквент», але з акцентом на витягнення та вибір даних, а не на ствердження.

- Наприклад: **Person(?p) ^ hasAge(?p, ?a) ^ swrlb:greaterThan(?a, 21) -> sqwrl:select(?p)** Цей запит можна інтерпретувати так: «Для кожного індивіда **?p**, який є **Person** і має вік **?a** більший за 21, вибрати (або повернути) **?p**».

4. **Оператори та вбудовані функції SQWRL**: SQWRL вводить декілька вбудованих операцій специфічно для запитів, зокрема:
   - **sqwrl:select** для вказівки, які змінні повертати.
   - агрегаційні операції, такі як **sqwrl:count**, **sqwrl:sum** тощо.
   - операції з колекціями, такі як **sqwrl:makeSet**, **sqwrl:size** тощо.
   - операції порядку для сортування результатів, наприклад, **sqwrl:orderBy**.
5. **Виконання**: запити SQWRL, як правило, виконуються за допомогою систем виводу онтології, які підтримують SWRL, часто інтегрованих у середовища або інструменти розробки онтологій.
6. **Переваги та обмеження**:
   - **Переваги**: Надає механізм запиту на основі правил для онтологій OWL, дозволяючи виконувати складні запити, які можуть використовувати виразність SWRL.
   - **Обмеження**: Як і в SWRL, потрібна обережність, щоб забезпечити прийнятність та тривалість, особливо при роботі зі складними онтологіями та великими наборами даних. (Connor, Shankar, Tu, Parrish, & Das, 2009)

ВИСНОВКИ ДО РОЗДІЛУ 2

OWL відіграє ключову роль у сучасній семантичній мережі (semantic web), дозволяючи детально представляти складні знання. ЇЇ походження з описових логік забезпечує її надійність, але пов’язана з нею обчислювальна складність вимагає ретельного проектування та реалізації.

Незважаючи на те, що OWL є потужною, вона також складна, і пов’язана з нею крива навчання є крутою. Крім того, зі збільшенням складності завдань міркування (reasoning) вимоги до обчислювальних ресурсів можуть стати непомірними або час обчислень може стати неприйнятним.

За останні роки семантична мережа та пов'язані технології так і не змогли розкрити свій потенціал, і мові OWL все ще бракує певних удосконалень та розширень. Втім, і наявний інструментарій має достатню цінність для його застосування у вирішенні широкого кола задач поза межами семантичного веб, зокрема – для рефакторингу програмного коду.

Кожен профіль OWL 2 відповідає окремому набору вимог і обчислювальних проблем. Хоча вони пропонують меншу експресивність порівняно з повним стандартом OWL 2, вони забезпечують перевагу більш ефективного міркування, що робить їх практичними для конкретних областей застосування.

У сфері онтології, класи є абстрактними представленнями понять, подібно до «множин». Властивості представляють відносини, або між класами, або між класом та значеннями примітивних типів даних. Індивіди – це конкретні екземпляри класів. Аксіоми – це фундаментальні істини, які визначають структуру та поведінку онтології. Анотації пропонують метадані для елементів онтології. Виразні конструкції в онтологіях, зокрема в OWL, включають передові засоби моделювання, такі як обмеження властивостей та еквівалентність.

Коли необхідно проаналізувати велику кодову базу або знайти шаблони в ній, онтологія, описана в OWL, та пов'язані з нею можливості логічного виведення можуть бути надзвичайно цінними.

OWL є надійною перевіреною платформою для представлення знань, але вона має свій набір викликів. Розуміння того, коли та як використовувати можливості OWL, усвідомлюючи її обмеження, є ключовим для його ефективного використання.

На практиці метод таблиці для DL може бути набагато складнішим через оптимізації, роботу з номіналами, зворотними ролями, числовими обмеженнями тощо. На ефективність алгоритму може значно впливати порядок застосування правил, обробка блокування для запобігання нескінченним моделям та інші оптимізації.

Алгоритм гіпертаблиці – це приклад розвитку алгоритмічних підходів в основі систем логічного виводу для DL, який підкреслює важливість технік оптимізації для роботи з онтологіями реального світу, які часто є великими та складними.

Алгоритми наближення поліноміального часу є ключовим інструментом у комп'ютерних науках та дослідженнях операцій, знаходячи практичні рішення для проблем, які з обчислювальної точки зору є нерозв'язними за реалістичних часових рамок для більшості випадків.

Модульна декомпозиція – це метод, який використовується для розділення великих онтологій на менші, більш керовані модулі. Мета полягає в підвищенні обслуговуваності, зрозумілості та масштабованості онтологій. Працюючи з мовою онтологій Web (OWL), модульна декомпозиція стає особливо важливою, враховуючи великі та складні онтології, які можна побудувати в цьому фреймворку.

Використовуючи системи логічного виведення (reasoners), розробники можуть гарантувати, що онтології, які вони створюють, є логічно узгодженими та максимально інформативними, тим самим максимізуючи їх корисність у таких застосуваннях як відкриття знань, інтеграція даних та семантичний пошук.

Вибір системи логічного виведення часто залежить від конкретних вимог завдання, таких як використовуваний профіль OWL, складність та розмір онтології та питання продуктивності.

SWRL (Semantic Web Rule Language) – це розширення мови онтології Семантичного вебу, OWL (Ontology Web Language). Воно дозволяє користувачам виражати правила, додатково до визначень онтології. Правила SWRL складаються з антецедентів (умов) та консеквентів (висновків) і дозволяють отримувати нову інформацію на основі існуючих даних. За допомогою SWRL, логічне виведення над екземплярами онтології стає більш виразним, дозволяючи робити більш складні висновки в поєднанні з системами логічного виводу онтології, такими як Pellet та HermiT. Хоч SWRL і збільшує виразність OWL, воно дотримується обмежень щодо прийнятності, щоб забезпечити обчислення за скінченний час.

SQWRL (Semantic Query-Enhanced Web Rule Language) – це розширення мови правил Семантичного вебу, SWRL. SQWRL дозволяє формулювати запити до онтологій OWL, використовуючи синтаксис, схожий на правила. З його допомогою можна витягувати специфічну інформацію з онтологій, комбінуючи властивості SWRL для формулювання складних запитів.

З аналізу властивостей та можливостей OWL можемо зробити висновок, що виразність профілів OWL2 дозволяє представити початковий код із усіма його залежностями та складностями. Побудова моделі початкового коду на основі онтології про об’єктно-орієнтовану мову програмування із використанням додаткових засобів, таких як SWRL та SQWRL, дозволить ефективно застосовувати визначені правила рефакторингу, робити запити на вимогу розробників програмного забезпечення та проводити автоматизований аналіз початкового коду для подальшого його покращення. Математично вивірені та обґрунтовані алгоритми систем логічного виводу, що застосовуватимуться в пропонованій моделі, гарантуватимуть однозначність і консистентність дій рефакторингу, описаних із використанням різноманітних засобів.

РОЗДІЛ 3. ДОСЛІДЖЕННЯ МОЖЛИВОСТЕЙ ВБУДОВАНОГО РЕФАКТОРИНГУ У МОВІ SWIFT

3.1 Огляд можливостей рефакторингу у мові Swift

В утиліті sourcekit, що є частиною набору розробника мови програмування Swift існує два способи задання фрагменту коду для рефакторингу – відносно поточної позиції курсора в коді та в рамках певного вибраного діапазону коду. Локальний рефакторинг відбувається в межах одного файлу. Прикладами локального рефакторингу є винесення частини існуючого методу і оформлення його як нового методу (method extraction) та об’єднання повторюваних виразів (duplicates consolidation) (Alcocer, Antezana, Santos, & Bergel, 2020). Глобальні дії рефакторингу, які змінюють код у кількох файлах (наприклад, глобальне перейменування), наразі вимагають спеціальної координації інтегрованого середовища розробки Xcode (початковий код якого є пропрієтарним) і в даний час не можуть бути реалізовані покладаючись лише на власне початковий код мови Swift. Надалі розглядаються локальні рефакторинги.

Можливі дії рефакторингу залежать від позиції курсора в редакторі чи виділеної області (Kaur & Singh, 2017) (Saca, 2017). Відповідно до того, як вони ініціалізовані, дії рефакторингу класифікуються як курсорні або діапазонні. Ціллю рефакторингу по заданій позиції курсора у вихідному файлі Swift може бути, наприклад, перейменування поточної лексеми. Рефакторинг на основі діапазону потребує і початкової, і кінцевої позиції курсору, щоб вказати його ціль, наприклад, винесення частини існуючого методу і оформлення його як нового методу. Щоб полегшити реалізацію цих двох категорій, обробник sourcekit надає попередньо проаналізовані результати під назвою ResolvedCursorInfo та ResolvedRangeInfo, щоб відповісти на питання про позицію курсора або діапазон

у файлі з початковим кодом мовою Swift. (Tkachuk & Bulakh, Research of possibilities of default refactoring actions in Swift language, 2022)

Наприклад, ResolvedCursorInfo може надати інформацію про те, чи розташування курсору у вихідному файлі вказує на початок виразу і, якщо так, надати відповідний вузол абстрактного синтаксичного дерева для цього виразу. Крім того, якщо курсор вказує на лексему, що позначає ім'я сутності (змінної, класу, методу), ResolvedCursorInfo містить оголошення (декларацію), що відповідає цьому імені. Аналогічно ResolvedRangeInfo містить інформацію про заданий діапазон початкового коду, наприклад, чи має діапазон кілька точок входу або виходу. (Ткачук, 2020)

3.2 Реалізація нового сценарію рефакторингу для мови Swift

Щоб реалізувати новий рефакторинг для Swift, не потрібно починати з визначення позицій курсору або діапазону у початковому коді і здійснювати попередню його обробку; натомість можливо почати з ResolvedCursorInfo та ResolvedRangeInfo (які надаються sourcekit), на основі яких можна отримати детальну інформацію про початковий, що необхідна для рефакторингу.

Для написання функціоналу для здійснення рефакторингу, необхідно зважати на специфіку процесу компіляції в цілому. (Inoue & Roy, 2021)

Для виконання рефакторингу важливими є лише кілька перших основних етапів компіляції, які перетворюють початковий код у абстрактне синтаксичне дерево (AST).

Абстрактне синтаксичне дерево (AST) – це граф, основними елементами якого є оператори (тобто проміжні вершини графу) і операнди (тобто кінцеві вершини).

Усі вузли абстрактного синтаксичного дерева Swift можна поділити на три типи: Decl (оголошення), Expr (вирази) і Stmt (оператори).

Вони відповідають трьом сутностям, які використовуються в самій мові Swift. Імена функцій, структур, параметрів – це оголошення. Вирази – це

сутності, які повертають значення; наприклад, виклик функції. Оператори є частинами мови, які визначають потік управління і виконання коду, але не повертають значення (наприклад, if або do-catch). На Рисунок 5 зображено частину абстрактного синтаксичного дерева, яка описує оголошення членів класу. (Ткачук, 2020)

Вже після отримання AST можна здійснювати рефакторинг, закладений у компілятор (тобто застосовується код із компілятора, що описує можливі методи рефакторингу до початкового коду). Для рефакторингу коду, відповідно до потреб програміста, застосовується API, який дає змогу працювати із AST. Код, що описує рефакторинг, додається у спеціальний файл, що згодом компілюється і сам стає частиною засобів компіляції даної мови.

Для того, щоб створити інструмент рефакторингу, необхідно знання способів роботи з AST Swift. Важливою характеристикою AST Swift є те, що воно походить від лексем, а отже безпосередньо пов'язане з початковим кодом. Це означає, що можна отримати посилання на місце у файлі із початковим кодом, яке представляє конкретний вузол AST (Рисунок 5). Без цієї інформації неможливо було б перейменувати ідентифікатори, здійснювати переміщення, спрощувати виклик – загалом здійснювати рефакторинг. (Ткачук, 2020)

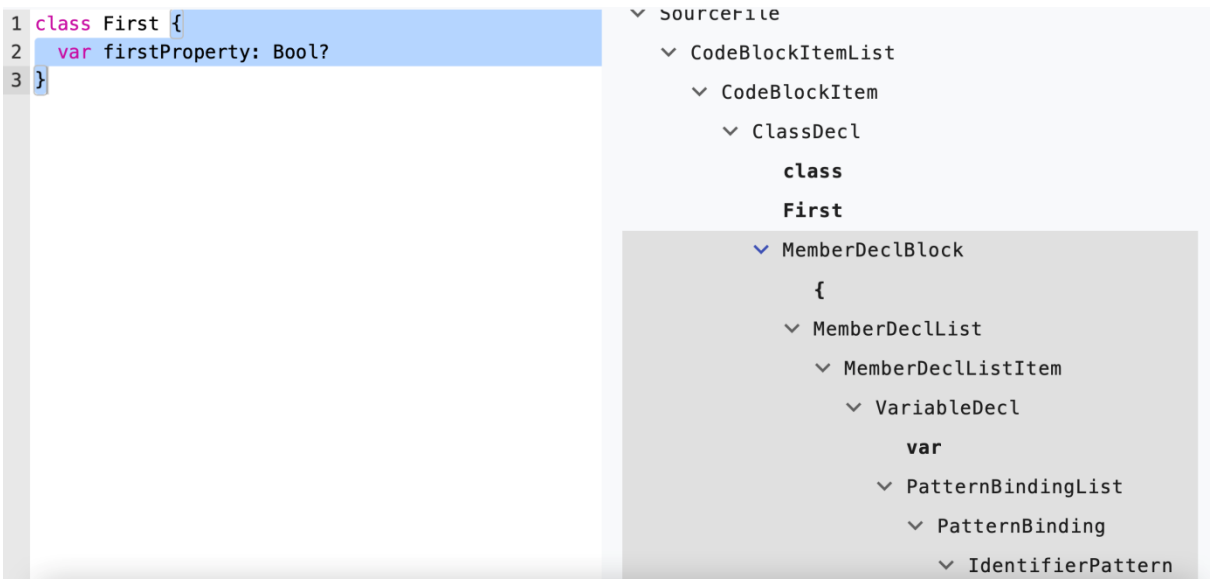

Рисунок 5. Наочне представлення абстрактного синтаксичного дерева

AST у фінальній формі має визначені типи та посилання на початковий код. Тільки для такого AST можливо здійснювати рефакторинг у плані опрацювання коду (перевірка виконання правил, наявності повторів, необхідних викликів тощо) та заміну початкового коду на необхідний.

Для пошуку можливостей покращення рефакторингу необхідно:

– дослідити принципи додавання нових дій рефакторингу в мові Swift;

– проаналізувати принципи збирання та побудови початкового коду Swift, їх специфічні особливості;

– оцінити складність розробки нових дій для рефакторингу та їх інтеграції із середовищем розробки.

Для виконання поставлених завдань доцільно звернутись до одного із відкритих завдань-пропозицій про додавання нової дії з рефакторингу в системі відстеження дефектів Swift (Swift issues, 2022) та спробувати здійснити його імплементацію. Обраним завданням було додати можливість автоматизованого доповнення коду методом, який вимагається протоколом Equatable.

Щоб реалізувати рефакторинг на основі позиції курсору (Add Equatable Conformance), потрібно спочатку оголосити цей рефакторинг у файлі RefactoringKinds.def із записом, зображеним на Рисунок 6. (Tkachuk & Bulakh, Research of possibilities of default refactoring actions in Swift language, 2022)

```
CURSOR_REFACTORING(AddEquatableConformance, "Add Equatable Conformance", add.equatable.conformance)
```

Рисунок 6. Оголошення рефакторингу

Це необхідно для того, щоб IDE при взаємодії із sourcekit мав змогу надати інформацію про доступний рефакторинг і показати його у відповідному контекстному меню. CURSOR_REFACTORING вказує, що цей рефакторинг ініціалізується у місці розташування курсору і, таким чином, використовуватиме ResolvedCursorInfo у реалізації (Ткачук, 2020). Реєстрація рефакторингу містить

у собі кілька додаткових речей, як-от внутрішній ідентифікатор рефакторингу для унікального адресування, рядкове представлення, яке буде відображатись користувачам під час відкривання контекстного меню, а також стабільний ключ. Оскільки пропоновані дії рефакторингу стануть частиною пакету розробки для мови Swift, внесена інформація має бути зрозумілою компілятору мови Swift (компілятор C++). Цей запис також дозволяє згенерувати «скелет» опису класу RefactoringActionAddEquatableConformance для рефакторингу та його викликів. Враховуючи це, необхідно відзначити, що одним із недоліків такої розробки є те, що розробник має фокусуватись не безпосередньо на реалізації необхідних функцій, а на написанні коректного коду в рамках доволі непростої інфраструктури класів для вбудованого рефакторингу. (Lacerda, Petrillo, Pimenta, & Guéhéneuc, 2020)

Після того, як здійснено оголошення нового рефакторингу, необхідно програмно реалізувати дві функції, які дадуть представлення про те:

1) коли необхідно показати дію рефакторингу;

2) яку зміну коду слід застосувати, коли користувач викличе цю дію рефакторингу.

Обидві функції генеруються автоматизовано як частина «скелету» класу після додавання запису про реєстрацію нової дії рефакторингу. Щоб інтегроване середовище розробки могло відобразити нову дію рефакторингу як доступну для застосування за правильних умов, необхідно реалізувати функцію isApplicable класу RefactoringActionAddEquatableConformance в Refactoring.cpp (як показано на Рисунок 7). ResolvedCursorInfo – це об’єкт, що містить в собі опис контексту, в якому було викликано дію рефакторингу (прив’язку до коду).

```
bool RefactoringActionAddEquatableConformance::
isApplicable(ResolvedCursorInfo Tok, DiagnosticEngine &Diag) {
  return AddEquatableContext::getDeclarationContextFromInfo(Tok).isValid();
}
```

Рисунок 7. Реалізація методу isApplicable

У методі isValid() (Рисунок 8) здійснюється перевірка, чи обраний контекст оголошення (клас, структура, перелічення) відповідає умовам здійснення рефакторингу, а саме: має збережені властивості (stored properties), не реалізує протокол Equatable та чи вимоги протоколу є правильними. Якщо метод повертає true, то такий тип рефакторингу буде доступний в інтегрованому середовищі розробки при розташуванні курсору у відповідному місці.

Далі необхідно реалізувати те, як слід змінити код під курсором, якщо застосовано дію рефакторингу. Для цього нам потрібно реалізувати метод performChange класу RefactoringActionAddEquatableConformance. Під час реалізації performChange можливо отримати доступ до того ж об'єкта ResolvedCursorInfo, який було отримано в isApplicable. (Ткачук, 2020)

```
bool isValid() {
  // FIXME: Allow to generate explicit == method for declarations which
  // already have compiler-generated == method
  return StartLoc.isValid() && ProtInsertStartLoc.isValid() &&
  !conformsToEquatableProtocol() && isPropertiesListValid() &&
  isRequirementValid();
}
```

Рисунок 8. Реалізація методу isValid()

На Рисунок 9 в методі performChange() здійснюється отримання контексту виклику, на основі якого відбувається пошук місця вставки для назви протоколу, що додається, а також для членів класу (функції), які будуть додані. У тілі функції можемо використовувати EditConsumer об'єкт для здійснення редагування тексту навколо виразу, вказаного курсором, з відповідними викликами API (insertAfter).

Усі дії по роботі із початковим кодом реалізовано в інших додаткових методах.

Розроблена дія рефакторингу (після її застосування до початкового коду) вставляє рядок «: Equatable» після лексеми обраного оголошення (declaration), що позначає реалізацію протоколу класом, структурою чи переліченням. Потім

здійснюється вставка в тіло оголошення (declaration) сутності реалізації методу, що вимагається протоколом Equatable.

```
bool RefactoringActionAddEquatableConformance::
performChange() {
  auto Context = AddEquatableContext::getDeclarationContextFromInfo(CursorInfo);
  EditConsumer.insertAfter(SM, Context.getStartLocForProtocolDecl(),
                           Context.getInsertionTextForProtocol());
  EditConsumer.insertAfter(SM, Context.getInsertStartLoc(),
                           Context.getInsertionTextForFunction(SM));
  return false;
}
```

Рисунок 9. Реалізація методу performChange()

На Рисунок 10 показано код, що здійснює отримання назви протоколу і її формування буферу тексту, який необхідно вставити у початковий код.

На Рисунок 11 описано код, що здійснює отримання тексту методу, що необхідно вставити. Спочатку здійснюється пошук вимог протоколу (назви методу і параметрів). Далі здійснюється пошук необхідного відступу із врахуванням місця вставки тексту методу. Після цього конфігурується клас, що здійснює формальний друк методу у початковому коді із врахуванням всіх параметрів.

```
std::string AddEquatableContext::
getInsertionTextForProtocol() {
  StringRef ProtocolName = getProtocolName(KnownProtocolKind::Equatable);
  std::string Buffer;
  llvm::raw_string_ostream OS(Buffer);
  if (ProtocolsLocations.empty()) {
    OS << ": " << ProtocolName;
    return Buffer;
  }
  OS << ", " << ProtocolName;
  return Buffer;
}
```

Рисунок 10. Отримання тексту для протоколу

Окрім описаних вище методів необхідно було здійснити розробку відносно великої кількості допоміжних функцій для організації коректної роботи

основних функцій. (Tkachuk & Bulakh, Research of possibilities of default refactoring actions in Swift language, 2022)

Для написання тестів використовується спеціальний підхід. Необхідно задекларувати наявність певного роду тестів для дій рефакторингу, створити файли із вхідними та вихідними даними, застосувати функції і тоді фреймворк тестування здійснюватиме порівняння отриманого результату із бажаним.

```
std::string AddEquatableContext::
getInsertionTextForFunction(SourceManager &SM) {
  auto Reqs = getProtocolRequirements();
  auto Req = dyn_cast<FuncDecl>(Reqs[0]);
  auto Params = Req->getParameters();
  StringRef ExtraIndent;
  StringRef CurrentIndent =
    Lexer::getIndentationForLine(SM, getInsertStartLoc(), &ExtraIndent);
  std::string Indent;
  if (isMembersRangeEmpty()) {
    Indent = (CurrentIndent + ExtraIndent).str();
  } else {
    Indent = CurrentIndent.str();
  }
  PrintOptions Options = PrintOptions::printVerbose();
  Options.PrintDocumentationComments = false;
  Options.setBaseType(Adopter);
  Options.FunctionBody = [&](const ValueDecl *VD, ASTPrinter &Printer) {
    Printer << " {";
    Printer.printNewline();
    printFunctionBody(Printer, ExtraIndent, Params);
    Printer.printNewline();
    Printer << "}";
  };
  std::string Buffer;
  llvm::raw_string_ostream OS(Buffer);
  ExtraIndentStreamPrinter Printer(OS, Indent);
  Printer.printNewline();
  if (!isMembersRangeEmpty()) {
    Printer.printNewline();
  }
  Reqs[0]->print(Printer, Options);
  return Buffer;
}
```

Рисунок 11. Отримання тексту для методу протоколу

На Рисунок 12 наведено приклад одного з таких тестів.

```
extension TestAddEquatable {
    func test() -> Bool {
        return true
    }
}

extension TestAddEquatable {
}

// RUN: rm -rf %t.result && mkdir -p %t.result

// RUN: %refactor -add-equatable-conformance -source-filename %s -pos=1:16 > %t.result/first.swift
// RUN: diff -u %S/Outputs/basic/first.swift.expected %t.result/first.swift
```

Рисунок 12. Написання тесту

Подібним чином відбувається і розробка дій рефакторингу, які базуються на виділеному діапазоні коду:

– оголошення дії у файлі RefactoringKinds.def;

– реалізація методу isApplicable(), що показує, коли можна застосовувати дію рефакторингу (для опису контексту виклику використовується об’єкт ResolvedCursorInfo);

– реалізація методу performChange() для застосування змін;

– тестування.

Важливо, що в тілі функції performChange можливо отримати доступ не тільки до оригінального ResolvedCursorInfo чи ResolvedRangeInfo для вибраного користувачем місця чи діапазону, але й до інших важливих утиліт, таких як EditConsumer та SourceManager, що робить реалізацію більш зручною.

3.3 Результати проведеного дослідження

Розглянемо результат роботи розробленого рефакторингу. Початковий файл для проведення рефакторингу зображено на Рисунок 13.

Здійснимо застосування рефакторингу для оголошення класу TestAddEquatable. Результат роботи утиліти зображено на Рисунок 14 – додано назву протоколу після назви класу, а також додано реалізацію методу в тілі класу із використанням всіх властивостей класу.

```
class TestAddEquatable {
    var property = "test"
    private var prop = "test2"
    let pr = "test3"
}

extension TestAddEquatable {
    func test() -> Bool {
        return true
    }
}

extension TestAddEquatable {
}
```

Рисунок 13. Початковий файл

Далі застосуємо рефакторинг до розширення класу TestAddEquatable, що містить в своєму тілі якусь інформацію.

```
class TestAddEquatable: Equatable {
    var property = "test"
    private var prop = "test2"
    let pr = "test3"

    static func == (lhs: TestAddEquatable, rhs: TestAddEquatable) -> Bool {
        return lhs.property == rhs.property &&
               lhs.prop == rhs.prop &&
               lhs.pr == rhs.pr
    }
}

extension TestAddEquatable {
    func test() -> Bool {
        return true
    }
}

extension TestAddEquatable {
}
```

Рисунок 14. Результат застосування утиліти для класу

Результат роботи виглядає так, як показано на Рисунок 15.

```
class TestAddEquatable {
    var property = "test"
    private var prop = "test2"
    let pr = "test3"
}

extension TestAddEquatable: Equatable {
    func test() -> Bool {
        return true
    }

    static func == (lhs: TestAddEquatable, rhs: TestAddEquatable) -> Bool {
        return lhs.property == rhs.property &&
               lhs.prop == rhs.prop &&
               lhs.pr == rhs.pr
    }
}

extension TestAddEquatable {
}
```

Рисунок 15. Результат застосування утиліти для розширення класу, що містить додатковий код

Таким чином, розширення класу містить декларацію реалізації вказаного протоколу.

ВИСНОВКИ ДО РОЗДІЛУ 3

Для дослідження можливостей розширення вбудованого рефакторингу було досліджено програмну реалізацію компонента sourcekit мови програмування Swift, що відповідає за роботу із початковим кодом як звичайним текстом і його попередню обробку, а також реалізовано додавання нової дії з рефакторингу з його використанням. Для виконання плану дослідження було обрано одну дію рефакторингу, що не була присутня в утилітах рефакторингу, а саме – додавання реалізації протоколу Equatable. Було розроблено її програмну імплементацію за допомогою компонентів і ресурсів, що надаються в межах компоненту sourcekit. Для перевірки правильності та відповідності умовам розробки було створено та проведено ряд випробувань.

Встановлено, що обидва механізми рефакторингу, які підтримуються мовою програмування Swift, мають обмежений контекст і обмежену зону дії та застосування. Саме тому можливість розширення функціоналу має базуватись не на локальному рівні опрацювання коду, а на верхньому рівні, де можливо поєднати кілька вихідних файлів, що часто відбувається у реальних проєктах. Робота була направлена на розробку власної дії рефакторингу для аналізу та отримання досконалого представлення про переваги та недоліки існуючого компоненту. Як результат, було запропоновано новий підхід до здійснення рефакторингу, що дозволить вирішити описані вище проблеми.

У результаті експериментів з існуючим API утиліти sourcekit, що відповідає за рефакторинг у мові Swift, було встановлено, що для додавання лише одної відносно простої дії рефакторингу необхідно було написати близько 300 рядків коду. І немає причин вважати, що в інших випадках потрібно буде написати суттєво меншу кількість рядків коду. Таким чином, існує актуальна потреба в удосконаленні та автоматизації процесу розширеного рефакторингу, що дозволила б програмісту уникнути написання чималої кількості «інфраструктурного» коду, більше зосередившись на кінцевій меті рефакторингу. Написаний код (реалізація методів isApplicable(), performChange()) є частиною

мови Swift і не може параметризуватись при виконанні (усі нові функції рефакторингу, які додаються, строго формалізовані і реалізуються лише відповідно до наданого фреймворку).

Хоч проведене дослідження і мало обмеження через направленість на один конкретний тип рефакторингу, що додавався, воно дало змогу оцінити перспективні напрямки подальших досліджень, серед яких можна виділити розробку методів і способів рефакторингу, що не залежать повністю від початкового коду мови програмування. Це пов’язано з тим, що новий підхід не буде мати обмежень до рефакторингів, які є в існуючому (як от доступ лише до контексту викликаної дії). (Tkachuk & Bulakh, Research of possibilities of default refactoring actions in Swift language, 2022)

РОЗДІЛ 4. ЗАСТОСУВАННЯ ФОРМАЛІЗОВАНИХ ЗНАНЬ ПРО ПОЧАТКОВИЙ КОД ДЛЯ ЗДІЙСНЕННЯ РЕФАКТОРИНГУ У МОВІ SWIFT

4.1 Огляд підходів до удосконалення процедури рефакторингу

Використання стандартного підходу (того, що пропонується як невід'ємний компонент мови) для рефакторингу у мові Swift є обмеженим. (Tkachuk & Bulakh, Research of possibilities of default refactoring actions in Swift language, 2022) Додавання нових дій рефакторингу здійснюється суто за одним принципом, що унеможливлює додавання обробки нестандартних ситуацій.

Для написання дії рефакторингу обов'язково мати компілятор, оскільки рефакторинг працює не із «сирим» кодом, а із його представленням – абстрактним синтаксичним деревом. Використання доданих дій можливе лише за їх «основним» призначенням без можливості параметризації та зміни властивостей під час роботи із утилітою. Та й написання нових дій є складним, що уповільнює процес появи нових можливостей рефакторингу (Lacerda, Petrillo, Pimenta, & Guéhéneuc, 2020). Саме ці перелічені обмеження, що накладаються на процес розробки нових дій рефакторингу, зумовлюють відсутність різноманіття дій, що можуть виконуватись, вузьку направленість тих дій, що вже існують (застосування лише до певної частини коду із однією метою). Для того, аби мати можливість здійснювати повноцінний рефакторинг із застосуванням методик, що пропонуються в каталозі антипатернів коду (Catalog of Refactoring, 2023), необхідно мати великий набір інструментів, що неможливо через складність їх розробки (Almogahed, Omar, & Zakaria, 2022). Розробники, що здійснюють рефакторинг, змушені в такому разі багато роботи під час рефакторингу виконувати вручну, що зводить нанівець завдання автоматизованого рефакторингу і збільшує кількість помилок у коді. (Kaur & Singh, 2019)

Для того, аби уникнути усіх описаних вище обмежень, пропонується використання сутностей і понять зі складу повноцінної бази знань при здійсненні рефакторингу. Тоді користувач повинен описово сформулювати завдання на рефакторинг. (Morales, Soh, Khomh, Antoniol, & Chicano, 2017)

Отже, такий метод здійснення рефакторингу, що і пропонується в даній дисертації, повинен здійснювати рефакторинг на основі вхідного запиту, який складений з понять бази знань, які описують деталі початкового коду. Інноваційність такого підходу в тому, що на даний момент на ринку не існує програмних продуктів для рефакторингу, в самій основі яких було б закладено орієнтацію на використання бази знань про початковий код, як це пропонується в даній дисертації.

4.2 Дослідження застосування формалізованих знань у процесі рефакторингу

Під час виконання дослідження було здійснено пошукову роботу та виявлено проблеми, які не можуть бути вирішені засобами стандартного рефакторингу. (De Nicola, Di Stefano, Inverso, & Uwimbabazi, 2022)

Для прикладу і порівняння було обрано процедуру пошуку всіх методів, які мають більше, ніж 3 параметри. Було перевірено вбудований засіб рефакторингу для отримання результату для порівняння (завдання виявилось непосильним). Такий же тест було проведено для додаткових утиліт рефакторингу, які працюють на регулярних виразах. Було отримано результат, що вони не здатні вирішити таке завдання. (Hammad, Babur, Basit, & van den Brand, 2022)

Для потреб даного дослідження «з чистого аркуша» було розроблено архітектуру програмної бази знань, архітектуру програмного двигуна і їх реалізацію.

Розроблена програмна утиліта приймає на вхід рядок, який подібний до наступного: *Test.swift find func paramsCount greaterThanOrEquals 3*.

Для того, аби всі запити вирішувались успішно, необхідно здійснити опис сутностей та понять і правильно запрограмувати їх обробку (Al Dallal, 2012). Саме для цього і слугує база знань. У даному проекті база знань – це формалізований опис початкового коду, написаного мовою Swift. Вона містить його властивості, відповідності між «сирим» кодом, сутностями і поняттями коду (класи, структури, рядкові літерали), їхніми властивостями (назва, ідентифікатор, кількість параметрів) та діями, які можуть виконуватися (пошук, перейменування). Такий опис може бути формалізований, наприклад, за допомогою описових логік (та відповідних мов на кшталт OWL-DL), однак у даному дослідженні, для спрощення, створюється мета-опис за тією ж мовою програмування Swift. (Tkachuk & Bulakh, Usage of formalized knowledge about source code for refactoring actions in Swift, 2022)

На Рисунок 16 подано діаграму класів. Вона описує частину бази знань, яка містить у собі власне знання про тип сутності String та які властивості вона має.

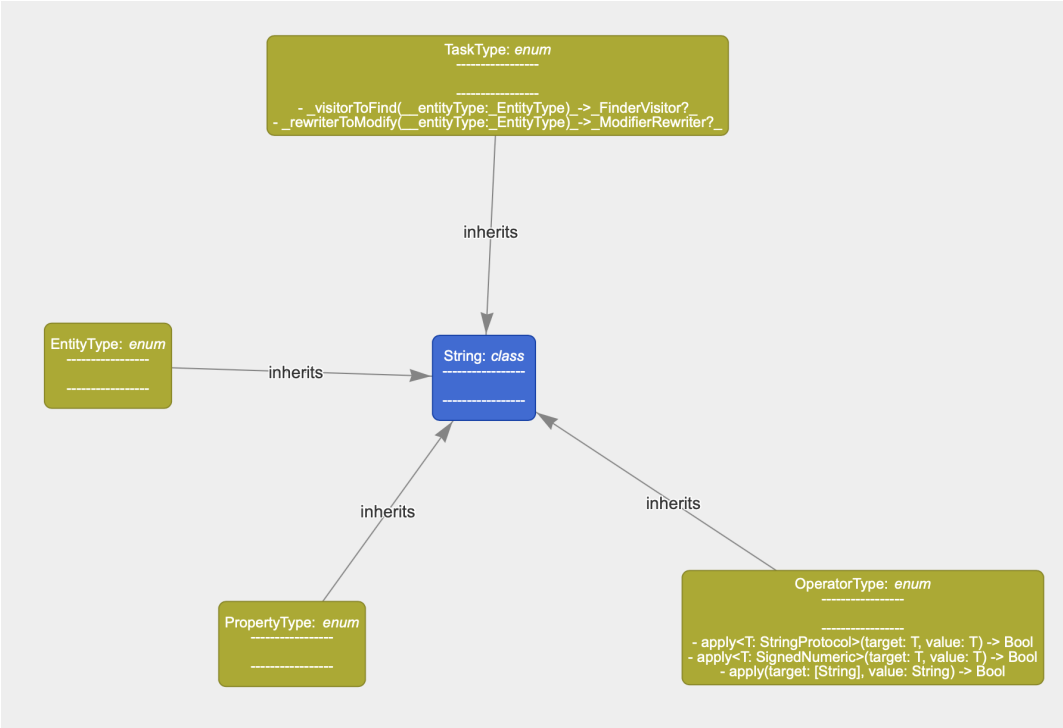

Рисунок 16. Діаграма класів для сутності String

Завдяки протокольній організації коду, будь-яку сутність можна описати за допомогою формалізованих методів, що дозволить ефективно розширювати функціонал програмного продукту з рефакторингу на основі даного підходу.

На Рисунок 17 зображено діаграму класів для програмного двигуна. Він застосовується у разі надходження запиту на аналіз, як-от «пошук», чи зміну (власне, застосування рефакторингу).

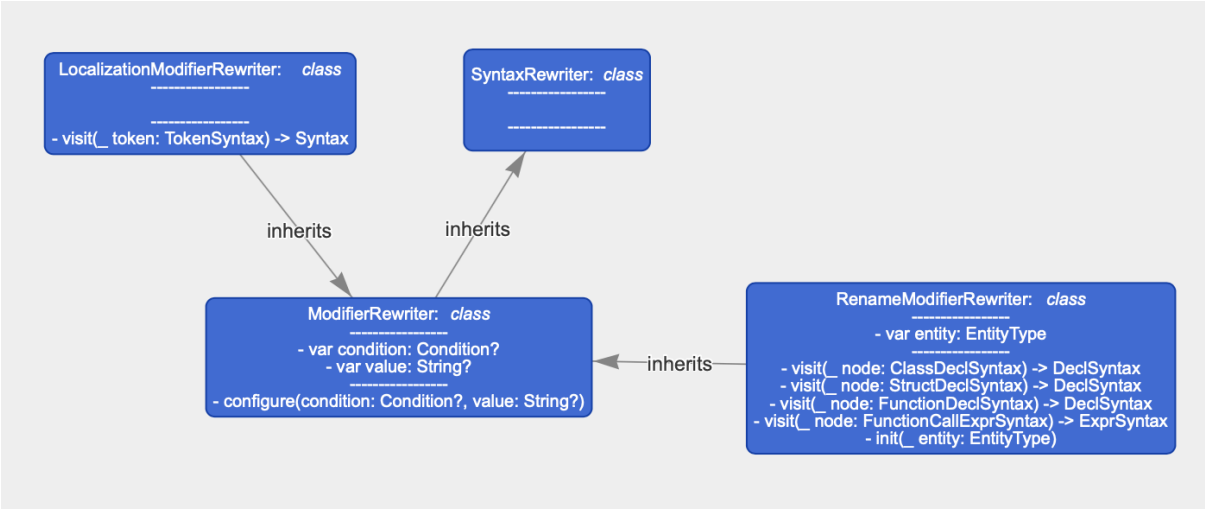

Рисунок 17. Діаграма класів для програмного двигуна

Двигун здійснює прочитання початкового коду, надсилає його на оцінку в модуль бази знань і повертає результат користувачеві у вигляді текстової відповіді чи зміненого початкового коду.

На Рисунок 18 зображено діаграму класів, що описує взаємодію класів двигуна із класами, що представляють базу знань.

Розроблений прототип є утилітою командного рядка. Створення графічного інтерфейсу користувача не є доцільним, оскільки утиліта може виступати суб-процесом для інтегрованого середовища розробки чи спеціального текстового редактора. (Tkachuk & Bulakh, Usage of formalized knowledge about source code for refactoring actions in Swift, 2022)

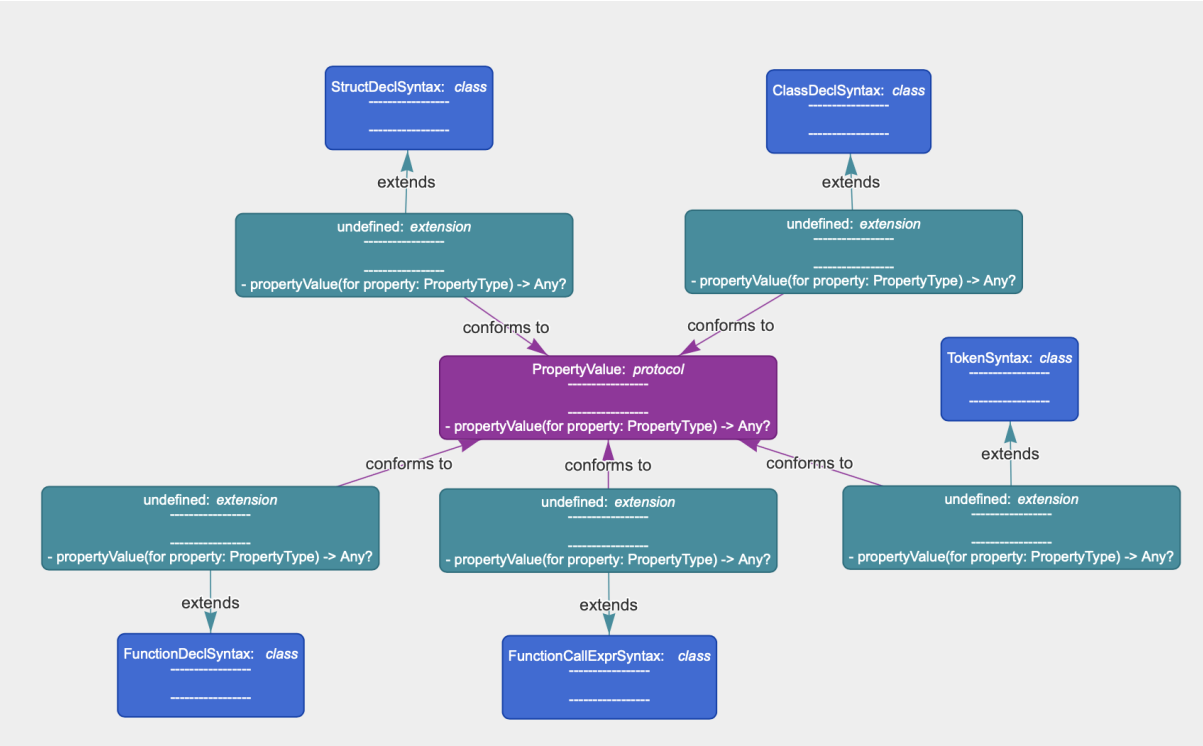

Рисунок 18. Діаграма класів у частині взаємодії двигуна із базою даних

Для перевірки усіх поставлених до продукту вимог було здійснено ряд тестових запусків і перевірка крайових умов виконання.

4.3 Результати проведеного дослідження

Важливою властивістю розробленого прототипу програмного продукту є те, що команди, які подаються, не є «жорсткими», тобто не мають бути сформовані лише у певному фіксованому вигляді. Наприклад, команда *rename class name equals Test Renamed* не повинна бути запрограмованою саме в такому вигляді. Вона повинна успішно відпрацювати, навіть якщо змінити тип сутності на структуру: *rename struct name equals Test Renamed.* Тобто, програма підтримує логічне виведення із комбінації доступних (реалізованих) сутностей, понять, дій і правильно виконує завдання. Саме завдяки такій властивості досягається високий показник масштабування програмного продукту, оскільки на початковій стадії розробляється двигун, що опрацьовує команду і виконує її, а потім доповнюється лише база знань описом сутностей, їх властивостей, дій, що можуть бути виконані.

Основною перепоною при розробці програмного прототипу було отримання можливості працювати із абстрактним синтаксичним деревом. Обробляти «сирий» початковий код і формалізувати його власноруч – неефективне завдання, оскільки саме таке завдання вирішується початковими етапами компілятора мови Swift. Тобто у випадку власної реалізації функції обробки, це було б написанням компілятора «з нуля». Доцільність виконання такої роботи залишається під великим сумнівом. (Tkachuk & Bulakh, Usage of formalized knowledge about source code for refactoring actions in Swift, 2022)

Для того, аби отримати доступ до абстрактного синтаксичного дерева, що генерується компілятором, необхідно мати доступ до лексичного аналізатора та токенізатора, що є складовими компілятора із бібліотеки libSyntax, написаної на C++. Для розробки прототипу було використано бібліотеку SwiftSyntax, яка є високорівневою обгорткою для бібліотеки libSyntax.

Блок-схема алгоритму роботи програми зображена на Рисунок 19.

Робота із програмою здійснюється через термінал. Для початку роботи необхідно ввести команду в термінал. Після цього програма працює самостійно і не потребує введення додаткових даних від користувача.

Після отримання команди починається її перевірка на коректність.

У випадку, якщо команда виявилася некоректною, програма завершує виконання і виводить відповідну помилку. Якщо ж команда є коректною, то програма продовжує роботу.

Після перевірки команди здійснюється ініціалізація класу, за допомогою якого здійснюється проходження по файлу, що містить початковий код. Клас вибирається та налаштовується відповідно до написаної в команді задачі.

Для кожного вузла програми, що підпадає під умови, зазначені в команді, виконується оцінка умови і застосування завдання, що включає виведення результату роботи у тій чи іншій формі.

Після обробки всіх вузлів програма завершує виконання.

Для реалізації запропонованого методу в програмному прототипі, для початку можна рекомендувати підтримку сутностей, властивостей та задач, які описані в Таблиця 2.

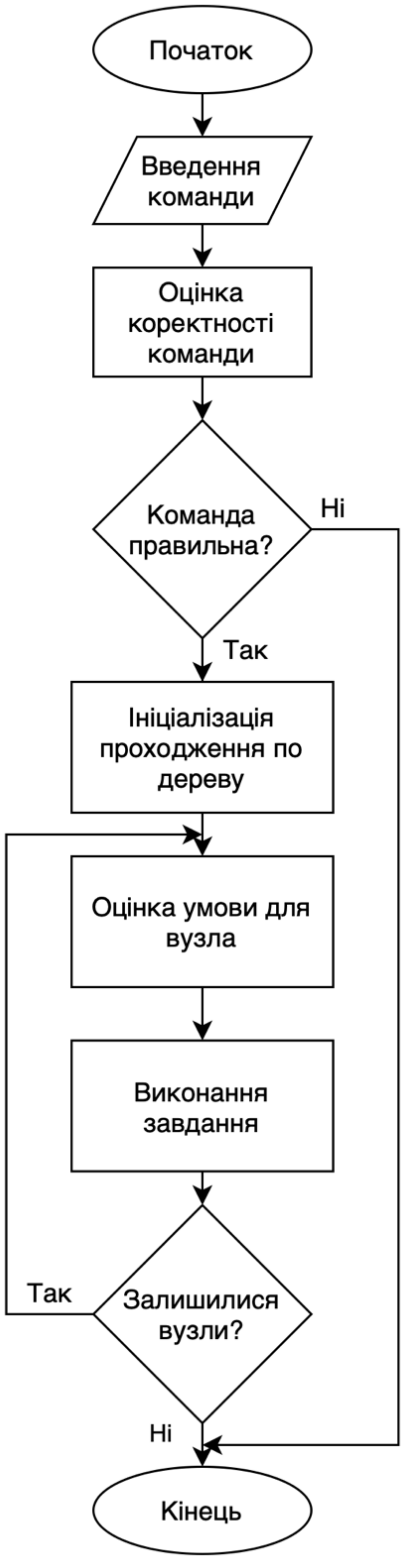

Рисунок 19. Блок-схема алгоритму програми

Під час оцінки результатів було проведено ряд тестів. Найбільш показові тести подано нижче.

На Рисунок 20 зображено початковий код, написаний мовою Swift. Для перевірки можливості створеного програмного прототипу знаходити в коді об’єкти, які неможливо знайти із використанням стандартних засобів рефакторингу, виконаємо наступну команду: refactor Test.swift find class conforms to FirstProtocol. Результатом роботи програми є діапазон у тексті початкового коду (рядок початку і символ та рядок закінчення і символ).

Таблиця 2. Пропонований функціонал

Тип параметра	Доступні варіанти
Задачі	find (пошук), rename (перейменування), localize (локалізація)
Сутності	class (клас), struct (структура), string (рядковий літерал), func (функція/метод)
Властивості сутностей	name (назва), paramsCount (кількість параметрів), conforms (реалізує), inherits (успадковує)
Оператори	equals (дорівнює), contains (містить), from (від), to (до), greaterThanOrEquals (більше чи дорівнює)

На Рисунок 21 виділено діапазон, який програма знайшла в результаті отриманого запиту. Очікуваний результат – розташування оголошення класу SecondTestClass.

```
class FirstTestClass {
    class SecondTestClass: FirstProtocol {
        var name = "SecondTestClass"

        func shouldReturnTrue(for item: String, secondItem: String) {
            return true
        }
    }

    var name = "FirstTestClass"

    func returnTrue() -> Bool {
        return true
    }
}

protocol FirstProtocol {
    func shouldReturnTrue(for item: String, secondItem: String)
}
```

Рисунок 20. Початковий код до змін

Як можна бачити, програма успішно впоралася із виконанням завдання, незважаючи навіть на те, що шуканий клас задекларований як внутрішній для іншого класу.

```
class FirstTestClass {
    class SecondTestClass: FirstProtocol {
        var name = "SecondTestClass"

        func shouldReturnTrue(for item: String, secondItem: String) {
            return true
        }
    }

    var name = "FirstTestClass"

    func returnTrue() -> Bool {
        return true
    }
}

protocol FirstProtocol {
    func shouldReturnTrue(for item: String, secondItem: String)
}
```

Рисунок 21. Результат для першого тесту

У другому тесті використаємо той же ж початковий код, що й в першому (Рисунок 20). Але здійснимо перевірку пошуку об’єктів на основі їх характеристик. Для цього запустимо на виконання команду: refactor Test.swift find func paramsCount equals 2. Очікуваний результат – програма знайде метод

shouldReturnTrue. (Tkachuk & Bulakh, Usage of formalized knowledge about source code for refactoring actions in Swift, 2022)

У цьому випадку програма видала два результати. Перший (Рисунок 22) – оголошення методу, який має два параметри, в протоколі.

```
17  protocol FirstProtocol {
18      func shouldReturnTrue(for item: String, secondItem: String)
19  }
```

Рисунок 22. Результат 1 для другого тесту

Другий (Рисунок 23) – реалізація методу в класі, який реалізує вищезгаданий протокол.

```
1  class FirstTestClass {
2      class SecondTestClass: FirstProtocol {
3          var name = "SecondTestClass"
4
5          func shouldReturnTrue(for item: String, secondItem: String) {
6              return true
7          }
8      }
9
10     var name = "FirstTestClass"
11
12     func returnTrue() -> Bool {
13         return true
14     }
15 }
```

Рисунок 23. Результат 2 для другого тесту

Отже, очікуваний результат відповідає дійсності – було правильно знайдено всі згадки методів у початковому коді, які відповідають заданим умовам.

ВИСНОВКИ ДО РОЗДІЛУ 4

За результатами експериментальних досліджень було отримано підтвердження того, що системи автоматизованого аналізу й рефакторингу, що вже існують, не повністю задовольняють потреби кінцевого користувача і не здатні виконувати відносно нетривіальні завдання рефакторингу.

Для зменшення кількості помилок, внесених під час виконання рефакторингу, спрощення самого процесу виконання рутинних дій пропонується використовувати новий спосіб для рефакторингу, що полягає в роботі із високорівневими командами користувача на основі формалізованого опису початкового коду разом із базою знань, що містить опис сутностей коду та їх властивостей (які конкретні дії можна виконати із ними). У дослідженні рефакторинг початкового коду здійснено на прикладі мови програмування Swift. Рекомендований підхід компонентної архітектури (база знань, програмний двигун) в подальшому дозволяє розширити функціонал програмного продукту на інші мови програмування.

Для перевірки запропонованого підходу було виконано роботу з розробки прототипу програмного продукту із використанням запропонованого підходу для перевірки та порівняння результатів з іншими засобами рефакторингу. Розроблено утиліту командного рядка, яка приймає на вхід вербальну команду та здійснює вивід результатів обробки та аналізу початкового коду (пошук складних конструкцій у коді) або застосовує запропоновану зміну. У результаті проведеного тестування встановлено, що використання запропонованого підходу дозволяє виконувати складні завдання рефакторингу за допомогою простої вербальної формалізованої команди. Виконання такого ж завдання із використанням тільки вбудованих засобів рефакторингу вимагає значно більше часу та зусиль або ж взагалі неможливе.

Перевагою пропонованого програмного прототипу є те, що він може виконувати завдання рефакторингу нестандартного типу, адаптуватися під потребу користувача. Проте головною перевагою є швидкість і гнучкість

додавання нового функціоналу. Для цього варто лише описати у базі знань (у даному прототипу – це окремі файли в програмі), як саме пов’язати код із властивостями для певної сутності і додати ключові слова в обробник команд.

Обмеженням даного дослідження є те, що не було розглянуто можливість отримання абстрактного синтаксичного дерева для інших мов програмування та можливість його інтеграції із розробленою схемою бази знань.

Під час аналізу результатів було виявлено, що досліджений підхід та прототип на його основі має перспективи для розвитку і подальшої комерціалізації. Такими напрямками слід вважати підтримку кількох файлів, об’єднання кількох умов в одній команді та підтримку інших мов програмування. Також у якості розвитку ідеї доцільно розглянути опис знань про код не у формі ієрархії класів мови програмування, а у формі ієрархії понять та сутностей. Вони можуть бути описані мовами типу OWL із використанням відповідних засобів логічного виведення.

Для експериментального підтвердження даної ідеї було проведено ряд нетривіальних операцій, недоступних для стандартних засобів рефакторингу. Однак сформульовані твердження справедливі і для обробки коду, написаного іншими сучасними високорівневими мовами програмування.

РОЗДІЛ 5. ОПИС ЗНАНЬ ПРО ПОЧАТКОВИЙ КОД З ВИКОРИСТАННЯМ ОНТОЛОГІЇ

5.1 Опис методу здійснення автоматизованого рефакторингу

Створивши онтологію, яка представляє поняття, відносини та властивості елементів початкового коду, таких як класи, методи, змінні та їх взаємодія, стає можливим охоплювати та організовувати знання про кодову базу. Це зробить можливою реалізацію різних корисних функцій системи рефакторингу, таких як автоматизований аналіз коду, генерація документації коду, семантичний пошук та міркування про структуру та поведінку коду. Онтології можуть допомогти розробникам та інженерам програмного забезпечення краще розуміти та маніпулювати початковим кодом, надаючи структуроване представлення його семантики. (Happel & Seedorf, 2006)

Розширені системи автоматизованого рефакторингу та аналізу початкового коду можуть отримати переваги від властивостей бази знань, які підтримуються онтологією. Вони можуть зберігати посилання між сутностями коду, строго визначати їх типи, надсилати запити до бази знань та виконувати будь-які види логічного виведення щодо даних. Вони можуть допомогти додавати або переписувати початковий код з гарантією, що він буде компілюватися (на відміну від систем ШІ з неспецифікованими або нечіткими продукційними правилами).

Використання онтологій для зберігання знань про початковий код може бути складним у створенні та обслуговуванні, вимагаючи експертизи в предметній області та постійних зусиль, направлених на підтримку актуальності таких онтологій. Крім того, відтворення повного обсягу кодової бази та управління масштабованістю може бути викликом, що робить підходи на основі онтологій менш зручними для великих та різноманітних кодових баз.

Дослідимо можливості використання онтології як формальної бази знань про початковий код у автоматизованих системах.

OWL вважається кращим для представлення онтологій завдяки своїй вищій виразності, відповідності стандартам семантичного вебу, можливостям логічного виведення та добре розробленому екосистемі інструментів, що дозволяє точне моделювання та взаємодію. Однак вибір мови представлення онтології в кінцевому підсумку залежить від конкретних вимог та складності домену, який моделюється. (Tkachuk & Bulakh, Describing the knowledge about the source code using an ontology, 2023)

Враховуючи всі вищезазначені факти, OWL було обрано основним інструментом для створення бази знань про початковий код.

Формалізований процес рефакторингу з використанням запропонованого методу зображено на Рисунок 24.

Визначення методу аналізу та модифікації коду з використанням бази знань:

Нехай X – множина тверджень початкового коду (оголошення, присвоєння, т. д.), X' – множина тверджень початкового коду після опрацювання, Y – множина елементів БЗ, Y' – множина елементів БЗ після опрацювання. Необхідно розробити алгоритм $a: X \to Y$ здатний створити сутність y в повноцінній онтології для $\forall x \in X$, а також алгоритм b: $Y' \to X'$.

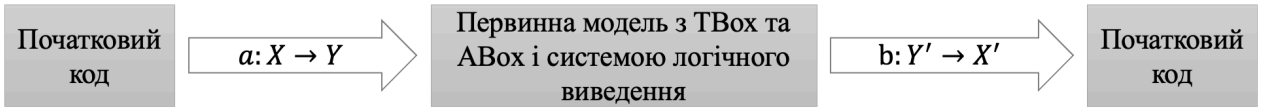

Рисунок 24. Формальне представлення методу

База знань у цьому методі – модель початкового коду, яка містить набір TBox аксіом (класи, атрибути і відношення про синтаксис мови програмування, основні властивості синтаксичних конструкцій), а також базовий набір аксіом для можливостей класифікації і виведення нових знань про екземпляри. Такий набір завантажується в базу знань із онтології про об’єктно-орієнтовану мову

програмування. Друга частина бази знань – набір ABox аксіом, які додаються на основі синтаксичного аналізу конкретного початкового коду. У кінцевому результаті – база знань містить вичерпний перелік класів, властивостей, відношень і екземплярів, які описують конкретний початковий код. Під час роботи системи логічного виведення усі невказані явно факти про код буде додано в базу знань, а усі суперечності виявлено і повідомлено про них. Це відповідатиме вимогам автоматизованого рефакторингу, де код буде змінено відповідно до знань про такий рефакторинг у використаній онтології.

На блок-схемі (Рисунок 25) зображено кроки запропонованого методу:

- користувач вводить необхідну команду для виконання рефакторингу (командою може виступати запит на отримання інформації про початковий код чи здійснення зміни початкового коду або ж ініціалізація повністю автоматизованого рефакторингу на основі попередньо внесених знань про дії рефакторингу – залежно від потреб та реалізації системи);
- якщо раніше прочитаний початковий код змінився або не був прочитаний, здійснюється його прочитання і попередня обробка засобами конкретної мови програмування, якою написаний код. Якщо код не змінився – виконання одразу переходить до процесу логічного виведення в базі знань;
- із прочитаного коду будується AST, а на наступному кроці для кожного вузла створюється відповідний «об’єкт для передачі даних» (англ. DTO – див. далі) до бази знань;
- далі здійснюється додавання отриманих об’єктів в базу знань;
- у той же ж час в базу знань завантажуються онтології (якщо вони не були завантажені раніше або були змінені з моменту попереднього завантаження), які будуть описувати і код, і предметну область;
- коли усі операції із внесення даних в базу знань завершено, запускається процес логічного виведення на основі всіх аксіом і перевірка на

узгодженість. Результатом роботи методу на даному кроці є набір нових аксіом;

- на наступному кроці отримані аксіоми додаються в базу знань, таким чином створюючи повне цілісне представлення початкового коду із здійсненими можливими модифікаціями, які представлені аксіомами, виведеними системою логічного виведення на попередньому кроці;
- далі здійснюється обернене перетворення елементів бази знань в допоміжні об’єкти для передачі даних з БЗ до AST. Варто зауважити, що такі об’єкти запитуються з бази знань на вимогу, а вимога і спосіб запиту залежать від конкретної реалізації системи перетворення, що застосовується;
- відповідно до отриманих об’єктів, відбувається заміна вузлів AST.

Блоки на сірому фоні відображають кроки, які здійснюються вперше.

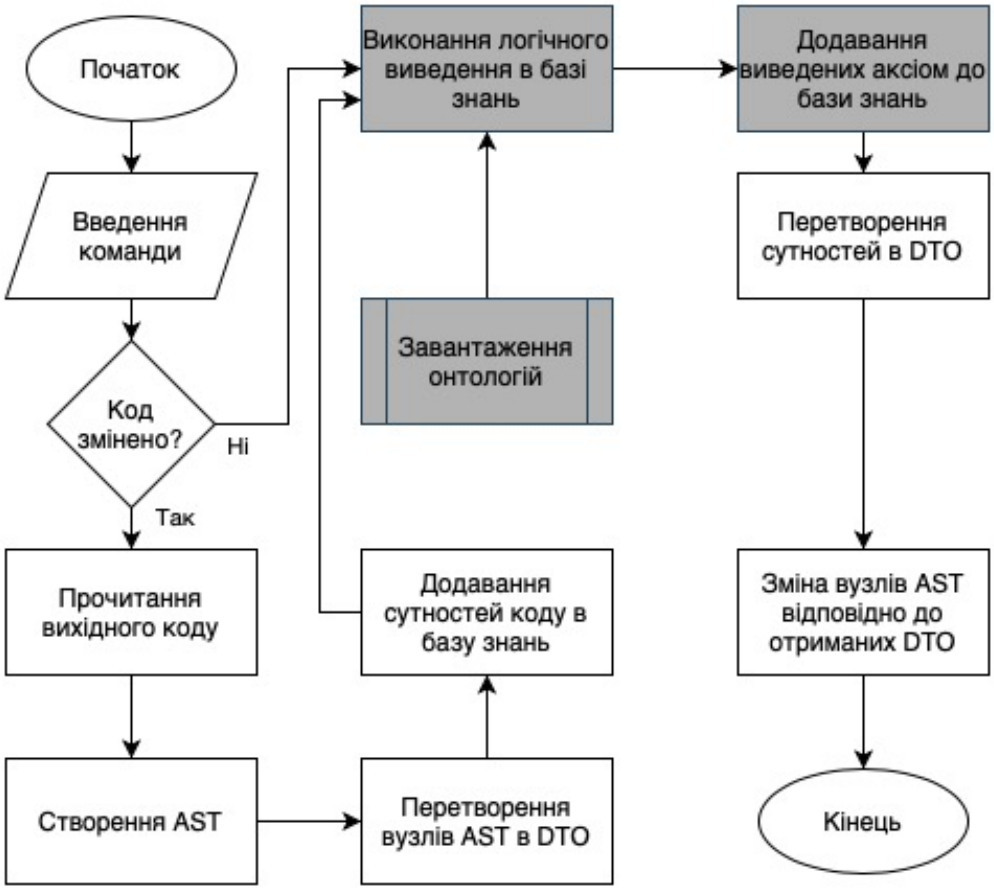

Рисунок 25. Блок-схема запропонованого методу

Для перевірки концепції та доведення, що база знань, заснована на онтології, дасть очікуваний результат, можуть бути використані чи створені наступні представлення: великі типи груп (такі як класи, структури, переліки, функції тощо) можуть бути представлені класами OWL, кожна сутність у початковому коді може бути представлена як екземпляр, що належить до відповідного класу, властивості екземплярів можуть бути виражені властивостями даних, а відносини між екземплярами можуть бути виражені об'єктними властивостями. Також, щоб заповнити всі прогалини, побудувати справжнє семантичне представлення кодової бази та забезпечити виведення нових знань, може бути використано великий набір аксіом (включаючи, але не обмежуючись, підкласом, еквівалентним класом, роздільним класом, властивостями даних, об'єктними властивостями аксіоми).

Щоб мати змогу отримати відсутні аксіоми в онтології, перевірити узгодженість з проблемами коду (якщо такі визначені в онтології) та відповідно вносити зміни в кодову базу, слід використовувати систему логічного виведення. Вона використовує визначені аксіоми, правила та логічні конструкції в онтології для отримання нових знань, перевірки узгодженості та відповіді на запити. Для потреб цього дослідження було обрано систему логічного виведення HermiT. (Glimm, Horrocks, Motik, Stoilos, & Wang, 2014)

Попередні результати дослідження показують, що існують рішення, які мають на меті використання онтологій для представлення знань про початковий код. Ці рішення зазвичай застосовують техніки з аналізу програмного забезпечення, обробки природних мов та інженерії онтологій. (Tkachuk & Bulakh, Usage of formalized knowledge about source code for refactoring actions in Swift, 2022)

Важливо зазначити, що, хоча ці рішення існують, процес додавання знань про код в базу знань, що засновується на онтології, може бути складним, і отримана БЗ може вимагати ручного вдосконалення та обслуговування. Складність і нюанси коду часто ускладнюють точне фіксування всієї необхідної

інформації. Крім того, вибір конкретних інструментів або методів може відрізнятися залежно від мови програмування, розміру кодової бази та бажаного представлення бази знань.

Подальші дослідження можливих застосувань цих інструментів (JOIE, SemTK, SourcererCC) виявили, що навіть якщо вони надають деякі можливості для легкого керування триплетами RDF або онтологіями OWL, вони не надають API чи будь-якої іншої можливості транслювати початковий код в базу знань. Такий висновок свідчить про те, що мета поточного дослідження має певну новизну.

Процес отримання знань із початкового коду та представлення його у вигляді бази знань передбачає аналіз кодової бази для визначення різних елементів коду, таких як класи, методи, змінні та їхні зв'язки. Мета полягає в тому, щоб охопити структуру, семантику та взаємозалежності в кодовій базі та перетворити їх у структуроване представлення, яке відповідає концепціям онтології, властивостям і зв'язкам.

Такий процес зазвичай включає методи аналізу програмного забезпечення, розбір (parsing) та семантичного розуміння. Ці методи можуть включати статичний аналіз, обхід абстрактного синтаксичного дерева (AST), визначення типів та розпізнавання шаблонів. Аналізуючи кодову базу, витягаючи релевантну інформацію та відображаючи її в конструкціях онтології, процес має на меті забезпечити формальне представлення структури та семантики коду. (Tkachuk & Bulakh, Usage of formalized knowledge about source code for refactoring actions in Swift, 2022)

База знань, отримана з коду, може застосовуватись у різних цілях, таких як пошук коду, повторне використання коду, створення документації та автоматизоване обґрунтування поведінки коду. Це покращує розуміння та використання кодової бази, надаючи структуроване та семантичне представлення її елементів і зв'язків.

У дослідженні ми зосереджуємось на процесі створення бази знань, що базується на онтології, з коду та прагнемо реалізувати модель, яка може використовуватися автоматизованими інструментами для рефакторингу та аналізу. Спочатку ідея полягала в тому, щоб використовувати ту саму мову програмування (яка використовується для написання початкового коду) для представлення бази знань (Tkachuk & Bulakh, Describing the knowledge about the source code using an ontology, 2023) (Tkachuk & Bulakh, Research of possibilities of default refactoring actions in Swift language, 2022). Це дало можливість швидко інтерпретувати початковий код і перетворювати його на сутності та аксіоми OWL.

З цієї причини було розглянуто декілька фреймворків та інструментів для роботи із формалізованими знаннями. Оскільки основною мовою дослідження є Swift, до уваги бралися лише бібліотеки з підтримкою такої мови. Одним з них є OWL-API для iOS (Ruta, Scioscia, Di Sciascio, & Bilenchi, 2016) з відповідним обчислювальником – Mini-ME Swift (Ruta, Gramegna, Bilenchi, & Di Sciascio, 2019). Їхні автори стверджують, що для мови Swift бракує семантичних систем логічного виводу і бібліотек маніпулювання OWL, що робить доцільним їх розробити. Перенесення існуючих рішень або використання багатоплатформних бібліотек може бути складним через архітектурні відмінності.

На жаль, розглянуті фреймворки та бібліотеки знаходяться на початковій стадії розробки. Вони не підтримують активну роботу з онтологією (наявний тільки синтаксичний аналіз імпортованих даних), а окремі сутності OWL (такі як властивості даних) все ще недоступні.

Крім того, використання однієї мови для клієнта (parsing client) та бази знань робить ці два екземпляри тісно пов’язаними, що може призвести до непотрібної переробки, коли один із них змінюється (Рисунок 26).

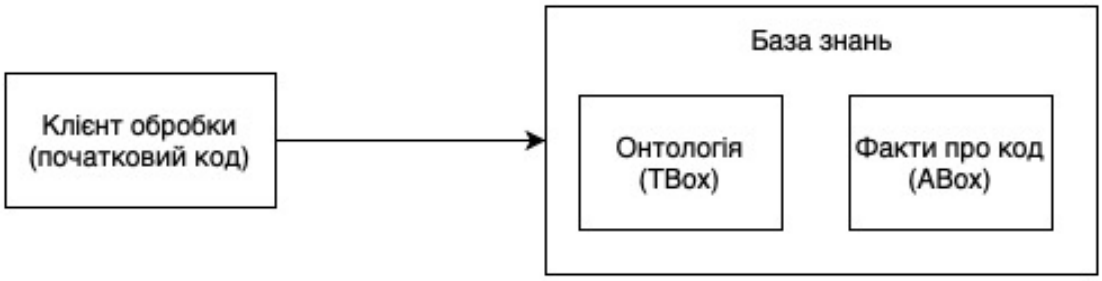

Рисунок 26. Залежність між клієнтом і онтологією

Щоб розірвати зв'язок, слід використовувати абстракцію (відповідно до принципу інверсії залежностей – дає змогу створювати гнучкий і підтримуваний код шляхом інвертування напрямку залежностей, де модулі вищого рівня залежать від абстракцій, а не від конкретних реалізацій, що дозволяє легше замінювати та розширювати компоненти) і клієнт-серверну архітектуру. Об’єкт передачі даних (DTO) може бути використаний як посередник у цьому випадку. Угода DTO може бути встановлена, а моделі DTO будуть створені з обох сторін – клієнта та обробника онтології (Рисунок 27). Це допоможе роз’єднати екземпляри та надасть можливість інтегрувати абстрактну онтологію з різними клієнтами (кілька мов програмування) або системами керування онтологіями.

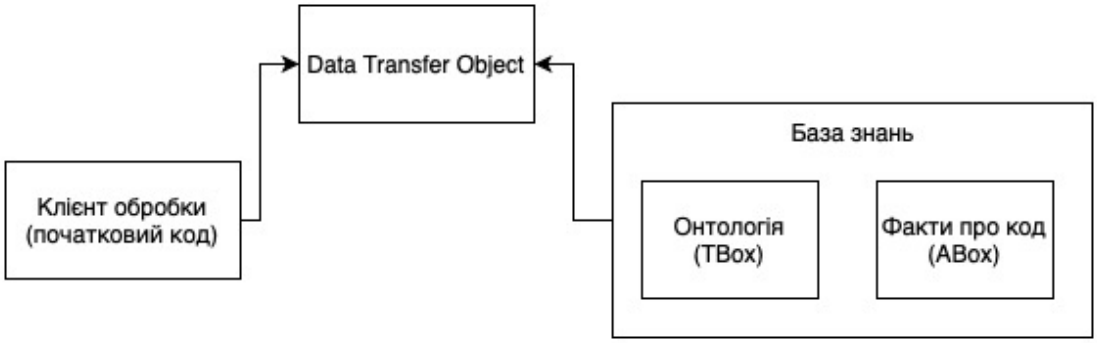

Рисунок 27. Інверсія залежності за допомогою DTO

Схематичне зображення розподілу відповідальностей між клієнтом і сервером відносно процесу рефакторингу зображено на Рисунок 28.

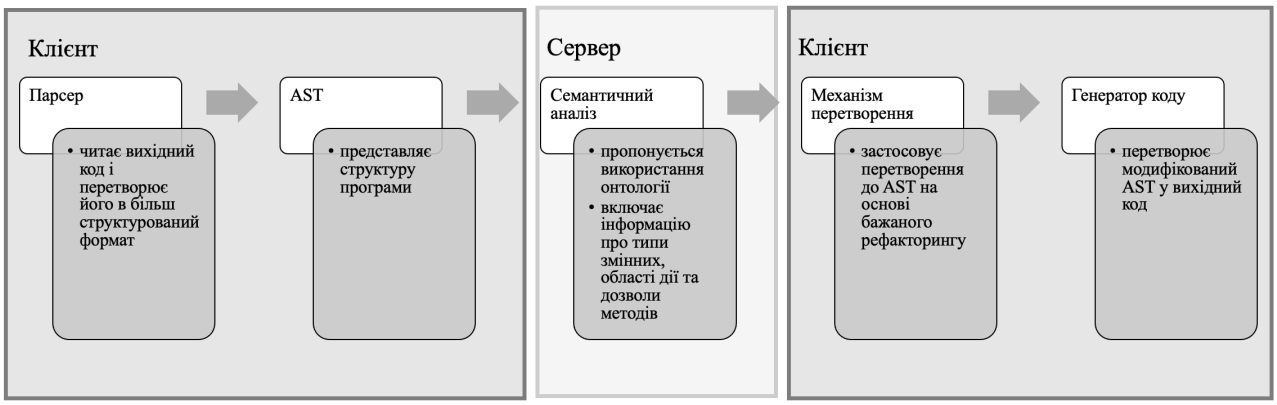

Рисунок 28. Розподіл відповідальності між клієнтом та сервером

Варто сказати, що впровадження власної бібліотеки для маніпулювання онтологіями OWL є недоцільним. По-перше, це вимагатиме значних зусиль і ресурсів для створення та підтримки, хоча створення такої бібліотеки не є основною метою дослідження. По-друге, розробники та дизайнери повинні володіти відмінними знаннями про ці технології. Останнім часом існуючі бібліотеки та фреймворки OWL пропонують добре перевірені та оптимізовані рішення з широкою підтримкою інструментів, що може заощадити зусилля на розробку та забезпечити сумісність із встановленими стандартами та практиками.

Щоб підтримувати всі необхідні риси бази знань, згенерованої системою створення онтології з коду (і побудувати таку систему), слід брати до уваги спеціальні бібліотеки. Варто згадати OWL-API (Matentzoglu & Palmisano, 2016) і ONT-API (ONT-API: an RDF-centric Java library to work with OWL2, 2023). Це бібліотеки Java для роботи з онтологіями. OWL-API забезпечує підтримку онтологій OWL2 (можливість їх створення, оновлення, серіалізації та десеріалізації). Навпаки, ONT-API (який створений на основі OWL-API) забезпечує підтримку RDF і побудований на основі Apache Jena. Ці бібліотеки мають усі необхідні властивості онтології, щоб їх можна було використовувати в системі, що досліджується.

5.2 Дослідження реалізації процесу рефакторингу з використанням БЗ

Беручи до уваги всі висновки останніх досліджень і статей, було прийнято рішення побудувати прототип системи перетворення початкового коду в базу знань, яка є незалежною від мови, використовує абстракції, бібліотеки маніпулювання OWL з достатньою функціональністю, легко розширюється та здатна інтегруватися з іншими системами чи засобами (такими як онтологія або маніпулювання кодовою базою).

Щоб побудувати доказ концепції та показати, що запропонований метод є цілком і повністю працездатним, вирішено створити:

- клієнт мовою Swift, який аналізуватиме кодову базу Swift;
- RESTful API, який прийматиме JSON DTO сутностей коду (Dirsumilli & Mossakowski, 2016);
- менеджер бази знань по роботі з онтологією OWL, який оброблятиме всі взаємодії з нею.

Діаграма головних компонентів зображена на Рисунок 29. Завдяки архітектурі, може існувати кілька клієнтів, які підключаються до сервісу Ontology, а сам сервіс Ontology може працювати з різними менеджерами онтології або інструментами для логічного виведення або візуалізації онтології.

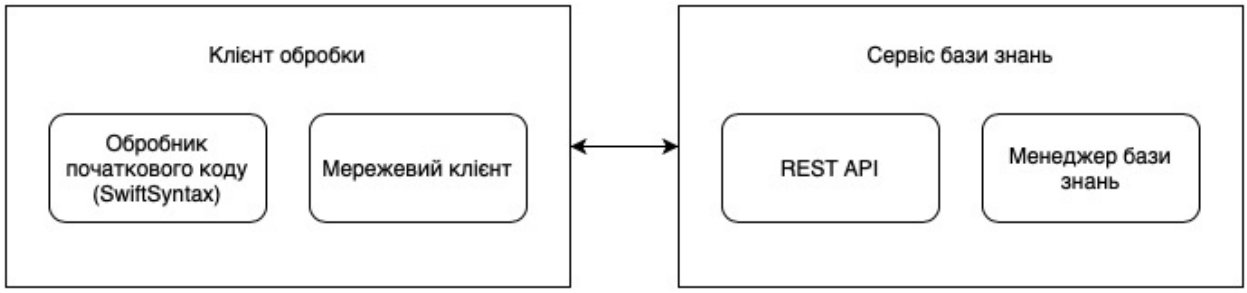

Рисунок 29. Принципова схема прототипу системи перетворення початкового коду в базу знань

Перш ніж почати роботу з онтологією, варто спроектувати її структуру. Для перевірки теоретичних засад пропонованого методу на практиці

скористаємось «спрощеною» версією онтології про об’єктно-орієнтовану мову програмування (пропоновану модель бази знань, що ґрунтується на онтології, буде розглянуто в наступному розділі) За основу взято онтологію з відкритим доступом (Atzeni & Atzori, 2017), але ця онтологія не містить властивостей та відношень, які є необхідними для здійснення рефакторингу. Тому в «спрощеній» онтології в цьому розділі будемо використовувати створені власноруч сутності (класи, властивості), які в подальшому будуть перенесені в основну модель.

На Рисунок 30 зображено структуру «спрощеної» онтології.

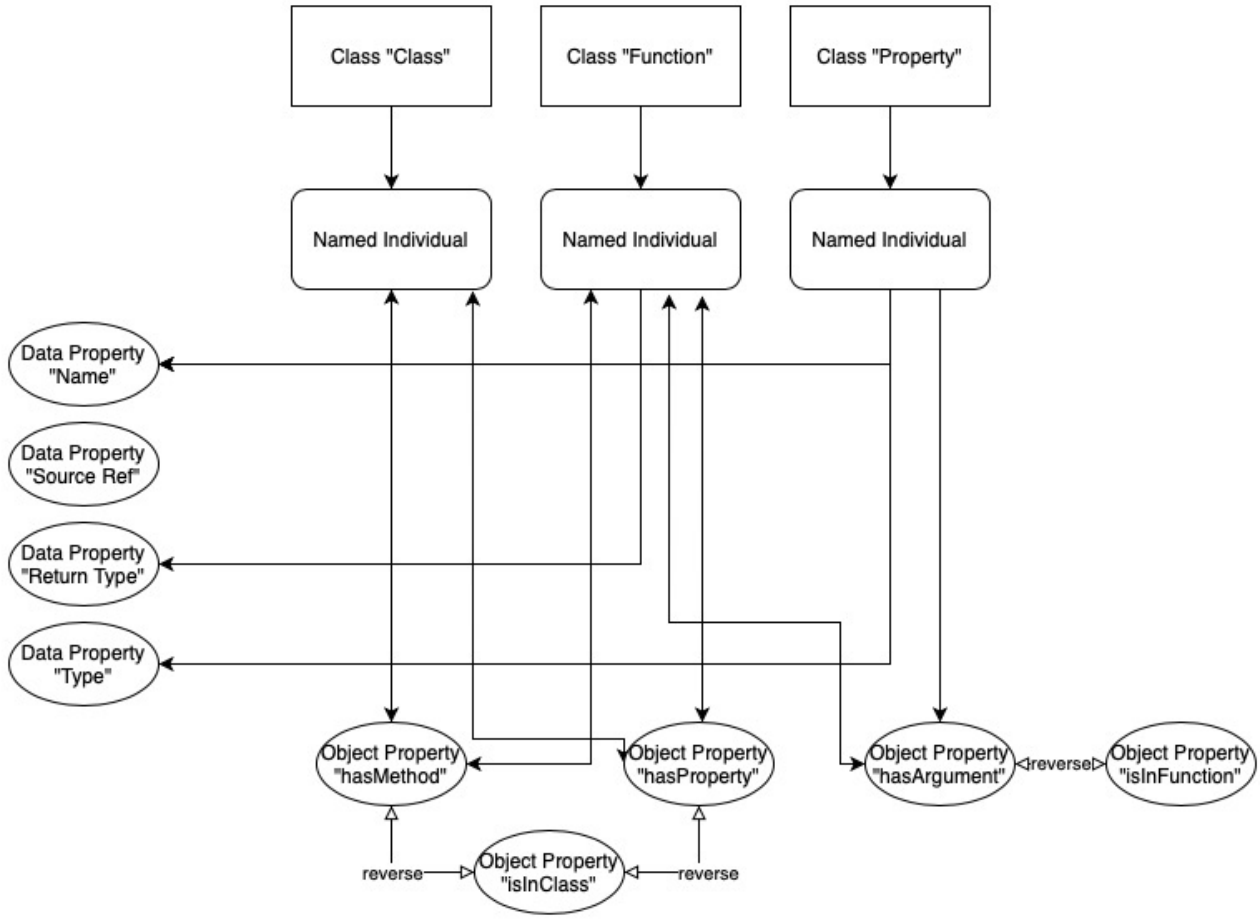

Рисунок 30. Пропонована залежність понять в онтології

Було створено три класи: *Class, Function, Property* . Усі сутності з кодової бази, які відповідають типу класу, будуть перетворені в екземпляри, які належать до відповідного класу онтології. Ці екземпляри будуть пов’язані один з одним такими властивостями: «*hasMethod*», «*hasProperty*», «*hasArgument*». Для того,

щоб підтримувати двонаправлені посилання, слід створити дві інверсні властивості об’єкта: «*isInClass*», «*isInFunction*».

Щоб мати можливість описувати екземпляри, створюється набір властивостей даних. Це «*name*», «*type*» (тип даних, наприклад *integer*), «*returnType* » (тип даних повернення). Крім того, існує властивість «*sourceRef*» (що означає «*SourceCodeReference*»), яка вказує, де в кодовій базі знаходиться конкретна сутність. Ця властивість є корисною, оскільки вона дозволить унікально ідентифікувати сутність з початкового коду та забезпечить зворотну сумісність – щоб мати можливість генерувати або змінювати код на основі сутностей з БЗ. Крім того, об’єктні властивості «*isInClass*» та «*isInFunction*» визначаються як взаємовиключні, що означає, що екземпляр, який теоретично може мати або ту, або іншу властивість (у нашому випадку це екземпляр класу Property), не може мати ці два зв’язки одночасно (іншими словами – властивість у програмному коді не може одночасно бути членом класу і аргументом функції).

На стороні клієнта бібліотека SwiftSyntax (Tkachuk & Bulakh, Research of possibilities of default refactoring actions in Swift language, 2022) використовується для аналізу початкового коду, написаного мовою Swift. Потім розібрані вузли AST перетворюються на визначені DTO сутності, які відомі сервісу. Наступний фрагмент зображує код, який прочитує вузол VariableDeclSyntax (представляє властивості класу, структури, перелічення або звичайні змінні) і перетворює його на об’єкт Property:

```
override func visit(_ node: VariableDeclSyntax) -> SyntaxVisitorContinueKind {
    let name = node.bindings.first?.pattern.description ?? ""
    let type = node.bindings.first?.typeAnnotation?.type.description ?? ""
    let reference = node.sourceRange(converter: converter).debugDescription
    let property = Property(name: name,
                            type: type,
                            sourceCodeReference: reference)
    properties.append(property)
    return .skipChildren
}
```

На останньому етапі бібліотека Alamofire використовується для підключення до сервісу онтології через HTTP та передачі всіх відвіданих об’єктів у наданому початковому коді. На стороні сервісу простий HTTP-сервер із пакету Sun забезпечує REST API для зв’язку. Ось фрагмент коду Java, який обробляє вхідні дані та створює нову іменовану особу, яка належить до класу «Клас»:

```
@Override
public String handleInput(String input) {
    Class classEntity = converter.fromJson(input, classOfT:Class.class);
    return InternalOntology.sharedInstance().addClass(classEntity);
}
```

Дані, які передаються між клієнтом та сервісом онтології, мають формат JSON. Наступний приклад коду створює новий екземпляр:

```
private OWLNamedIndividual createFunction(Function func) {
    IRI identifier = IRI.create(iri, UUID.randomUUID().toString());
    OWLNamedIndividual funcDecl = dataFactory.getOWLNamedIndividual(identifier);
    ontology.addAxiom(dataFactory.getOWLDeclarationAxiom(funcDecl));
    ontology.addAxiom(dataFactory.getOWLClassAssertionAxiom(
        functionEntity, funcDecl));
    ontology.addAxiom(dataFactory.getOWLDataPropertyAssertionAxiom(
        nameProperty, funcDecl, func.name));
    ontology.addAxiom(dataFactory.getOWLDataPropertyAssertionAxiom(
        returnTypeProperty, funcDecl, func.returnType));
    ontology.addAxiom(dataFactory.getOWLDataPropertyAssertionAxiom(
        sourceCodeReferenceProperty, funcDecl, func.sourceCodeReference));

    for (Property prop: func.arguments) {
        OWLNamedIndividual propDelc = createProperty(prop);
        ontology.addAxiom(dataFactory.getOWLObjectPropertyAssertionAxiom(
            hasArgumentProperty, funcDecl, propDecl));
    }
    return funcDecl;
}
```

По-перше, для нового екземпляра створюється унікальний IRI . По-друге, створюється новий іменований індивідуальний об'єкт і додаються всі можливі

аксіоми. Наостанок, усі залежні сутності (у цьому випадку екземпляри класу Property) створюються таким же ж способом, який описано в перших двох кроках.

Щоб переконатися, що очікувана онтологія містить усі необхідні аксіоми, слід провести тест. Для цілей тестування було створено простий фрагмент коду. Припустимо, що цей фрагмент є частиною якогось більшого файлу з початковим кодом (зауважимо, що було додано номери рядків для кращого розуміння того, як використовується *посилання на початковий код).* Цей демонстраційний фрагмент коду має дуже просту ієрархію класів, але вона демонструє успадкування та перевизначення методів:

```
 7 ...
 8 protocol Say {
 9     func say() -> String
10 }
11
12 class Animal: Say {
13     let age: Int
14
15     init() {
16         age = 0
17     }
18
19     func say() -> String {
20         return ""
21     }
22 }
23
24 final class Cat: Animal {
25     private var lives: Int
26
27     override init() {
28         lives = 9
29         super.init()
30     }
31
32     override func say() -> String {
33         return "meow"
34     }
35 }
36
37 final class Dog: Animal {
38     private var senses: Int
39
40     override init() {
41         senses = 6
42         super.init()
43     }
44
45     override func say() -> String {
46         return "woof"
47     }
48 }
49 ...
```

Після обробки початкового коду було створено базу знань. Потім було запущено систему логічного виводу HermiT, щоб добудувати всі можливі аксіоми. У результаті були додані інверсії та базова ієрархія класів онтології. Оновлену онтологію зображено на Рисунок 31, де жирним шрифтом виділено результати висновків, додані до початкової онтології системою HermiT.

На жаль, не всі очікувані знання про код з'явилися в онтології . Як видно на фрагменті коду, існують ієрархічні зв’язки (успадкування та відповідність

протоколу), але ні початкова онтологія, ні оновлена – після міркування – не містять цих зв’язків.

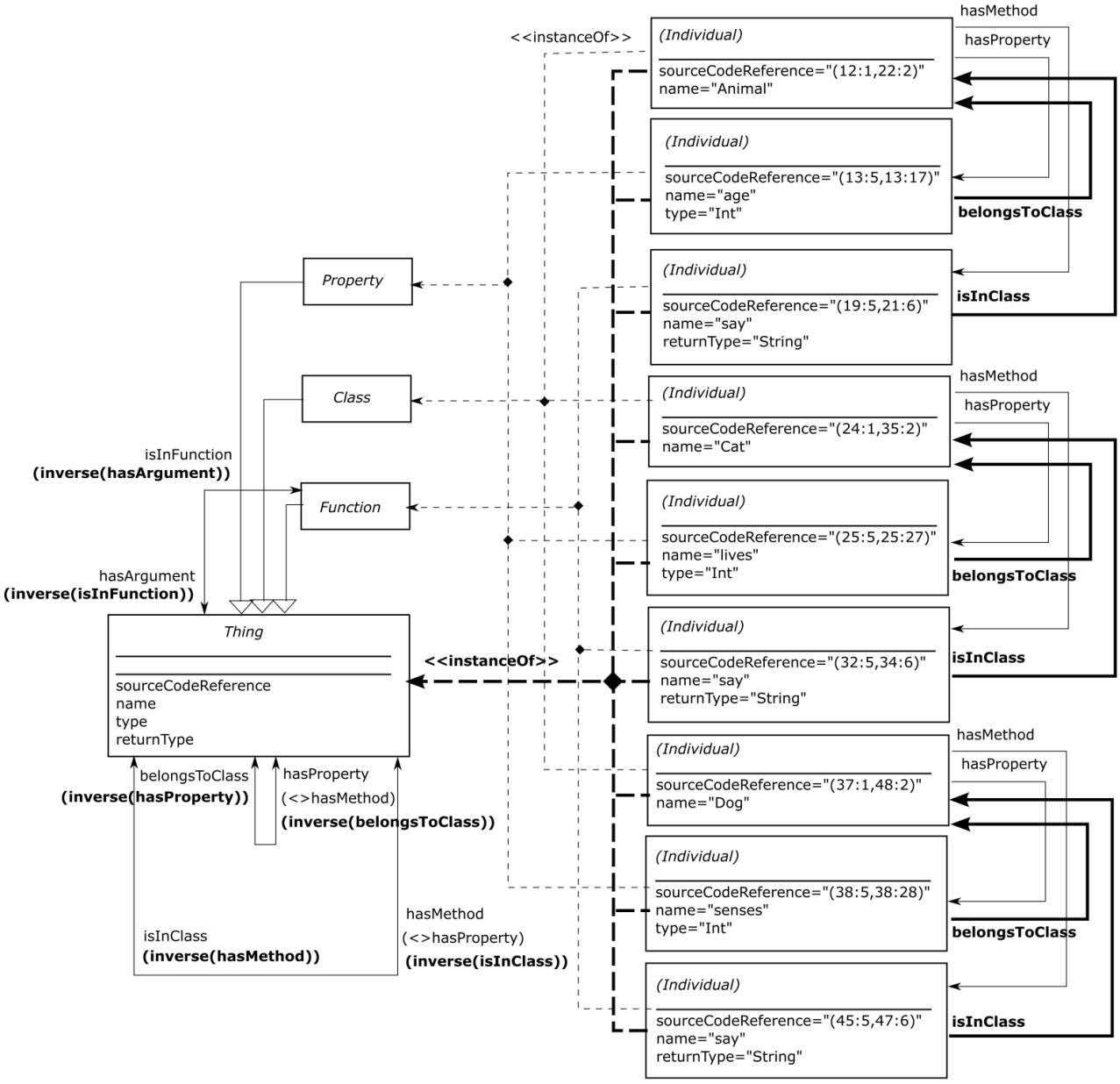

Рисунок 31. Онтологія після опрацювання (додані аксіоми виділені жирним шрифтом, ідентифікатори особи пропущені для стислості)

Це означає, що слід провести подальше дослідження та проектування, щоб знайти та описати оптимальний набір властивостей та сутностей, які повинні підтримуватися системою створення бази знань з коду, щоб мати можливість отримати більше знань про початковий код. (Tkachuk & Bulakh, Usage of formalized knowledge about source code for refactoring actions in Swift, 2022)

ВИСНОВКИ ДО РОЗДІЛУ 5

Використання бази знань, що містить факти, які відповідають певній онтології, є, вочевидь, ефективним способом зберігання знань, в тому числі – і про початковий код.

Щоб мати можливість ефективно представляти кодову базу, необхідно визначити широкий діапазон «будівельних блоків» онтології початкового коду. Вони включають, але не обмежуються класами, екземплярами, властивостями даних, властивостями об’єктів і широким спектром аксіом.

Коли всі аксіоми в базі знань (побудованій поверх онтології) правильно представляють стан кодової бази, система логічного виводу може допомогти вивести неявні або невідомі відношення/аксіоми та додати їх до початкової онтології. Така здатність пропонованої системи дозволяє додавати правила до бази знань (наприклад, шаблони, що описують антипатерни, та підходи до їх виправлення), щоб видаляти, змінювати або генерувати код, який буде компілюватися або інтерпретуватися.

Щоб перевірити запропоновану концепцію, було розроблено структуру простої бази знань і побудовано прототип системи отримання фактів про код з файлу початкового коду. Було досліджено і обґрунтовано архітектуру такої системи та основні моменти реалізації. У результаті фрагмент коду був перетворений у просту БЗ. Неявні та невідомі аксіоми були виведені системою логічного виводу і успішно додані до початкової бази знань.

Ця проста структура бази знань, яка використовується, не охоплює всіх можливих зв’язків і сутностей коду, тому для того, щоб система могла виводити складні дані (наприклад, відповідність протоколу), необхідно розробити набір аксіом, які дозволять системі логічного виводу зробити це. Подальші дослідження будуть присвячені розробці більш складної семантичної основи для вилучення та маніпулювання знаннями з початкового коду.

РОЗДІЛ 6. ВИКОРИСТАННЯ ЛОГІЧНИХ ПРАВИЛ В ПРОЦЕСІ РЕФАКТОРИНГУ

6.1 Опис методу автоматизованого рефакторингу із використанням логічних правил

Перетворення початкового коду в базу знань, збудовану навколо онтології, дає можливість формалізувати початковий код, зробити його зрозумілим для програмних систем, які оперують тими ж поняттями (сутностями), що присутні у коді. Хоча власне онтології мають безліч переваг у різноманітті доступних типів властивостей та відношень, деякі з них, втім, у своєму використанні можуть призвести до недостатньої гнучкості та накладання певних обмежень, що в свою чергу робить неможливим реалізацію властивостей систем аналізу та рефакторингу початкового коду.

Візьмемо до прикладу ситуацію, коли системі рефакторингу необхідно здійснити визначення факту реалізації класом певного інтерфейсу (протоколу). Розробник чи компілятор зважає на наявність реалізованих у конкретному класі методів і властивостей, що вимагаються інтерфейсом (протоколом), а також на декларацію реалізації цього інтерфейсу. Для онтології виконання такого завдання власне засобами OWL є неможливим через обмеження системи продукції знань (reasoner) здійснювати обчислення функціональних залежностей.

Для вирішення зазначеної вище проблеми використаємо додаткові правила SWRL. SWRL дозволяє виразити складні логічні умови і правила з метою використання їх для генерації нових знань на основі вже існуючих даних.

Визначення методу опрацювання моделі початкового коду із застосуванням правил для аналізу та рефакторингу:

Одним із завдань дисертації є розробка набору перетворень $g \in G$, $g: Y \to Y'$ із застосуванням засобів OWL та S(Q)WRL, таким що $\forall y \in Y \; \exists g \; g(y) = y'$. Таким чином як додатковий крок в процедурі автоматизованого аналізу і

рефакторингу в методі, запропонованому в Розділі 5, будемо використовувати розроблений набір перетворень. Як результат – модель вихідного коду буде опрацьовуватись системою логічного виведення на основі правил. Схематично такий процес показано на Рисунок 32.

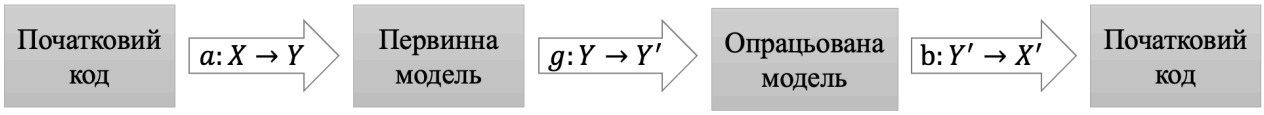

Рисунок 32. Формалізований процес рефакторингу з використанням запропонованого методу

У результаті – блок-схема методу з Розділу 5 із врахуванням кроків методу опрацювання моделі на основі правил зображено на Рисунок 33.

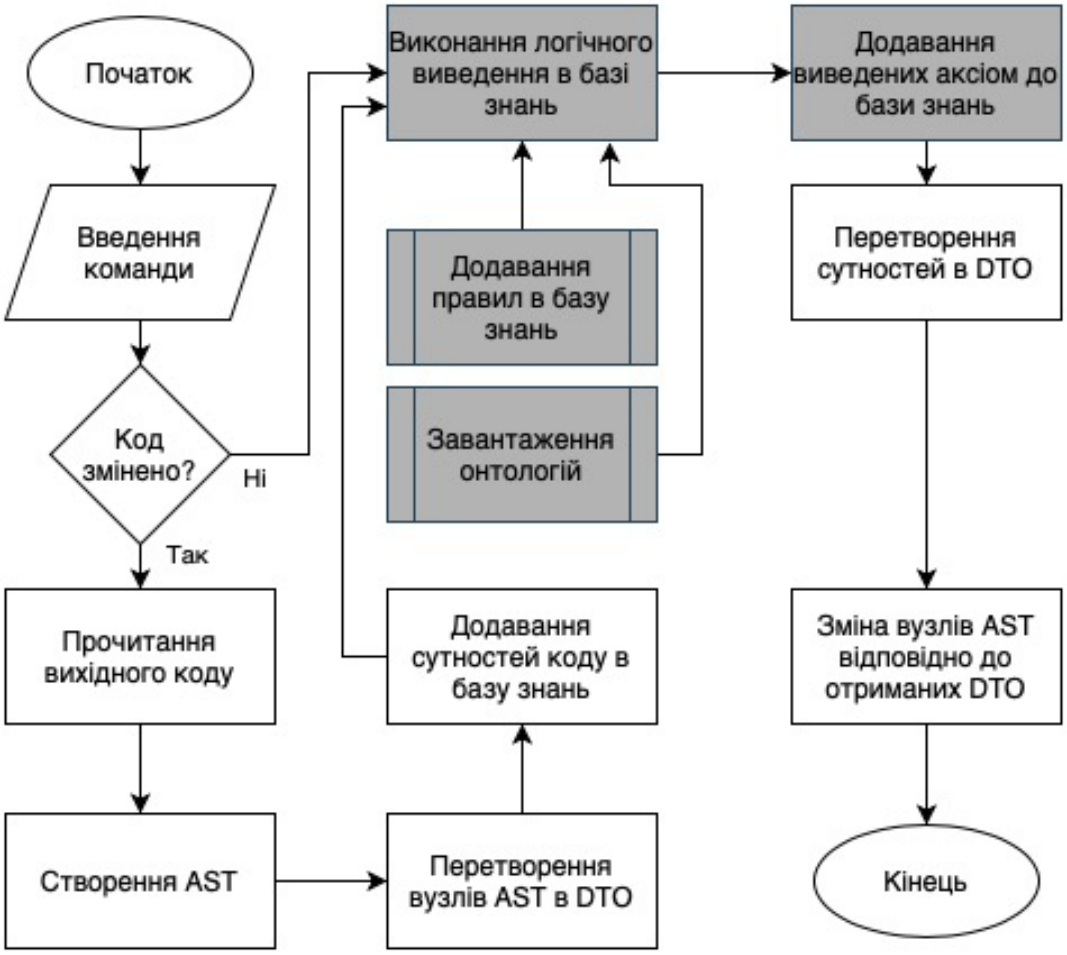

Рисунок 33. Блок-схема методу рефакторингу з використанням БЗ, розширеного запропонованим методом здійснення перетворення моделі на основі логічних правил

Крок, що доданий до методу з Розділу 5, – процес додавання правил в базу знань. Він є окремою процедурою зміни бази знань, не приймає на вхід жодних даних. На виході процесу – набір правил і запитів S(Q)WRL, що оперують тими ж поняттями, що й онтологія, на основі якої будується модель початкового коду. Це дозволяє без додаткових складнощів створювати описові правила для маніпуляції даними, що містяться в базі знань.

Головною метою SWRL є розширення можливостей OWL, дозволяючи визначати більш складні умови і відношення між даними в онтологіях. Це зроблено шляхом додавання правил імплікації, які доповнюють стандартні конструкції OWL, такі як підкласи, об'єктні властивості та екземпляри.

SWRL базується на комбінації логічних мов, таких як логіка першого порядку (FOL – First-Order Logic) і логіка правил.

SWRL дозволяє створювати складніші моделі знань, що допомагає вирішувати складні завдання у будь-яких системах, що використовують семантичні технології.

Метод передбачає два типи логічних правил:

- власне правила – набори правил, об’єднані метою однієї дії рефакторингу. Такий набір правил (або ж одне правило, якщо рефакторинг простий) описують екземпляри, їх властивості і функціональні залежності між ними з точки зору кроків, які необхідно виконати для досягнення поставленої цілі конкретного рефакторингу. Ці правила знаходяться в базі знань і застосовуються до моделі початкового коду як тільки буде запущено процес логічного виведення (без додаткового втручання користувача);
- запити – такі логічні правила будуть відповідати можливостям розробленої системи автоматизованого рефакторингу: знаходити екземпляри, що відповідають певним критеріям, шукати властивості та інше. З одного боку – такі запити можуть одразу надходити від користувача, оскільки сутності бази знань представляють елементи початкового коду, тому створення таких запитів користувачами з урахуванням синтаксису правил не є

проблематичним. З іншого боку – система рефакторингу може приховувати синтаксис правил від користувача і вже динамічно транслювати запит від користувача (в обумовленому форматі) в правило і передавати його на виконання в базу знань.

Такий набір конкретних правил для пошуку і видалення антипатернів шляхом здійснення рефакторингу можемо вважати системою, що продукує знання (продукційною системою).

6.2 Опис моделі початкового коду, що базується на онтології

Визначимо базу знань про початковий код як набір TBox та ABox аксіом. Основа моделі – онтологія про об’єктно-орієнтовану мову програмування (Atzeni & Atzori, 2017).

Перший компонент моделі – статичний і включає незмінну базову онтологію, що містить класи, атрибути, відношення про:

- спільний синтаксис об’єктно-орієнтованих мов програмування та специфічних конструкцій конкретної мови чи мов програмування (у нашому випадку – мови Swift);
- структуру програмного продукту («модулі», підсистеми, сервіси, файли коду, функції продукту, користувачі, звіти, тести тощо у зв'язку з поняттями про код);
- поняття прикладної предметної області, до якої відноситься продукт (для зв'язку функцій продукту, модулів з поняттями предметної області типу «фінансова транзакція», «покупець», «чек», «товар» тощо).

Також важливим для роботи методів є опис властивостей тих чи інших сутностей моделі з точки зору аксіом еквівалентності, ієрархічності для можливості здійснення автоматизованого рефакторингу на основі класифікації чи встановлення відношень. Наприклад, необхідно, щоб клас «PublicFunction» містив аксіому про те, що усі методи, які мають властивість «hasPublicModifier» зі значенням «true», є екземплярами цього класу.

Це все дасть змогу виражати запити (завдання) на аналіз та рефакторинг не просто використовуючи поняття початкового коду такі як «клас», «метод», а й уточнювати стосунок до областей використовуючи конструкції типу «що має відношення до користувачів» або «що має відношення до транзакцій».

Наприклад, «в усі класи, що мають відношення до транзакцій, додати метод accept(visitor)» (для реалізації патерну Відвідувач для документування поточного стану як частина гіпотетичного складного сценарію рефакторингу).

Інший приклад – уявімо ситуацію, що розробник отримав завдання здійснити рефакторинг частини початкового коду, що стосується генерації звітів. Код не поділений на модулі, але функції, що створюють звіти позначені особливим чином. Розробнику необхідно знайти усі такі функції, які мають кількість аргументів більшу, ніж 4. Для цього йому потрібно лише написати правило яке б казало «усі функції, що стосуються генерації звітів і мають більше ніж 4 аргументи» (реалізацію правила розглянуто далі).

Другий компонент моделі – динамічний. Він складається із набору ABox аксіом, які створюються на основі конкретного початкового коду в результаті синтаксичного аналізу цього коду. Ці аксіоми будуть додавати екземпляри до моделі, які відповідатимуть конкретним синтаксичним конструкціям, а також знання про приналежність таких конструкцій до певного модуля, класу, зони відповідальності на основі коментарів чи інших додаткових позначень.

Схематично таку структуру моделі зображено на Рисунок 34.

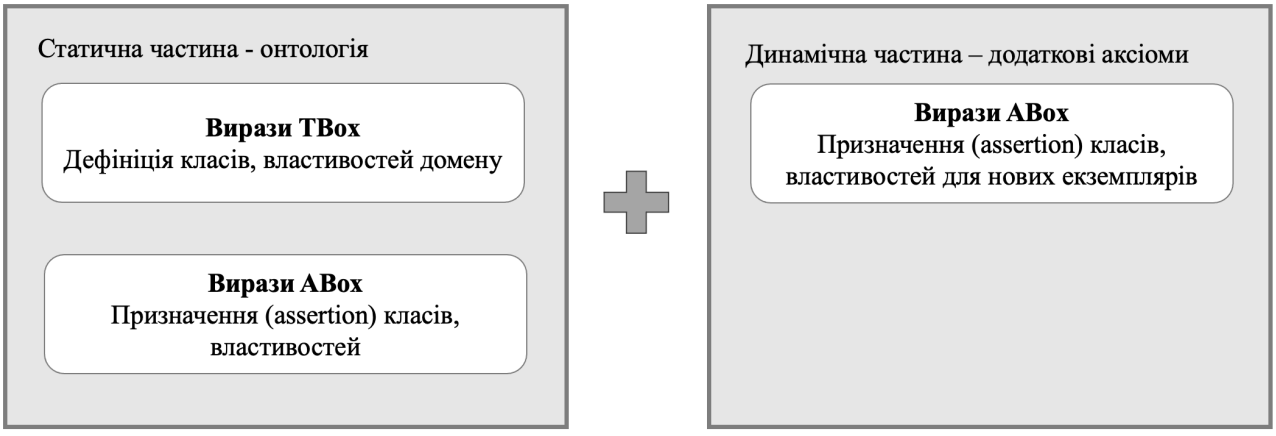

Рисунок 34. Схематична структура моделі

Таким чином можемо вважати, що моделювання конкретного початкового коду здійснюється шляхом моделювання формалізованого представлення знань.

Розглянемо конкретні приклади сутностей (найбільш значущі), які містяться в моделі. Зведені дані представлено в Таблиця 3.

Таблиця 3. Список основних сутностей в моделі

Класи	**Властивості**	**Екземпляри**
Онтологія про об’єктно-орієнтовану мову програмування		
Class declaration statement, Field, Local variable, Parameter, Complex type, Primitive type, Class, Function, Property, Interface, Modifier, Statement, Wildcard, Library, *інші.*	belongsTo, conformsTo, equalsTo, isInFunction, references, returns, invokes, hasProperty, hasModifier, hasGenericType, hasPart, isIn, argumentCount, sourceCodeReference, returnType, name, *інші.*	Float, Int, Boolean, Private, Public, Static, Final, *інші.*
Онтологія структури програмного продукту		
Scope, Area, Module, Functionality, Business unit, *інші.*	belongsTo, appliesTo, worksIn, *інші.*	Transaction, Logging, Print, Business, Unit testing, *інші.*
Правила розширених синтаксичних конструкцій		
–	–	Існуючим екземплярам БЗ додаються нові відношення та властивості на основі заданих правил
Динамічні елементи моделі		
–	–	Екземпляри, створені на основі конкретних синтаксичних структур початкового коду

На Рисунок 35 зображено метрику по кількості класів, властивостей і екземплярів у статичній частині моделі. Розмір динамічної частини (кількість екземплярів) буде напряму залежати від розміру початкового коду, який буде представлятись моделлю.

Metrics

Axiom	7 149
Logical axiom count	6 318
Declaration axioms count	469
Class count	67
Object property count	101
Data property count	18
Individual count	284
Annotation Property count	3

Рисунок 35. Статистика статичної частини моделі

Наведені показники є лише частиною усіх загальних показників, проте саме зазначені – характеризують незмінну частину моделі (усі TBox аксіоми).

6.3 Дослідження процедури використання правил в процесі рефакторингу

Для перевірки можливості SWRL вирішити поставлену задачу, додамо до раніше розробленої схеми бази знань клас Interface, а також властивість conformsTo, що буде вказувати на реалізацію екземпляром певного інтерфейсу, і зворотню до неї supportedBy.

```
protocol Say {
    func say() -> String
}

class Animal {
    let age: Int

    init() {
        age = 0
    }

    func say() -> String {
        return ""
    }
}
```

Рисунок 36. Початковий код

Для прикладу розглянемо випадок перетворення в онтологію початкового коду, зображеного на Рисунок 36.

Після виконання перетворення коду в факти, що додані в базу знань, для інтерфейсу (протокол у початковому коді) буде створено екземпляр наступного типу (Рисунок 37):

```
<!-- owl:Ontology#15488467-b91b-40bd-99af-5a19b553846f -->

<owl:Thing rdf:about="owl:Ontology#15488467-b91b-40bd-99af-5a19b553846f">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
    <rdf:type rdf:resource="owl:Ontology#Interface"/>
    <hasMethod rdf:resource="owl:Ontology#9f30cdf7-3d38-41a3-a08d-a7b465292f01"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Say</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(8:1,10:2)</sourceCodeReference>
</owl:Thing>
```

Рисунок 37. Опис екземпляру інтерфейсу

Для початкового коду, що зображено на рисунку, буде створено наступний екземпляр класу (Рисунок 38):

```
<!-- owl:Ontology#f6283464-d6de-4203-b638-a4158ed174f0 -->

<owl:Thing rdf:about="owl:Ontology#f6283464-d6de-4203-b638-a4158ed174f0">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
    <rdf:type rdf:resource="owl:Ontology#Class"/>
    <hasMethod rdf:resource="owl:Ontology#578f62ad-7a0f-4bde-a0d8-c91a30d25cdb"/>
    <hasProperty rdf:resource="owl:Ontology#87000b06-c136-4e07-a0cd-894ab4b69c33"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Animal</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(12:1,22:2)</sourceCodeReference>
</owl:Thing>
```

Рисунок 38. Опис екземпляру класу

Як бачимо, екземпляр не містить жодної інформації про те, що даний клас реалізує інтерфейс.

Оскільки обидва екземпляри містять інформацію про те, що вони мають однакову функцію, можна зробити висновок, що даний клас реалізує інтерфейс. Однак, ці знання не можуть бути отримані з онтології OWL, оскільки навіть наявність аксіоми про те, що обидва екземпляри вищезгаданих методів є одним об’єктом, не дає можливості системі логічного виводу створити знання про функціональну залежність.

Для розв’язання даної проблеми скористаємось правилами SWRL та запитами мовою SQWRL, що разом позначимо як S(Q)WRL.

Для того, щоб описати, що наявність одних і тих же ж методів у інтерфейсі та класі свідчить про реалізацію даним класом інтерфейсу, застосуємо таке правило (Рисунок 39):

```
Class(?cl) ^ hasMethod(?cl, ?x) ^ Interface(?pr) ^ hasMethod(?pr, ?y) ^ name(?x, ?x_name) ^ name(?y, ?y_name) ^ swrlb:equal(?x_name, ?y_name) ^ returnType(?x, ?x_type) ^ returnType(?y, ?y_type) ^ swrlb:equal(?x_type, ?y_type) -> conformsTo(?cl, ?pr)
```

Рисунок 39. Правило conformsTo

Правило говорить про те, що якщо у класу «cl» є метод «x», і цей метод є частиною інтерфейсу «pr», який також має метод «y», і якщо ім'я методу «x» дорівнює імені методу «y», і тип повернення «x» співпадає з типом повернення «y», то клас «cl» відповідає інтерфейсу «pr». Умови для застосування правила:

?cl – змінна, яка представляє клас.

?x – змінна, яка представляє метод «x».

?pr – змінна, яка представляє інтерфейс.

?y – змінна, яка представляє метод «y».

?x_name – змінна, яка представляє ім'я методу «x».

?y_name – змінна, яка представляє ім'я методу «y».

?x_type – змінна, яка представляє тип повернення методу «x».

?y_type – змінна, яка представляє тип повернення методу «y».

Дії, які виконуються, якщо умови виконуються:

Система додає твердження «conformsTo(?cl, ?pr)» до своєї бази знань, що означає, що клас «cl» відповідає інтерфейсу «pr».

Після застосування наведеного правила для попереднього початкового коду отримаємо такий результат (Рисунок 40):

```
<!-- owl:Ontology#2144fe9e-201c-4289-99e8-290495d1da4e -->

<owl:Thing rdf:about="owl:Ontology#2144fe9e-201c-4289-99e8-290495d1da4e">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
    <rdf:type rdf:resource="owl:Ontology#Class"/>
    <hasProperty rdf:resource="owl:Ontology#e2aecabb-1ab2-4074-aec9-bfc9ec5c8406"/>
    <conformsTo rdf:resource="owl:Ontology#0e968089-c3f0-4799-98fc-56fdce596834"/>
    <hasMethod rdf:resource="owl:Ontology#296ea3e9-5bca-4c9d-b16e-d2d1b9bd93bc"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Animal</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(12:1,22:2)</sourceCodeReference>
</owl:Thing>
```

Рисунок 40. Опис екземпляру після виконання правила

Як бачимо, до індивіду класу було додано властивість conformsTo, що вказує на реалізацію цим класом відповідного інтерфейсу (Рисунок 41).

```
protocol Say {
    func say(name: String) -> String
}
```

Рисунок 41. Протокол з параметром

До попереднього початкового коду внесемо зміну – додамо до методу, що є членом протоколу (інтерфейсу) параметр. З точки зору раніше написаного SWRL правила, методи в протоколі (інтерфейс) та в класі (його реалізація) все ще однакові, тому після запуску системи перетворення початкового коду в базу знань з виконанням правил та продукуванням знань, отримаємо результат, ідентичний попередньому. Рисунок 42 підтверджує зроблений висновок.

```
<!-- owl:Ontology#55df1896-0e1e-4b66-b626-ce76476c2810 -->

<owl:Thing rdf:about="owl:Ontology#55df1896-0e1e-4b66-b626-ce76476c2810">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
    <rdf:type rdf:resource="owl:Ontology#Class"/>
    <conformsTo rdf:resource="owl:Ontology#bccc9268-a3ed-4388-8c93-822eb56f46f3"/>
    <hasMethod rdf:resource="owl:Ontology#221bea71-8a9f-449b-8213-e1744d6a4779"/>
    <hasProperty rdf:resource="owl:Ontology#9fbf63e4-a7fc-4bc7-a34b-140f021c3257"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Animal</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(12:1,22:2)</sourceCodeReference>
</owl:Thing>
```

Рисунок 42. Опис екземпляру після змін до початкового коду

Для того, аби запобігти такій ситуації, необхідно більш точно визначити подібність двох методів – в інтерфейсі та в реалізації. Для цього пропонується додати правила SWRL, які будуть порівнювати екземпляри, що позначають методи, і використовувати додаткову властивість, щоб позначити однакові екземпляри. Цією властивістю може бути стандартне позначення OWL sameAs, однак накладання такої властивості (чи навіть обмеження у певних випадках) призведе до утотожнення їхніх попередніх властивостей (такий як sourceCodeReference), що є небажаним у контексті початкового коду. Приклад такого логічного виведення зображено на Рисунок 43.

Для вирішення проблеми введемо в онтологію власне поняття тотожності, яке буде використовуватись усіма сутностями і буде означати власне синтаксичну подібність, яка потрібна для порівняння сутностей початкового коду, а не зазначення тотожності власне індивідів онтології. Варто зазначити, що дану властивість слід зазначити як обернену до самої себе.

```
<!-- owl:Ontology#4c4b4745-1dbd-4e3d-a422-acc73c88234d -->

<owl:Thing rdf:about="owl:Ontology#4c4b4745-1dbd-4e3d-a422-acc73c88234d">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
    <rdf:type rdf:resource="owl:Ontology#Property"/>
    <isInFunction rdf:resource="owl:Ontology#3ecc3c58-ab8b-458f-83d2-191cd8512ded"/>
    <isInFunction rdf:resource="owl:Ontology#38f93ef4-1114-4c4b-842f-5f5056d671ca"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">name: String</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(19:14,19:26)</sourceCodeReference>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(9:14,9:26)</sourceCodeReference>
    <type rdf:datatype="http://www.w3.org/2001/XMLSchema#string">String</type>
</owl:Thing>
```

Рисунок 43. Приклад екземпляру із небажаними властивостями

Враховуючи вищенаведені роздуми, правило для позначення однакових властивостей (properties) у початковому коді буде виглядати так, як показано на Рисунок 44.

```
Ontology:Property(?p1) ^ Ontology:name(?p1, ?p1_name) ^ Ontology:type(?p1, ?p1_type) ^ Ontology:Property(?p2)
^ Ontology:name(?p2, ?p2_name) ^ Ontology:type(?p2, ?p2_type) ^ swrlb:equal(?p1_name, ?p2_name) ^
swrlb:equal(?p1_type, ?p2_type) -> Ontology:equalsTo(?p1, ?p2)
```

Рисунок 44. Вдосконалене правило equalsTo

Розглянемо кожну частину цього правила:

Ontology:Property(?p1)^Ontology:name(?p1,?p1_name)^Ontology:type(?p1, ?p1_type): ця умова описує, що існує об'єктна властивість *?p1*, у якої є значення *?p1_name* (назва) та *?p1_type* (тип) для властивостей name та type відповідно.

Ontology:Property(?p2)^Ontology:name(?p2,?p2_name)^Ontology:type(?p2, ?p2_type): це аналогічна умова для другої об'єктної властивості *?p2* з її значеннями *?p2_name* (назва) та *?p2_type* (тип).

swrlb:equal(?p1_name, ?p2_name): ця умова перевіряє, що значення *?p1_name* та *?p2_name* рівні.

swrlb:equal(?p1_type, ?p2_type): ця умова перевіряє, що значення *?p1_type* та *?p2_type* рівні.

Якщо всі ці умови виконуються, то наступний висновок активується:

Ontology:equalsTo(?p1, ?p2).

Він створює нову об'єктну властивість *equalsTo*, яка пов'язує об'єктні властивості *?p1* та *?p2* у випадку, коли їхні назва та тип співпадають.

Отже, дане SWRL правило виражає логічний шаблон, за якого властивості (properties) з однаковою назвою та типом вважаються рівними і пов'язуються новою об'єктною властивістю «equalsTo».

Результат виконання даного правила зображено на Рисунок 45. Як можна помітити на рисунку, екземпляр має дві властивості, що вказують на подібність, але одна з них вказує на екземпляр, який має такий же ж ідентифікатор, як і поточний – тобто вказує сама на себе. Такий висновок стає очевидним, якщо врахувати, що об’єктна властивість equalsTo вказана як така, що обернена сама собі, а отже й екземпляр буде подібним собі.

```
<!-- owl:Ontology#f52aa436-5617-44ce-bb4b-234ced56e1f2 -->

<owl:Thing rdf:about="owl:Ontology#f52aa436-5617-44ce-bb4b-234ced56e1f2">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
    <rdf:type rdf:resource="owl:Ontology#Property"/>
    <equalsTo rdf:resource="owl:Ontology#a234752d-b690-4cd4-8414-4ac012b79923"/>
    <equalsTo rdf:resource="owl:Ontology#f52aa436-5617-44ce-bb4b-234ced56e1f2"/>
    <isInFunction rdf:resource="owl:Ontology#85a59939-2452-4970-8ac5-4a6d414a2475"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">name: String</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(9:14,9:26)</sourceCodeReference>
    <type rdf:datatype="http://www.w3.org/2001/XMLSchema#string">String</type>
</owl:Thing>
```

Рисунок 45. Екземпляр Property із властивістю equalsTo

Додамо правило (Рисунок 46) для порівняння методів, яке буде використовувати раніше додану властивість про подібність параметрів.

```
Ontology:Function(?f1) ^ Ontology:name(?f1, ?f1_name) ^ Ontology:returnType(?f1, ?f1_type) ^
Ontology:hasArgument(?f1, ?f1_arg) ^ Ontology:Function(?f2) ^ Ontology:name(?f2, ?f2_name) ^
Ontology:returnType(?f2, ?f2_type) ^ Ontology:hasArgument(?f2, ?f2_arg) ^ swrlb:equal(?f1_name, ?f2_name) ^
swrlb:equal(?f1_type, ?f2_type) ^ Ontology:equalsTo(?f1_arg, ?f2_arg) -> Ontology:equalsTo(?f1, ?f2)
```

Рисунок 46. Ускладнене правило equalsTo

Це SWRL правило вимагає, щоб дві функції (?f1 та ?f2), які мають однакові імена (name), типи повернення (returnType) та аргументи (hasArgument), були визнані рівними, вводячи нове відношення equalsTo.

Варто зауважити, що через особливість SWRL додане правило буде застосовуватись до всіх методів, які мають хоча б один параметр. Очевидно, що методи без параметрів теж можуть бути подібними. Для вирішення цієї проблеми додамо ще одне правило, яке буде працювати для методів, що не містять параметрів.

Для цього додамо нову властивість даних (data property) argumentCount (Рисунок 47), яка буде притаманною для екземплярів класу Function. Це потрібно для того, аби мати можливість однозначно і просто визначити наявність параметрів у методі. Оскільки онтології мають обмеження відкритого світу (припущення може стати неправдивим при додаванні нових даних), монотонності (нові знання можуть додаватись, але не видалятись), базуються лише на ствердних аксіомах, а SWRL не допускає заперечення як негативного ствердження саме через вищенаведені обмеження, визначити наявність параметрів у методі через перелічення наявних властивостей hasArgument не можливо.

```
Ontology:Function(?f1) ^ Ontology:argumentCount(?f2, ?f2_arg) ^ Ontology:Function(?f2) ^ swrlb:equal(?f1_arg, 0)
^ Ontology:name(?f1, ?f1_name) ^ Ontology:returnType(?f1, ?f1_type) ^ swrlb:equal(?f2_arg, 0) ^
Ontology:argumentCount(?f1, ?f1_arg) ^ swrlb:equal(?f1_name, ?f2_name) ^ swrlb:equal(?f1_type, ?f2_type) ^
Ontology:name(?f2, ?f2_name) ^ Ontology:returnType(?f2, ?f2_type) -> Ontology:equalsTo(?f1, ?f2)
```

Рисунок 47. Правило equalsTo з властивістю argumentCount

Також необхідно оновити попереднє правило (Рисунок 47) із врахуванням наявності нової властивості argumentCount – методи можуть бути подібними тоді і тільки тоді, коли кількість їхніх аргументів співпадає. Виразимо це ствердження у правилі (Рисунок 48).

```
Ontology:Function(?f1) ^ Ontology:name(?f1, ?f1_name) ^ Ontology:returnType(?f1, ?f1_type) ^
Ontology:hasArgument(?f1, ?f1_arg) ^ Ontology:argumentCount(?f1, ?f1_count) ^ Ontology:Function(?f2) ^
Ontology:name(?f2, ?f2_name) ^ Ontology:returnType(?f2, ?f2_type) ^ Ontology:hasArgument(?f2, ?f2_arg) ^
Ontology:argumentCount(?f2, ?f2_count) ^ swrlb:equal(?f1_name, ?f2_name) ^ swrlb:equal(?f1_type, ?f2_type) ^
Ontology:equalsTo(?f1_arg, ?f2_arg) ^ swrlb:equal(?f1_count, ?f2_count) -> Ontology:equalsTo(?f1, ?f2)
```

Рисунок 48. Додаткове правило equalsTo з властивістю argumentCount

При обробці раніше зазначеного початкового коду отримуємо наступний результат (Рисунок 49).

```
<!-- owl:Ontology#8779b482-dc28-46b3-a228-38cdb615592b -->

<owl:Thing rdf:about="owl:Ontology#8779b482-dc28-46b3-a228-38cdb615592b">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
    <rdf:type rdf:resource="owl:Ontology#Function"/>
    <equalsTo rdf:resource="owl:Ontology#eca47d1a-5155-4eeb-9702-f50b8dfe60e8"/>
    <hasArgument rdf:resource="owl:Ontology#8fd72696-512d-44d8-ae3e-fd99603c0f8c"/>
    <equalsTo rdf:resource="owl:Ontology#8779b482-dc28-46b3-a228-38cdb615592b"/>
    <isIn rdf:resource="owl:Ontology#9e098efc-929c-4ad4-8fe7-062b4a0aef81"/>
    <argumentCount rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">1</argumentCount>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">say</name>
    <returnType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">String</returnType>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(9:5,9:37)</sourceCodeReference>
</owl:Thing>
```

Рисунок 49. Екземпляр функції із хибним висновком

Як бачимо, метод визначений як подібний до іншого методу, що має такий же ж аргумент. Для того, аби виправити ситуацію із неправильним визначенням реалізації інтерфейсу класом, змінимо попереднє правило про наявність властивості conformsTo і приведемо його до такого вигляду, як показано на Рисунок 50.

```
Ontology:Class(?cl) ^ Ontology:hasMethod(?cl, ?x) ^ Ontology:Interface(?pr) ^ Ontology:hasMethod(?pr, ?y) ^ Ontology:equalsTo(?x, ?y) -> Ontology:conformsTo(?cl, ?pr)
```

Рисунок 50. Оновлене правило conformsTo

Як бачимо, перевірка на тотожність назви та типу повернення, а також відсутність перевірки на наявні параметри, у попередньому правилі були замінені на єдину перевірку equalsTo.

```
protocol Say {
    func say(name: String) -> String
}

class Animal {
    let age: Int

    init() {
        age = 0
    }

    func say(name: String) -> String{
        return ""
    }
}
```

Рисунок 51. Тестовий початковий код

Для початкового коду, що на Рисунок 51, отримаємо наступний результат (Рисунок 52):

```
<!-- owl:Ontology#77fda00c-6769-4723-9ccb-bce81fcfef79 -->

<owl:Thing rdf:about="owl:Ontology#77fda00c-6769-4723-9ccb-bce81fcfef79">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
    <rdf:type rdf:resource="owl:Ontology#Class"/>
    <conformsTo rdf:resource="owl:Ontology#bcc8bbd9-8d5e-4aba-8cf9-dca2a36479e0"/>
    <hasProperty rdf:resource="owl:Ontology#739d78c7-e7c3-4774-b2c2-eae92b4d9b4b"/>
    <hasMethod rdf:resource="owl:Ontology#805c2827-a32f-443d-9ded-164e9f4de159"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Animal</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(12:1,22:2)</sourceCodeReference>
</owl:Thing>
```

Рисунок 52. Опис екземпляру класу після виконання усіх правил

Змінимо початковий код так, щоб реалізація протоколу не відбувалась Рисунок 53.

```
protocol Say {
    func say(name: String) -> String
}

class Animal {
    let age: Int

    init() {
        age = 0
    }

    func say() -> String{
        return ""
    }
}
```

Рисунок 53. Початковий код без реалізації протоколу

У такому випадку система згенерує наступну відповідь:

```
<!-- owl:Ontology#87b5310f-38e3-48cd-a6ff-fb1e88dbc316 -->

<owl:Thing rdf:about="owl:Ontology#87b5310f-38e3-48cd-a6ff-fb1e88dbc316">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#NamedIndividual"/>
    <rdf:type rdf:resource="owl:Ontology#Class"/>
    <hasProperty rdf:resource="owl:Ontology#b0a1a232-abce-4af1-be09-3e01d3988b0d"/>
    <hasMethod rdf:resource="owl:Ontology#ecc83e16-afd1-4860-9326-26e175f25f9d"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Animal</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(12:1,22:2)</sourceCodeReference>
</owl:Thing>
```

Рисунок 54. Опис екземпляру класу для початкового коду без реалізації протоколу

Приклад із реалізацією протоколу має на меті показати можливість логічного виведення нових знань на основі початкового коду, базуючись на певному наборі функціональних правил. Очевидно, що приклад є спрощеним і не покриває різні крайові умови, такі як наявність кількох різних методів в

інтерфейсі. Однак наведені правила доводять, що розширення і додавання умов тепер може здійснюватися незалежно від синтаксису мови програмування і не вимагає знань про конкретну технологію.

Продукування і додавання нових знань, що базується на властивості монотонності онтологій, не завжди покриватиме задачі, які постають перед розробником програмного забезпечення під час рефакторингу. Часто доводиться вносити зміни в структуру початкового коду, додавати чи видаляти сутності. Власне онтологія та продукційні правила SWRL не передбачають можливість видалення знань чи їх модифікацію (що протирічить основним принципам онтології). Для зміни структури бази знань необхідне залучення зовнішнього чинника, як-от системи управління базою знань. Вона повинна міститись в одній логічній функціональній одиниці (сервіс, функція у serverless архітектурі, тощо) із базою знань і опрацьовувати її, коли вона вже сформована. При цьому система буде лише здійснювати наперед описані запити до бази знань і змінювати її відповідно до потреби сценарію. Такий підхід не протирічить ходу дослідження і основній ідеї формалізації початкового коду і узагальнення для здійснення рефакторингу, оскільки сценарії системи управління базою знань будуть накопичуватись і будуть незмінними відносно мови програмування чи технології, що використовуються у початковому коді, який обробляється.

Для прикладу розглянемо один із популярних антипатернів – довгий список параметрів. Як один із варіантів розв’язання проблеми пропонується заміна параметрів одним об’єктом, який буде містити в собі усі необхідні властивості.

Використовуючи попередні напрацювання, опишемо правила і сценарії, необхідні для виконання цього процесу в автоматичному режимі в базі знань.

Для здійснення запитів до бази знань (онтології) скористаємось розширенням SWRL – SQWRL. Відмінність SQWRL полягає в тому, що вона дозволяє також формулювати запити, в яких можна поєднувати питання з логічними правилами для отримання більш складних результатів.

Запити, написані на SQWRL, допомагають знаходити, вилучати та аналізувати дані в онтологіях. Такі запити можуть бути корисні для різноманітних завдань, включаючи виявлення зв'язків між об'єктами, виконання логічних операцій над даними та здійснення запитів з різних точок зору.

Для початку визначимо усі методи, які містять більше, ніж 5 параметрів. Для цього скористаємось раніше доданою властивістю argumentCount та запитом SQWRL.

Оскільки мова запитів SQWRL дозволяє об’єднувати умови і отримувати у запиті набори даних (відповідь на запит міститиме усі можливі поєднання даних, доступні в базі знань, що відповідають зазначеним умовам), одразу в запиті (Рисунок 55) отримаємо методи та їх аргументи.

```
Ontology:Function(?f) ^ Ontology:argumentCount(?f, ?count) ^ swrlb:greaterThan(?count, 4) ^
Ontology:hasArgument(?f, ?a) -> sqwrl:select(?f, ?a)
```

Рисунок 55. Запит для отримання всіх функцій, у яких кількість аргументів більша за 4

Розглянемо кожну частину правила:

1. *Ontology:Function(?f)* – це частина умови (антецедента) правила. Вона встановлює, що сутність *?f* є екземпляром класу *Ontology:Function*, тобто це якась функція або метод в онтології.

2. *Ontology:argumentCount(?f, ?count)* – це також умова. Вона встановлює, що для сутності *?f* існує властивість *Ontology:argumentCount*, яка пов'язує її з числом аргументів, представленим як *?count*.

3. *swrlb:greaterThan(?count, 4)* – це ще одна умова. Вона стверджує, що число *?count* повинно бути більше ніж 4.

4. *Ontology:hasArgument(?f, ?a)* – це також частина умови. Вона встановлює, що сутність *?f* пов'язана з сутністю *?a* через властивість *Ontology:hasArgument*, що вказує на те, що функція *?f* має аргумент *?a*.

5. -> – це символ наслідку, який розділяє умову та результат правила.

6. *sqwrl:select(?f, ?a)* – консеквент запиту. Він вказує на те, що при виконанні всіх попередніх умов буде виконаний запит на вибірку (select) для змінних *?f* та *?a*.

Підсумовуючи, це правило описує ситуацію, коли в базі знань є екземпляри класів, що описують такі конструкції в коді як «функція» або «метод» (*Ontology:Function*), у яких кількість аргументів (*Ontology:argumentCount*) більше ніж 4 (*swrlb:greaterThan(?count, 4)*), і для цих елементів визначені аргументи через властивість *Ontology:hasArgument*. Результатом виконання запиту буде вибірка всіх таких екземплярів і пов'язаних з ними аргументів.

Варто зауважити, що такий запит можна легко модифікувати і підлаштовувати під конкретні потреби, оскільки модель початкового коду побудована на онтології, що містить додаткову інформацію про код. Це дає змогу звузити область застосування правила. Наприклад, на Рисунок 56 зображено модифіковане попереднє правило, яке буде повертати усі такі функції, що стосуються функціоналу генерації звітності (завдяки додаванню перевірки на наявності відношення belongsTo функції і попередньо визначеної області ReportingFuncArea).

```
Ontology:Function(?f) ^ Ontology:ReportingFuncArea(?area) ^ Ontology:argumentCount(?f, ?count) ^
swrlb:greaterThan(?count, 4) ^ Ontology:hasArgument(?f, ?a) ^ belongsTo(?f, ?area) -> sqwrl:select(?f, ?a)
```

Рисунок 56. Запит для отримання усіх функції із кількістю аргументів більше, ніж 4, що стосуються генерації звітів

Надалі здійснимо обробку результатів, отриманих при виконанні запиту.

Представлений фрагмент Java-коду (Рисунок 57) є продовженням операцій з результатами запиту SQWRL для обробки отриманих екземплярів.

```
while (longResult.next()) {
    IRI funcIRI = longResult.getNamedIndividual(columnName:"f").getIRI();
    OWLNamedIndividual funcInd = dataFactory.getOWLNamedIndividual(funcIRI);
    IRI paramIRI = longResult.getNamedIndividual(columnName:"a").getIRI();
    OWLNamedIndividual paramInd = dataFactory.getOWLNamedIndividual(paramIRI);
    List<OWLNamedIndividual> temp = new ArrayList<OWLNamedIndividual>();
    if (args.get(funcIRI) != null) {
        temp.addAll(args.get(funcIRI));
    }
    temp.add(paramInd);
    args.put(funcIRI, temp);
    manager.removeAxioms(ontology, getAxiomsBetweenIndividuals(funcInd, paramInd));
}
```

Рисунок 57. Код для обробки отриманих даних як результат виконання запиту

Розглянемо кожен крок:

1. *while (longResult.next()) { ... }:* цикл *while* перебирає всі результати запиту *longResult*, доки є наступний результат.

2. Отримання IRI індивідуальних елементів з результату:

IRI funcIRI = longResult.getNamedIndividual("f").getIRI(): отримуємо IRI (ідентифікатор) індивідуального елемента, який був помічений як "f" (функція) в результаті.

IRI paramIRI = longResult.getNamedIndividual("a").getIRI(): отримуємо IRI екземпляра, який був помічений як "a" (аргумент) в результаті.

3. Створення об'єктів *OWLNamedIndividual*:

dataFactory.getOWLNamedIndividual(funcIRI): створюємо об'єкт *OWLNamedIndividual* для екземпляра функції на основі отриманого IRI.

dataFactory.getOWLNamedIndividual(paramIRI): Створюємо об'єкт *OWLNamedIndividual* для екземпляра аргументу на основі отриманого IRI.

4. Операції зі зберіганням даних:

Створюємо тимчасовий список *temp*, який міститиме екземпляри аргументів для кожної функції:

- перевіряємо, чи вже є збережений список аргументів для даної функції. Якщо так, додаємо новий аргумент до списку.
- додаємо оновлений список аргументів до *args*, де ключем є IRI функції.

- видаляємо аксіоми між функцією та її аргументом з онтології за допомогою *manager.removeAxioms(ontology,getAxiomsBetweenIndividuals(funcInd, paramInd)).*

Усі ці дії виконуються для кожного результату запиту SQWRL, що дозволяє обробляти та оновлювати дані в онтології на підставі результатів запиту.

Наведені у коді вище дії дозволяють знайти екземпляри в базі знань, які є результатом виконання запиту, а також видалити усі аксіоми, які їх пов'язують. У нашому випадку – це аксіоми позначення зв'язку *hasArgument*. Таким чином розриваємо зв'язок між екземплярами, залишаючи самі екземпляри (як функції, так і їх параметри).

Наступний етап – створення екземпляру структури, яка буде фігурувати як новий параметр для функції. Вона повинна містити властивості, які будуть відображати попередні параметри функції. Оскільки самі екземпляри параметрів (у вигляді екземплярів класу Property) в базі знань залишились, то необхідно тільки додати нову екземпляр класу Struct, а також пов'язати їх між собою властивістю *hasProperty*.

Оскільки раніше було додано властивість *argumentCount*, що відображає кількість аргументів у функції, її необхідно оновити. Для цього видаляємо попередню аксіому і створюємо нову.

Наведений нижче фрагмент коду (Рисунок 58) обробляє список аргументів для функцій та оновлює відповідні аксіоми в онтології.

```
for (Map.Entry<IRI, List<OWLNamedIndividual>> entry : args.entrySet()) {
    Struct newStruct = new Struct(name:"ArgumentList");
    OWLNamedIndividual structInd = addStruct(newStruct, entry.getValue());
    OWLNamedIndividual funcInd = dataFactory.getOWLNamedIndividual(entry.getKey());
    manager.addAxiom(ontology, dataFactory.getOWLObjectPropertyAssertionAxiom(hasArgumentProperty,
                                                                              funcInd,
                                                                              structInd));
    OWLAxiom argAxiom = getArgumentCountAxiom(funcInd);
    if (argAxiom != null) {
        manager.removeAxiom(ontology, argAxiom);
    }
    manager.addAxiom(ontology, dataFactory.getOWLDataPropertyAssertionAxiom(argumentCountProperty,
                                                                            funcInd,
                                                                            dataFactory.getOWLLiteral(value:1)));
}
```

Рисунок 58. Оновлення аксіом в моделі-онтології

Проаналізуємо його крок за кроком:

1. *for (Map.Entry<IRI, List<OWLNamedIndividual>> entry : args.entrySet()) { ... }*: цикл *for* перебирає всі записи в *args*, де кожен запис містить IRI екземпляр функції як ключ та список аргументів як значення.

2. Створення нової структури *Struct*:

Struct newStruct = new Struct("ArgumentList");: створюється новий екземпляр Struct з ім'ям "ArgumentList".

3. Додавання екземпляру структури до онтології:

OWLNamedIndividual structInd = addStruct(newStruct, entry.getValue()): виклик методу addStruct, який додає структуру *newStruct* до бази знань та повертає *OWLNamedIndividual*, що представляє цю структуру.

4. Отримання *OWLNamedIndividual* функції та додавання властивостей:

dataFactory.getOWLNamedIndividual(entry.getKey()): створюється *OWLNamedIndividual* для функції на основі IRI функції з *entry.getKey()*.

manager.addAxiom(ontology, dataFactory.getOWLObjectPropertyAssertionAxiom(hasArgumentProperty, funcInd, structInd));: додається аксіома відношення між функцією і створеною структурою *hasArgumentProperty*.

5. Оновлення аксіоми для кількості аргументів:

OWLAxiom argAxiom = getArgumentCountAxiom(funcInd);: викликається метод *getArgumentCountAxiom*, який отримує аксіому для кількості аргументів функції.

if (argAxiom != null) { ... }: перевіряється, чи знайдено аксіому. Якщо так, вона видаляється з онтології.

6. Оновлення аксіоми для кількості аргументів на значення 1:

manager.addAxiom(ontology, dataFactory.getOWLDataPropertyAssertionAxiom(argumentCountProperty, funcInd, dataFactory.getOWLLiteral(1)));: додається нова аксіома, що вказує, що кількість аргументів для функції дорівнює 1.

Усі ці операції виконуються для кожного екземпляру функції, що має аргументи. Вони відображають процес оновлення інформації про аргументи функцій у базі знань. Варто зауважити, що у попередніх двох операціях використовується визначення і видалення аксіом з онтології.

Наведений нижче фрагмент коду (Рисунок 59) реалізує метод getAxiomsBetweenIndividuals, який призначений для знаходження аксіом між двома екземплярами в онтології.

```
private Set<OWLAxiom> getAxiomsBetweenIndividuals(OWLNamedIndividual first, OWLNamedIndividual second) {
    Set<OWLAxiom> relevantAxioms = new HashSet<>();
    for (OWLAxiom axiom : ontology.getAxioms()) {
        if (axiom instanceof OWLObjectPropertyAssertionAxiom) {
            OWLObjectPropertyAssertionAxiom objectPropertyAssertion = (OWLObjectPropertyAssertionAxiom)axiom;
            OWLIndividual subject = objectPropertyAssertion.getSubject();
            OWLIndividual object = objectPropertyAssertion.getObject();
            if ((subject.equals(first) && object.equals(second)) ||
                (subject.equals(second) && object.equals(first))) {
                relevantAxioms.add(axiom);
            }
        }
    }
    return relevantAxioms;
}
```

Рисунок 59. Знаходження крос-залежності між екземплярами

Розглянемо його детальніше:

Set<OWLAxiom> getAxiomsBetweenIndividuals(OWLNamedIndividual first, OWLNamedIndividual second) { ... }: цей метод приймає два *OWLNamedIndividual* як аргументи та повертає множину OWLAxiom.

Set<OWLAxiom> relevantAxioms = new HashSet<>();: створюється порожня множина *relevantAxioms*, в яку будуть додаватися знайдені аксіоми.

for (OWLAxiom axiom : ontology.getAxioms()) { ... }: цикл for перебирає всі аксіоми в онтології.

Перевірка типу аксіоми:

if (axiom instanceof OWLObjectPropertyAssertionAxiom) { ... }: перевіряється, чи є аксіома типу *OWLObjectPropertyAssertionAxiom*, тобто аксіома відношення об'єктів.

OWLObjectPropertyAssertionAxiom objectPropertyAssertion = (OWLObjectPropertyAssertionAxiom)axiom;: якщо аксіома є відношенням об'єктів,

вона перетворюється на тип OWLObjectPropertyAssertionAxiom для подальшої обробки.

Отримання об'єкта та суб’єкта аксіоми:

OWLIndividual subject = objectPropertyAssertion.getSubject();: отримання екземпляра-суб’єкта аксіоми.

OWLIndividual object = objectPropertyAssertion.getObject();: отримання екземпляра -об'єкта аксіоми.

Перевірка відношення між двома екземплярами:

if ((subject.equals(first) && object.equals(second)) || (subject.equals(second) && object.equals(first))) { ... }: перевіряється, чи екземпляри *subject* та *object* співпадають з переданими *first* і *second* або навпаки.

Додавання аксіоми до множини аксіом:

relevantAxioms.add(axiom);: якщо відношення підходить, аксіома додається до множини *relevantAxioms*.

Повернення множини аксіом:

return relevantAxioms;: метод повертає множину аксіом, які мають відношення між переданими екземплярами.

Наведений нижче фрагмент коду (Рисунок 60) описує метод, призначений для знаходження аксіоми, що визначає кількість аргументів для заданого екземпляра в онтології.

```
private OWLAxiom getArgumentCountAxiom(OWLNamedIndividual ind) {
    for (OWLDataPropertyAssertionAxiom axiom : ontology.getDataPropertyAssertionAxioms(ind)) {
        if (axiom.getProperty().asOWLDataProperty().getIRI().equals(argumentCountProperty.getIRI())) {
            return axiom;
        }
    }
    return null;
}
```

Рисунок 60. Код для знаходження аксіоми кількості аргументів

Розглянемо його загальну структуру:

OWLAxiom getArgumentCountAxiom(OWLNamedIndividual ind) { ... }: метод приймає *OWLNamedIndividual* (екземпляр) як аргумент та повертає *OWLAxiom* (аксіому).

for (OWLDataPropertyAssertionAxiom axiom : ontology.getDataPropertyAssertionAxioms(ind)) { ... }: цикл for перебирає всі аксіоми, які вказують на властивість даних для заданого екземпляра *ind*.

Перевірка властивості даних та повернення аксіоми: *if (axiom.getProperty().asOWLDataProperty().getIRI().equals(argumentCountProperty .getIRI())) { ... }:* перевіряється, чи IRI властивості даних аксіоми відповідає IRI властивості даних для кількості аргументів (*argumentCountProperty*).

return axiom;: якщо властивість даних співпадає, повертається знайдена аксіома.

return null;: якщо у циклі не було знайдено аксіому, що відповідає властивості даних для кількості аргументів, метод повертає *null*.

Після виконання всіх дій, які зазначені вище, отримаємо наступний результат. Для попереднього початкового коду запустимо виконання системи (Рисунок 61).

```
<owl:NamedIndividual rdf:about="owl:Ontology#17972ba5-ee66-48e2-acd1-a870a8e5e074">
    <rdf:type rdf:resource="owl:Ontology#Function"/>
    <equalsTo rdf:resource="owl:Ontology#17972ba5-ee66-48e2-acd1-a870a8e5e074"/>
    <hasArgument rdf:resource="owl:Ontology#0ebee8d0-30ae-4cb5-8715-68d87f5a3022"/>
    <hasArgument rdf:resource="owl:Ontology#13563996-b3b5-4502-bdf5-628a09a5785d"/>
    <hasArgument rdf:resource="owl:Ontology#232b5339-b4cc-4fbf-8e17-00c3e83c0439"/>
    <hasArgument rdf:resource="owl:Ontology#62880953-fbcd-4963-bad3-c90a4827bfab"/>
    <hasArgument rdf:resource="owl:Ontology#9f0fef83-843f-4b0f-b152-089ed0f54b60"/>
    <isIn rdf:resource="owl:Ontology#409d7795-e0e4-4128-bee4-d65f70aa47a2"/>
    <argumentCount rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">5</argumentCount>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">say</name>
    <returnType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">String</returnType>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(19:5,21:6)</sourceCodeReference>
</owl:NamedIndividual>
```

Рисунок 61. Опис екземпляру функції до застосування рефакторингу

Як бачимо, замість п’ятьох аргументів, функція має лише один, що відображено у властивостях *hasArgument* та *argumentCount* (Рисунок 62).

```
<owl:NamedIndividual rdf:about="owl:Ontology#ddd376c2-7f2e-40d7-afec-07be4e3a51d5">
    <rdf:type rdf:resource="owl:Ontology#Function"/>
    <hasArgument rdf:resource="owl:Ontology#c47dbcc5-3a94-4fd8-abd0-815937c20c11"/>
    <isIn rdf:resource="owl:Ontology#9f9034b0-1800-4eb3-84c7-e7c21257b3b9"/>
    <argumentCount rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">1</argumentCount>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">say</name>
    <returnType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">String</returnType>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(19:5,21:6)</sourceCodeReference>
</owl:NamedIndividual>
```

Рисунок 62. Опис екземпляру функції після застосування рефакторингу

На Рисунок 63 показано створену структуру, яка об’єднує в собі всі властивості із функції.

```
<owl:NamedIndividual rdf:about="owl:Ontology#c47dbcc5-3a94-4fd8-abd0-815937c20c11">
    <rdf:type rdf:resource="owl:Ontology#Struct"/>
    <hasProperty rdf:resource="owl:Ontology#ad641473-8c27-4d86-9f85-3db7a56d4351"/>
    <hasProperty rdf:resource="owl:Ontology#24d9f6a1-25fb-48cf-9baf-abd029025db3"/>
    <hasProperty rdf:resource="owl:Ontology#6b11cd18-e444-4e6c-97b1-f814505cdcbd"/>
    <hasProperty rdf:resource="owl:Ontology#666325f4-d134-4ebb-ae44-7730fa4e3c9a"/>
    <hasProperty rdf:resource="owl:Ontology#96a75860-f4fe-4335-a79a-d7e2d9f7eb99"/>
    <isInFunction rdf:resource="owl:Ontology#ddd376c2-7f2e-40d7-afec-07be4e3a51d5"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">ArgumentList</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string"></sourceCodeReference>
</owl:NamedIndividual>
```

Рисунок 63. Екземпляр структури, створеної в результаті рефакторингу

Таким чином лише трьома кроками (визначення екземплярів, видалення аксіом, додавання аксіом) можна здійснити рефакторинг усіх методів, які підпадають під визначені умови.

Здійснимо застосування усіх запропонованих вище концепцій на одному конкретному прикладі. На Рисунок 64 зображено початковий код, у якому можна здійснити рефакторинг.

```
protocol Say {
    func say() -> String
}

class Animal {
    let age: Int

    init() {
        age = 0
    }

    func say() -> String{
        return "Hello"
    }

    func go(from: String,
            to: String,
            with: String,
            speed: Int,
            distance: Int) -> String{
        let time: Double = distance / speed
        return "Will arrive from \(from) to \(to) in \(time)"
    }
}
```

Рисунок 64. Початковий код, для якого потрібно здійснити рефакторинг

Клас Animal реалізує протокол Say, проте в коді це не зазначено явно. Функція go має 5 аргументів, а відповідно до загальноприйнятих практик розробки ПЗ, таку кількість параметрів необхідно замінити структурою даних.

На Рисунок 65, Рисунок 66 зображено представлення цього коду (частково) у базі знань.

```
<owl:NamedIndividual rdf:about="http://rdf.webofcode.org/woc/b0d98297-38e6-4dd1-948d-350f8d896a35">
    <rdf:type rdf:resource="http://rdf.webofcode.org/woc/ClassDeclarationStatement"/>
    <hasMethod rdf:resource="http://rdf.webofcode.org/woc/b304b1a2-6186-44eb-9bb8-0fa4fb8766da"/>
    <hasMethod rdf:resource="http://rdf.webofcode.org/woc/0985d16d-4f85-4e9b-8202-01a3e484bf26"/>
    <hasProperty rdf:resource="http://rdf.webofcode.org/woc/437649f2-bdf6-483c-a52d-1cdbb3c925c8"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Animal</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(12:1,31:2)</sourceCodeReference>
</owl:NamedIndividual>
```

Рисунок 65. Представлення класу в базі знань до рефакторингу

Як бачимо, екземпляр класу посилається на екземпляр методу.

```
<owl:NamedIndividual rdf:about="http://rdf.webofcode.org/woc/b304b1a2-6186-44eb-9bb8-0fa4fb8766da">
    <rdf:type rdf:resource="http://rdf.webofcode.org/woc/Method"/>
    <hasArgument rdf:resource="http://rdf.webofcode.org/woc/d80105b2-3f76-400a-97cf-dad64b935d18"/>
    <hasArgument rdf:resource="http://rdf.webofcode.org/woc/eeb801ea-e681-461e-a582-d048ba2fc926"/>
    <hasArgument rdf:resource="http://rdf.webofcode.org/woc/5352985d-5301-431e-bbc6-886792dc2bb8"/>
    <hasArgument rdf:resource="http://rdf.webofcode.org/woc/69b2312f-46c6-431a-8360-656b07f10941"/>
    <hasArgument rdf:resource="http://rdf.webofcode.org/woc/8a8aab96-539f-4947-95a5-849d01172ea6"/>
    <argumentCount rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">5</argumentCount>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">go</name>
    <returnType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">String</returnType>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(23:5,30:6)</sourceCodeReference>
</owl:NamedIndividual>
```

Рисунок 66. Представлення методу go в базі знань до рефакторингу

На Рисунок 67 частково зображено правила, які будуть застосовуватись до представлення початкового коду.

```
<rdf:Description>
    <swrla:isRuleEnabled rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">true</swrla:isRuleEnabled>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"></rdfs:comment>
    <rdfs:label rdf:datatype="http://www.w3.org/2001/XMLSchema#string">sameProperty</rdfs:label>
    <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#Imp"/>
    <swrl:body>
        <rdf:Description>
            <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#AtomList"/>
            <rdf:first>
                <rdf:Description>
                    <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#ClassAtom"/>
                    <swrl:classPredicate rdf:resource="http://rdf.webofcode.org/woc/Property"/>
                    <swrl:argument1 rdf:resource="http://rdf.webofcode.org/woc/p1"/>
                </rdf:Description>
            </rdf:first>
            <rdf:rest>
                <rdf:Description>
                    <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#AtomList"/>
                    <rdf:first>
                        <rdf:Description>
                            <rdf:type rdf:resource="http://www.w3.org/2003/11/swrl#DatavaluedPropertyAtom"/>
                            <swrl:propertyPredicate rdf:resource="http://rdf.webofcode.org/woc/name"/>
                            <swrl:argument1 rdf:resource="http://rdf.webofcode.org/woc/p1"/>
                            <swrl:argument2 rdf:resource="http://rdf.webofcode.org/woc/p1_name"/>
                        </rdf:Description>
                    </rdf:first>
```

Рисунок 67. Представлення правил (частково) у базі знань

Після застосування процедури логічного виведення сутності коду матимуть вигляд у базі знань такий, як показано на Рисунок 68, Рисунок 69, Рисунок 70.

```
<owl:NamedIndividual rdf:about="http://rdf.webofcode.org/woc/ae8a8adf-1fd5-4e3b-a30b-bc6814de5488">
    <rdf:type rdf:resource="http://rdf.webofcode.org/woc/ClassDeclarationStatement"/>
    <xkos:hasPart rdf:resource="http://rdf.webofcode.org/woc/6ac8f71b-7bdf-4dd8-8693-6920f750c973"/>
    <xkos:hasPart rdf:resource="http://rdf.webofcode.org/woc/7749a85b-2a93-4b5d-9b43-05926724c0bd"/>
    <conformsTo rdf:resource="http://rdf.webofcode.org/woc/e2aa8eca-bfa2-4eaf-addf-1abc73d67526"/>
    <declares rdf:resource="http://rdf.webofcode.org/woc/6ac8f71b-7bdf-4dd8-8693-6920f750c973"/>
    <declares rdf:resource="http://rdf.webofcode.org/woc/7749a85b-2a93-4b5d-9b43-05926724c0bd"/>
    <hasMethod rdf:resource="http://rdf.webofcode.org/woc/6ac8f71b-7bdf-4dd8-8693-6920f750c973"/>
    <hasMethod rdf:resource="http://rdf.webofcode.org/woc/7749a85b-2a93-4b5d-9b43-05926724c0bd"/>
    <hasProperty rdf:resource="http://rdf.webofcode.org/woc/bb369869-2b58-4fa3-9c96-425e52d1828a"/>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Animal</name>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(12:1,31:2)</sourceCodeReference>
</owl:NamedIndividual>
```

Рисунок 68. Представлення класу після рефакторингу

Структура містить такі властивості, що відповідають попереднім аргументам методу.

```
    <owl:NamedIndividual rdf:about="http://rdf.webofcode.org/woc/0451efb4-7370-47f2-8093-4157b8d99db1">
        <rdf:type rdf:resource="http://rdf.webofcode.org/woc/Struct"/>
        <hasProperty rdf:resource="http://rdf.webofcode.org/woc/dd38cd99-2805-4a0e-9ab5-5034525aeae6"/>
        <hasProperty rdf:resource="http://rdf.webofcode.org/woc/463ad42a-b2c6-4590-abc4-044a8326120f"/>
        <hasProperty rdf:resource="http://rdf.webofcode.org/woc/77e3e426-c5d1-4b3c-a9a7-4eea0a093611"/>
        <hasProperty rdf:resource="http://rdf.webofcode.org/woc/85029f56-f1f3-4fb4-a287-977b1c631762"/>
        <hasProperty rdf:resource="http://rdf.webofcode.org/woc/99b0057f-d19e-4558-9e4c-13edabd578b9"/>
        <isInFunction rdf:resource="http://rdf.webofcode.org/woc/447dd601-88dc-4f77-8e36-ac27027e28ce"/>
        <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">ArgumentList</name>
        <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string"></sourceCodeReference>
    </owl:NamedIndividual>
```

Рисунок 69. Представлення створеної структури з параметрами

Як бачимо з формалізованого опису, метод містить тепер лише один аргумент.

```
<owl:NamedIndividual rdf:about="http://rdf.webofcode.org/woc/447dd601-88dc-4f77-8e36-ac27027e28ce">
    <rdf:type rdf:resource="http://rdf.webofcode.org/woc/Method"/>
    <xkos:isPartOf rdf:resource="http://rdf.webofcode.org/woc/a07c97b4-0263-4645-9e6c-856a178f5a82"/>
    <hasArgument rdf:resource="http://rdf.webofcode.org/woc/0451efb4-7370-47f2-8093-4157b8d99db1"/>
    <isDeclaredBy rdf:resource="http://rdf.webofcode.org/woc/a07c97b4-0263-4645-9e6c-856a178f5a82"/>
    <isIn rdf:resource="http://rdf.webofcode.org/woc/a07c97b4-0263-4645-9e6c-856a178f5a82"/>
    <isMethodOf rdf:resource="http://rdf.webofcode.org/woc/a07c97b4-0263-4645-9e6c-856a178f5a82"/>
    <argumentCount rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">1</argumentCount>
    <name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">go</name>
    <returnType rdf:datatype="http://www.w3.org/2001/XMLSchema#string">String</returnType>
    <sourceCodeReference rdf:datatype="http://www.w3.org/2001/XMLSchema#string">(23:5,30:6)</sourceCodeReference>
</owl:NamedIndividual>
```

Рисунок 70. Представлення методу go після рефакторингу

На Рисунок 71 зображено початковий код, у якому здійснено рефакторинг. Клас Animal реалізує протокол Say і це зазначено явно. Функція go має 1 аргумент, а всі попередні параметри винесено в окрему структуру ArgumentList.

Можемо зробити висновок, що застосування такого підходу є ефективним і для виконання інших дій рефакторингу – відсутня необхідність розумітись на

семантиці конкретної мови програмування, а потрібно описати лише власне логіку рефакторингу покроково. Це зробити нескладно, оскільки база знань містить ті ж поняття і сутності, що й шаблони рефакторингу.

Оскільки база знань, що створена із початкового коду, описана із використанням стандарту OWL, то її можна завантажити до будь-якої системи, що працює із відповідним форматом і здійснювати подальше опрацювання.

```
protocol Say {
    func say() -> String
}

struct ArgumentList {
    var from: String
    var to: String
    var with: String
    var speed: Int
    var distance: Int
}

class Animal: Say {
    let age: Int

    init() {
        age = 0
    }

    func say() -> String{
        return "Hello"
    }

    func go(params: ArgumentList) -> String{
        let time: Double = params.distance / params.speed
        return "Will arrive from \(params.from) to \(params.to) in \(params.time)"
    }
}
```

Рисунок 71. Початковий код після рефакторингу

На Рисунок 72, Рисунок 73, Рисунок 74, Рисунок 75 частково показано базу знань.

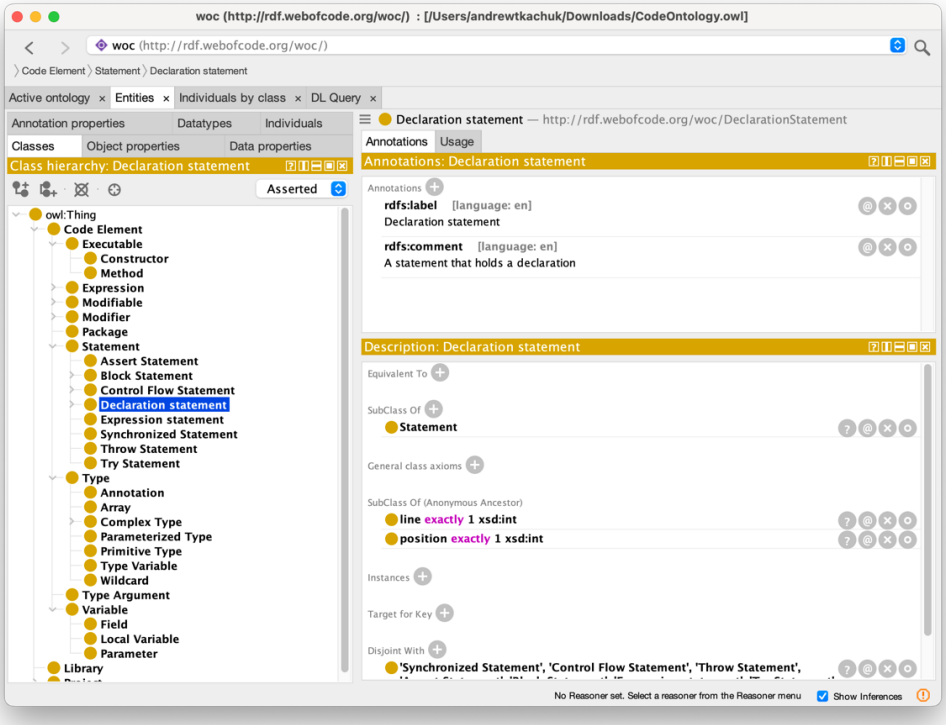

Рисунок 72. Ієрархія класів в онтології

Базу знань було вивантажено у форматі OWL, що дозволяє переглядати її засобами Protégé.

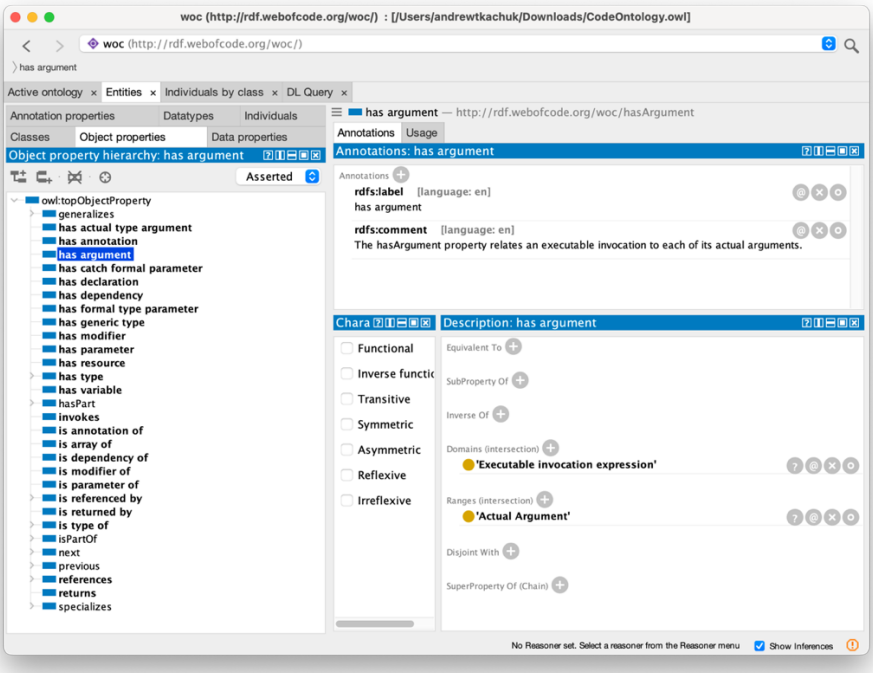

Рисунок 73. Об’єктні властивості

Рисунки демонструють класи, властивості та присутні екземпляри.

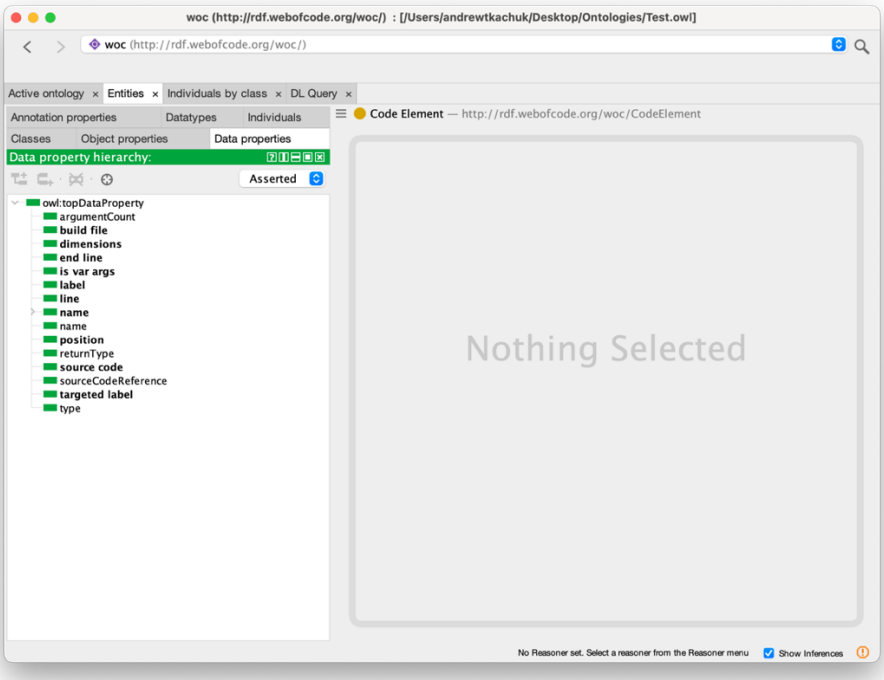

Рисунок 74. Властивості даних

Така база містить у собі усі ABox та TBox аксіоми (отримані з онтології та створені шляхом синтаксичного аналізу коду), що власне й показано на рисунках.

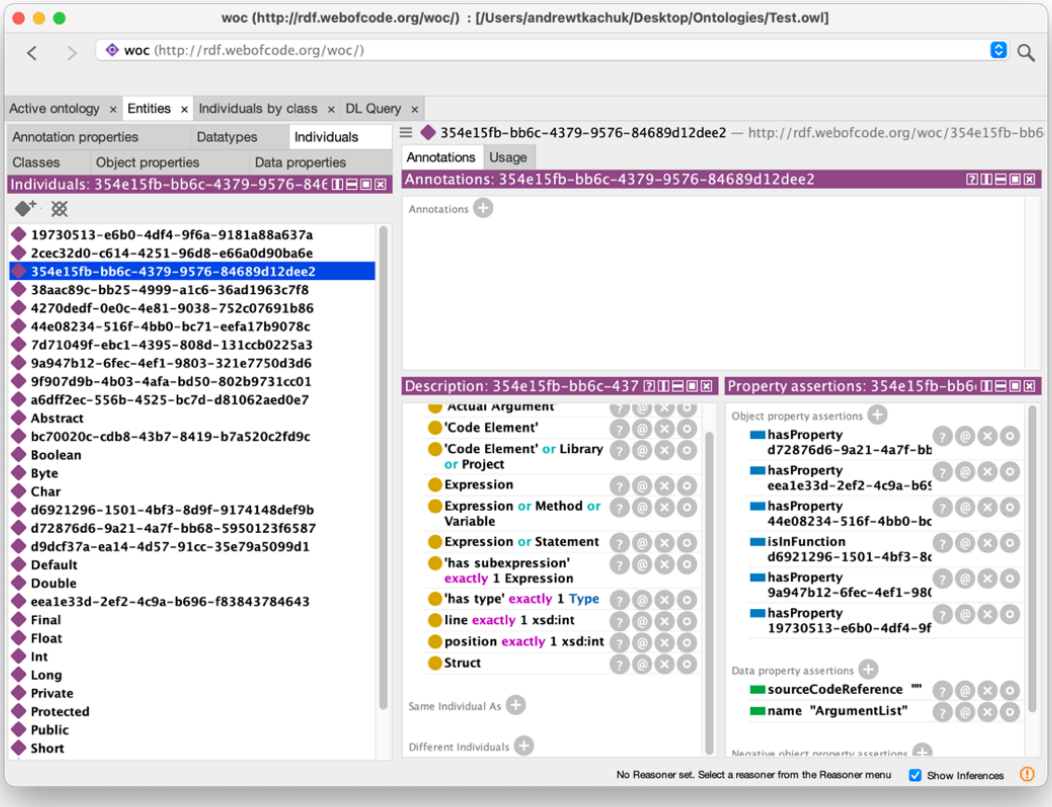

Рисунок 75. Список екземплярів

Як бачимо на рисунку, екземпляри, що створені в результаті синтаксичного аналізу, мають назву у формі UUID.

ВИСНОВКИ ДО РОЗДІЛУ 6

У цьому розділі було розглянуто одне з основних обмежень системи логічного виводу стосовно продукції знань в базі знань, побудованій на онтології, - неможливості здійснювати обчислення функціональних залежностей, які в тій чи іншій мірі важливі для опису дій рефакторингу і його автоматизованого виконання.

Для розширення можливостей класичних систем логічного виводу було запропоновано використовувати правила SWRL, які використовують поняття бази знань TBox і дозволяють продукувати нові знання, що базуються на функціональних залежностях.

Було формалізовано метод використання правил для аналізу і рефакторингу початкового коду і продемонстровано можливість його інтеграції із загальний методом аналізу і рефакторингу, що використовує базу знань. Для порівняння було наведено схематичне зображення процесу рефакторингу, а також блок-схему і вказано місце запропонованого методу у них.

Було запропоновано комплексну модель для представлення знань про початковий код, яка складається з двох компонентів.

Перший компонент – базується на онтологіях і є незмінним. Було зазначено, що важливим для роботи методу є опис властивостей тих чи інших сутностей моделі з точки зору аксіом еквівалентності, ієрархічності для можливості здійснення автоматизованого рефакторингу на основі класифікації чи встановлення відношень.

Другий компонент складається із набору ABox аксіом, які створюються на основі конкретного початкового коду в результаті синтаксичного аналізу цього коду.

Було здійснено огляд основних властивостей і можливостей S(Q)WRL, підходів до написання правил і запитів.

Для практичної перевірки корисності пропонованих правил було проведено ряд випробувань, які мали на меті додати до бази знань правила

здійснення кількох дій рефакторингу, а саме – позначити подібні сутності (варто зауважити, що подібність у початковому коді не є еквівалентністю екземплярів в термінах онтології чи бази знань) і на основі цієї подібності додати знання в модель про те, що клас має певний інтерфейс (реалізує протокол), а також здійснити автоматизоване покращення коду шляхом винесення параметрів методу в окрему структуру. Було зроблено висновок, що пропоновані правила є зручними і гнучкими, що дозволяє успішно виконати поставлене завдання.

РОЗДІЛ 7. ОЦІНКА ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ ЗАПРОПОНОВАНИХ МЕТОДІВ

Оцінка ефективності запропонованого методу не може відбуватись суто за кількісними характеристиками, оскільки поняття «якості рефакторингу» не визначене однозначно. Якість коду є комплексною оцінкою і визначається відсотком покриття тестами, кількістю антишаблонів і дефектів у коді, іншими показниками. Замір якості коду до виконання рефакторингу і після виконання рефакторингу буде оцінювати якість власне конкретного алгоритму рефакторингу, а не методу здійснення цього рефакторингу (за умови, що системи, що порівнюються, мають однаковий набір можливостей і функцій). Для оцінки ефективності було використано такі критерії:

- якісна оцінка наявності функціональних можливостей експертами;
- кількісна оцінка часу, затраченого розробниками, для виконання дії рефакторингу;
- кількісна оцінка витрачених часу й оперативної пам'яті для виконання поставлених завдань.

Для того, аби перевірити гіпотезу про пришвидшення роботи розробників при здійсненні рефакторингу, а також при взаємодії із власне системою рефакторингу було проведено ряд випробувань.

7.1 Оцінка конкурентних переваг методу

Трьом фокус-групам було запропоновано оцінити можливості і властивості створеного прототипу програмного продукту, що використовує запропонований метод рефакторингу, на основі кількох критеріїв: *кількість правил аналізу і рефакторингу*, *здатність до розширення*, *тип сховища правил*, *гарантія успішного виконання коду.*

Для врівноваження умов оцінки, учасникам було запропоновано оцінити будь-який існуючий засіб (програму) для рефакторингу, що базується на *основі*

правил, *на основі регулярних виразів*, *або на застосуванні засобів ШІ,* і порівняти можливості із запропонованим прототипом.

Отримані результати зведені у Таблиця 4.

Таблиця 4. Якісне порівняння методів

Існуючі методи	**Новий метод**
Різний набір можливостей для різних мов програмування (SonarQube, Checkmarx, Snyk)	Однаковий набір можливостей через введення рівня абстракції
Необхідність визначати дії рефакторингу для кожної мови	Потрібно описати дії один раз в онтології
Відсутність централізованого місця збереження автоматизованих патернів	Початкова онтологія містить опис відомих патернів, які будуть застосовані до знань про початковий код, що додаються
Відсутність гарантії успішної компіляції/інтерпретації коду після застосування рефакторингу (нейромережеві методи)	Успішна компіляція/інтерпретація коду через дотримання синтаксису та семантики завдяки використанню описових логік та правил на їх основі

Серед конкурентних переваг запропонованого методу аналізу і рефакторингу початкового коду загалом у порівнянні із існуючими було зазначено, що новий метод містить однаковий набір можливостей для усіх мов програмування, оскільки реалізації правил і можливостей відбувається на стороні бази знань, що не залежить від конкретної технології чи мови програмування. Важливо також зазначити, що усі правила, що додаються в базу знань, є агностичними відносно мови програмування, тому додавати їх необхідно тільки один раз.

Запропонований метод надає можливість одразу описати шаблони проектування чи дії, які повинні бути виконані системою логічного виводу онтології, що дозволяє застосовувати методи рефакторингу одразу після додавання початкового коду в базу знань і без втручання розробника.

Вищеописані переваги також є стійкими перетвореннями початкового коду, що гарантуватиме успішне компілювання чи інтерпретацію модифікованого за допомогою запропонованої програмної компоненти початкового коду.

7.2 Оцінка витрат часу на виконання дій рефакторингу

Для того, щоб оцінити скорочення часу, який розробники витрачають на рефакторинг, було проведено дослідження затрат часу із трьома фокус-групами.

Розробникам було запропоновано оцінити час, який вони витратили на виконання чотирьох рефакторингів (*перейменування змінної, винесення методу, заміна параметрів, заміна усіх методів із типом повернення*) двома способами: за допомогою звичного для них засобу рефакторингу і з використанням запропонованого програмного продукту.

На графіках зображено час (у хвилинах), який розробники витрачали для виконання певної дії рефакторингу. З усіх графіків можна зробити висновок, що необхідно витрачати меншу кількість часу на виконання рефакторингу із використанням запропонованого продукту, а також різниця затраченого часу у всіх фокус-групах зменшується за наявності автоматизованого засобу рефакторингу, що свідчить про те, що засіб працює ефективно як для розробників із досвідом, так і для новачків. Для перевірки цього було обраховано середню абсолютну дисперсію для кожного з випадків за формулою:

$$MAD = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 \qquad (5)$$

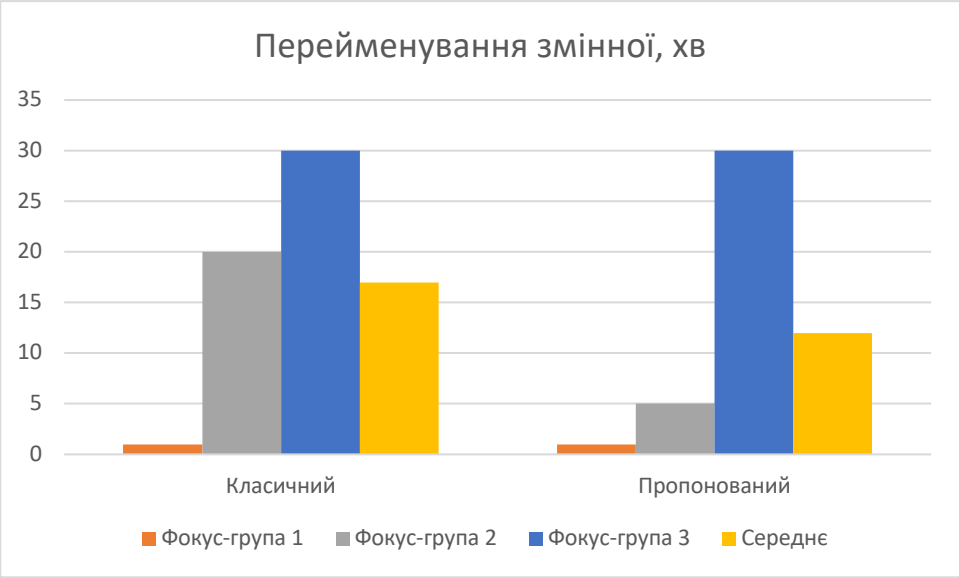

Рисунок 76. Порівняння часу, витраченого на перейменування змінної

$$MAD1 = 10.67; MAD2 = 12$$

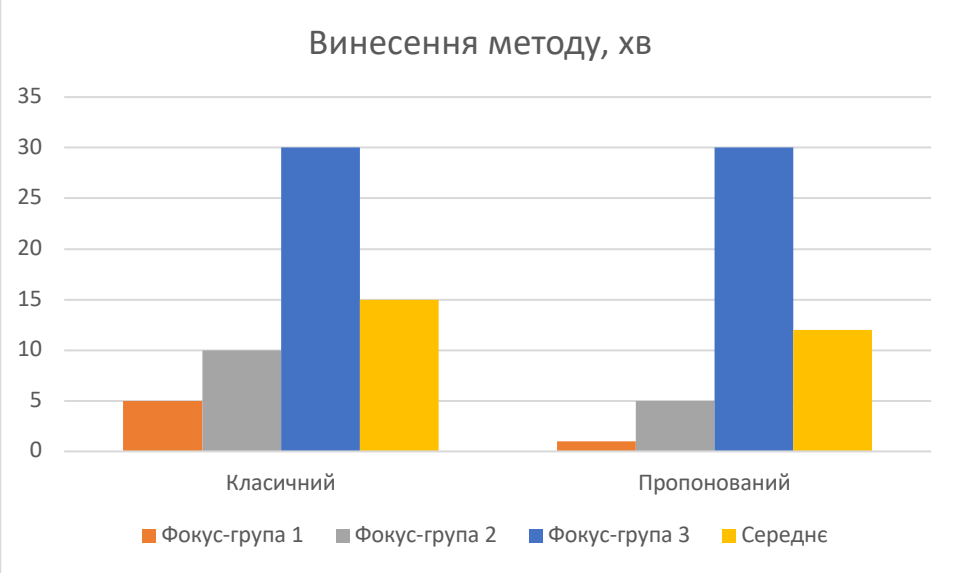

Рисунок 77. Порівняння часу, витраченого на винесення методу

$$MAD1 = 10; MAD2 = 12$$

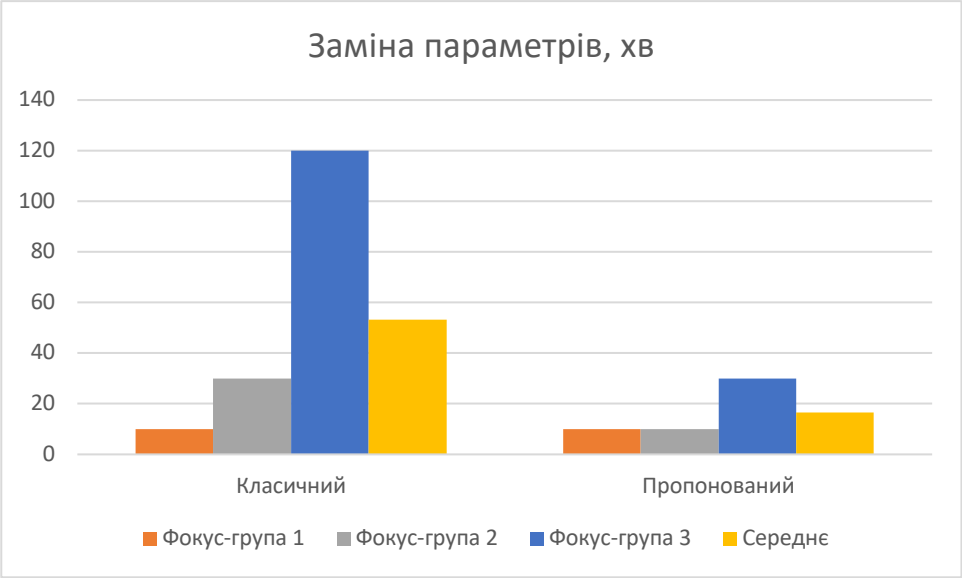

Рисунок 78. Порівняння часу, витраченого на заміну параметрів

$$MAD1 = 44.45; MAD2 = 8.89$$

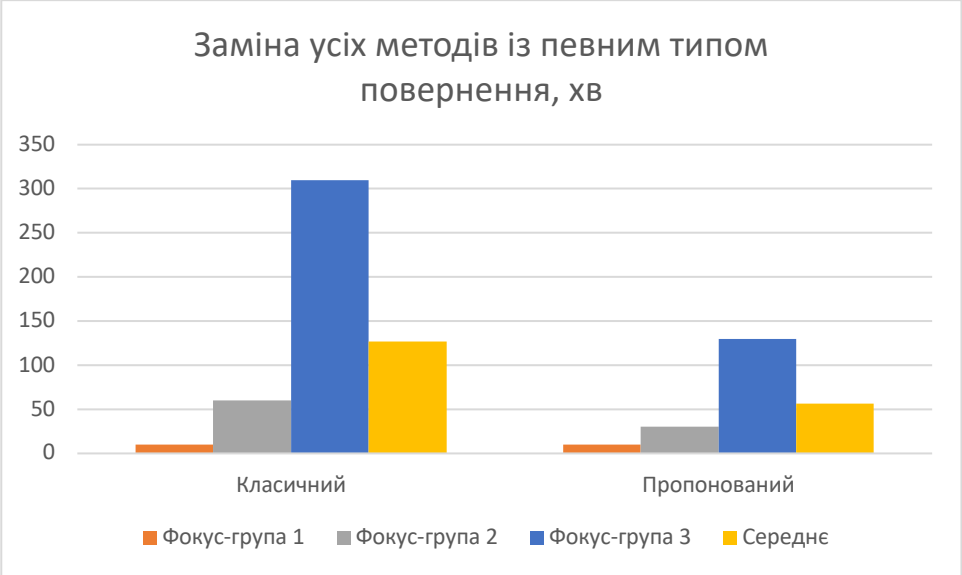

Рисунок 79. Порівняння часу, витраченого на винесення усіх зазначених методів

$$MAD1 = 197.78; MAD2 = 44.45$$

Середні значення затраченого часу для кожного із випадків зображено на графіку на Рисунок 80.

Рисунок 80. Середні значення затраченого часу для кожного із випадків рефакторингу

Проведене дослідження показало, що у найкращому випадку пришвидшення складає 36%.

Обчислені середні абсолютні дисперсії для двох способів здійснення рефакторингу показали, що різниця в часі під час застосування класичного рефакторингу у різних фокус-груп сильно варіюється (MAD~122.23 хв), а при здійсненні рефакторингу із використанням пропонованого засобу затрачений час відрізняється менше (MAD~48.89 хв), що можна пояснити уніфікуванням і спрощенням процесу.

7.3 Оцінка використання системних ресурсів

Для аналізу використання запропонованим методом системних ресурсів було проведено ряд досліджень, спрямованих на встановлення залежностей витрат часу і витрат оперативної пам'яті від кількості одиничних завдань рефакторингу, заданих до виконання однією командою, а також від кількості рядків коду.

У ході першого експерименту було виміряно час, який необхідний методу для виконання одиничних завдань рефакторингу за один прохід. Було здійснено заміри часу для завдань розміром від 100 завдань до 500 завдань із кроком 100. Результати зведено на графіку на Рисунок 81.

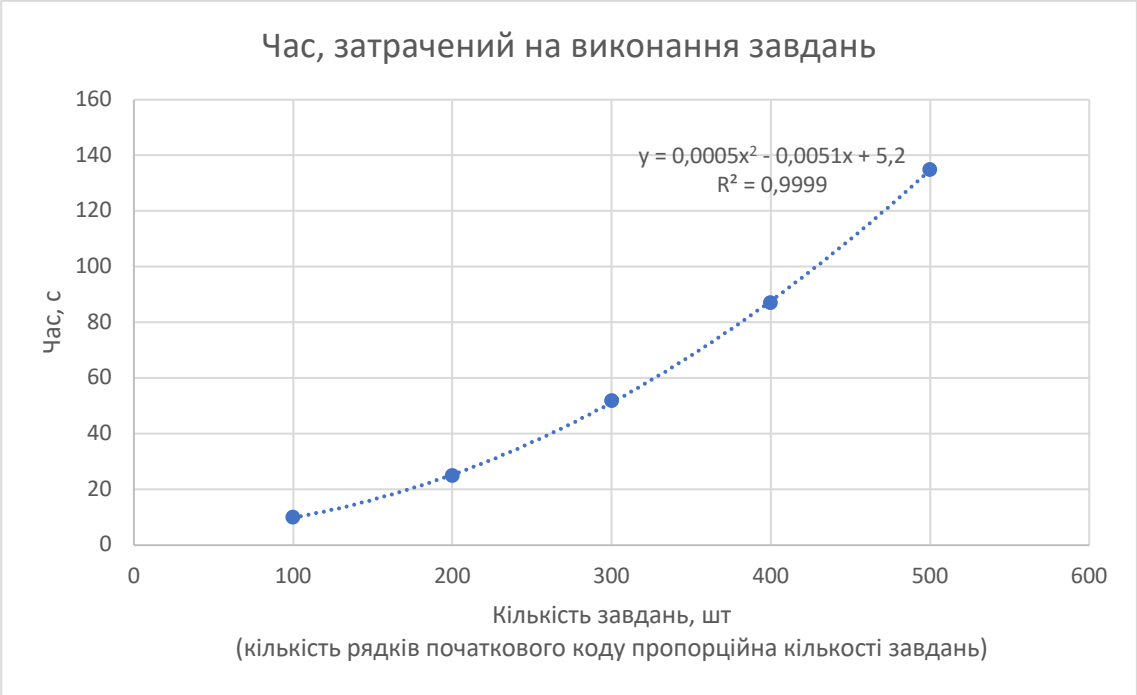

Рисунок 81. Графік залежності витрат часу від кількості завдань рефакторингу

Далі було здійснено заміри часу для одного завдання рефакторингу, який необхідно було виконати у коді різного розміру. Було здійснено заміри часу для початкового коду розміром від 1 тисячі рядків коду до 20 тисяч рядків коду. Результати зведено на графіку на Рисунок 82.

Проведене дослідження демонструє, що залежність часу описується поліномом другого степеня.

У ході другого експерименту було виміряно об’єм пам’яті, який необхідний методу для виконання тих же ж завдань, що й в першому експерименті.

Рисунок 82. Графік залежності витрат часу від розміру початкового коду

Графік залежності витрат пам’яті від кількості одиничних завдань рефакторингу зображено на Рисунок 83.

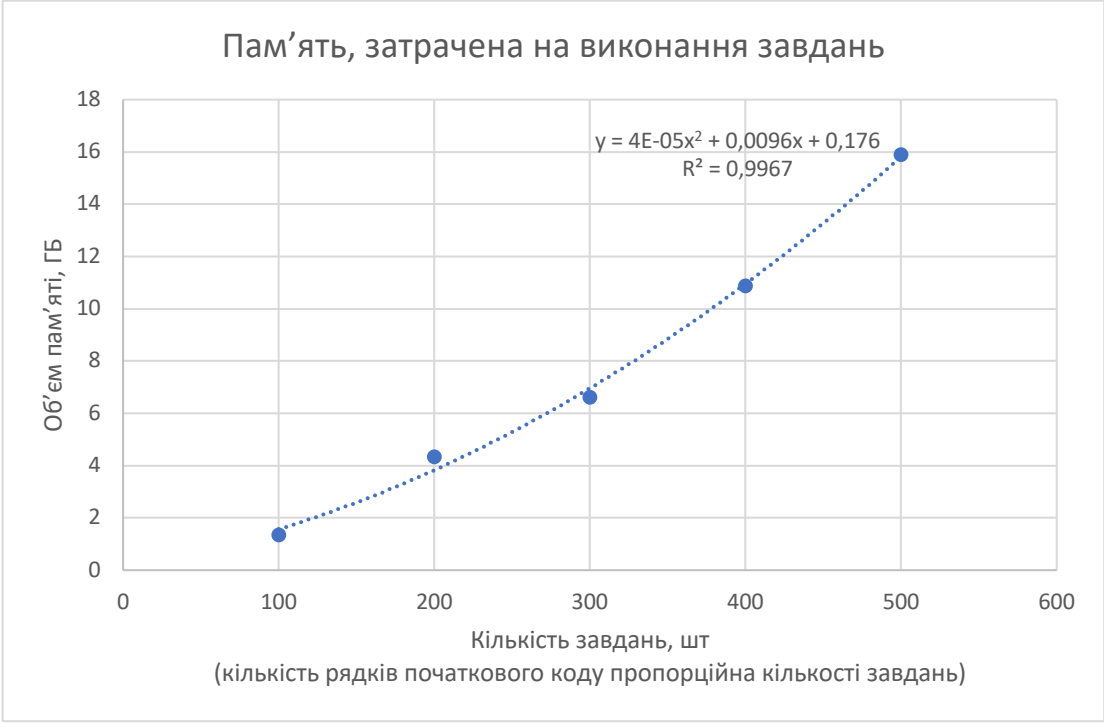

Рисунок 83. Графік залежності об’єму оперативної пам’яті від кількості завдань рефакторингу

Графік залежності витрат пам’яті від об’єму початкового коду зображено на Рисунок 84.

Рисунок 84. Графік залежності об’єму оперативної пам’яті від розміру початкового коду

Проведене дослідження демонструє, що витрата оперативної пам'яті від кількості одиничних рефакторингів описується поліномом другого степеня, а залежність витрат оперативної пам'яті від кількості рядків коду описується поліномом третього степеня.

Варто зазначити, що точна теоретична оцінка витрат пам'яті та процесорного часу не є можливою через надмірну кількість факторів, що впливають на результат. Але емпіричні заміри показали, що пропонований метод може бути застосований і для реальних великих проектів за адекватний час (через асимптотику $O(L^2)$), однак при цьому матиме суттєві вимоги по оперативній пам’яті (що втім є реальними для виконання, враховуючи що більшість існуючих систем аналізу коду також мають високі вимоги до програмних ресурсів).

ВИСНОВКИ ДО РОЗДІЛУ 7

У цьому розділі було проведено оцінку ефективності використання запропонованих методів та моделі.

Було зазначено, що ефективність запропонованого методу рефакторингу не може бути повністю визначена за числовими параметрами через те, що поняття "якості рефакторингу" є неоднозначним і не визначене чітко. Оцінка якості коду включає в себе різні аспекти, такі як відсоток покриття тестами, кількість антишаблонів і дефектів у коді, а також інші показники. Вимірювання якості коду до та після рефакторингу дозволяє оцінити ефективність конкретного алгоритму рефакторингу, а не лише методу.

Було запропоновано здійснювати якісну оцінку наявності функціональних можливостей, кількісну оцінку часу, затраченого на виконання визначених дій рефакторингу, а також кількісну оцінку витрачених часу і пам’яті.

Визначено, що серед конкурентних переваг запропонованого методу аналізу і рефакторингу і програмного продукту загалом у порівнянні із існуючими було зазначено, що новий метод містить однаковий набір можливостей для усіх мов програмування, оскільки реалізації правил і можливостей відбувається на стороні бази знань, що не залежить від конкретної технології чи мови програмування.

Оцінка затрат часу показує, що необхідно витрачати меншу кількість часу на виконання рефакторингу із використанням запропонованого продукту, а також що різниця затраченого часу у всіх фокус-групах зменшується за наявності автоматизованого засобу рефакторингу, що свідчить про те, що засіб працює ефективно як для розробників із досвідом, так і для новачків. Проведене дослідження показало, що у найкращому випадку пришвидшення складає 36%.

Проведені тестування методу на завданнях різного розміру і складності показали, що залежність часу роботи методу від розміру і складності вхідних даних описується поліномом другого степеня, так як і залежність витрати оперативної пам’яті від кількості завдань. Проте крива залежності витрати

оперативної пам’яті від кількості рядків початкового коду описується поліномом третього степеня.

Точна теоретична оцінка витрат пам’яті та процесорного часу непрактична через велику кількість факторів, які впливають на результат. Проте, емпіричні виміри підтверджують, що запропонований метод може успішно застосовуватися для обробки реальних великих проектів, забезпечуючи при цьому прийнятний час виконання (з асимптотикою $O(L^2)$). Важливо відзначити, що цей метод може потребувати значних обсягів оперативної пам’яті, але це виправдано враховуючи високі вимоги більшості існуючих систем аналізу коду до ресурсів обчислювальної системи.

ВИСНОВКИ

Дослідження в дисертаційній роботі було зосереджено на задачі покращення процесів аналізу та рефакторингу початкового коду прикладних програм із застосуванням формалізованих знань про початковий код. Його основними результатами є:

1. виконано аналіз методів аналізу і рефакторингу початкового коду, що існують на ринку, зокрема такі, що залучають формалізовані знання про код. Зазначено, що інтеграція семантичного аналізу в процеси аналізу коду та рефакторингу сприяє глибокому та точному розумінню поведінки коду, ведучи до більш надійних результатів рефакторингу. Обґрунтовано місце використання семантичного підходу в процесі рефакторингу і його доцільність;
2. проаналізовано можливості засобів і способів використання семантичної інформації для побудови формалізованого представлення знань про початковий код. Доведено, що онтологічні структури з урахуванням їх виразності є ефективним інструментом для формалізації та репрезентації доменних знань і семантики. Зокрема, в контексті відображення знань, пов'язаних із початковим кодом, застосування онтологій є цілком придатним та перспективним підходом. Обрано набір засобів OWL для подальших досліджень і побудови моделі представлення початкового коду;
3. досліджено можливості додавання та параметризації нових дій рефакторингу до інструментарію розробки мови програмування Swift. У якості експерименту додано нову дію рефакторингу і проведено її апробацію шляхом публікації на розгляд спільноті. Зроблено висновок про неможливість параметризації способу додавання і здійснення рефакторингу, що реалізується власне засобами інструментарію розробки для мови програмування Swift. Обрано Swift як основну мову для подальших досліджень, оскільки більшість популярних засобів

рефакторингу мають обмежену функціональність (або взагалі її не мають) для цієї мови програмування;

4. досліджено способи формалізації знань і їх використання як частини утиліти для здійснення рефакторингу. Доведено можливість і доцільність абстракції бази знань про початковий код від власне процесу рефакторингу задля досягнення якісно нових результатів. Зазначено необхідність розробки моделі представлення початкового коду, яка не залежатиме від тієї мови програмування, якою написаний початковий код, що підлягає рефакторингу;
5. *вперше запропоновано* метод автоматизації рефакторингу програмного коду, який *відрізняється від існуючих тим*, що здійснює перетворення коду на рівні сутностей бази знань, створеної на основі запропонованої моделі формалізації знань про синтаксис і семантику коду, для автоматичного виявлення і виправлення розповсюджених антишаблонів програмування. Зазначено отриману можливість абстрагуватись від конкретної мови програмування в проекті і здійснювати складні маніпуляції із початковим кодом шляхом маніпуляції його представленням у базі знань. Запропоновано використання системи логічного виведення для здійснення модифікацій;
6. *вперше запропоновано* метод модифікації проміжного представлення програмного коду в процесі рефакторингу, що уможливлює виправлення нетривіальних антишаблонів, який *відрізняється від існуючих* тим, що застосовує механізми логічного виведення до знань про код, сформульованих з використанням описових логік і правил логічного виведення. Запропоновано виконувати правила автоматично одразу після запуску логічного виведення в базі знань. Сформульовано підходи до написання правил для бази знань, що дозволить ефективно розширювати список підтримуваних «патернів» чи «антипатернів», які можуть бути виявлені та виправлені у початковому коді. *Удосконалено* модель

формалізації знань про програмний код для його рефакторингу на основі онтології об’єктно-орієнтованої мови програмування, що *відрізняється від існуючих тим*, що представляє не лише базові відомості про синтаксис, отримані від синтаксичного аналізатора, а й семантику складних конструкцій, шаблонів та антишаблонів з можливою прив'язкою до функцій програмного продукту. Доведено змогу виражати запити (завдання) на аналіз та рефакторинг не просто використовуючи поняття про синтаксичні конструкції в коді, але й поняття, що описують програмний продукт, його функції, будову тощо для уточнення запитів в рамках предметної області конкретного програмного продукту.

7. У ході експериментальних досліджень розроблено прототип програмного інструментарію, продемонстровано можливість відносно просто описувати довільні правила рефакторингу в онтології та здійснювати аналіз і рефакторинг різної складності, а також показано пришвидшення виконання таких завдань на 36% у порівнянні із класичними підходами до аналізу і рефакторингу. З’ясовано потреби розробленого програмного прототипу на основі запропонованих методів і моделі в системних ресурсах і зазначено, що такі потреби описуються поліноміальними залежностями від вхідних даних.

Практичне значення отриманих результатів полягає в тому, що розроблений метод аналізу та рефакторингу дозволяє повторно використовувати накопичені знання про «патерни» та «антипатерни» в базі знань при аналізі та модифікації вихідного коду різними мовами програмування. Також формалізовані знання про початковий код дозволять шукати семантичні дублікати, створювати документацію про програмний продукт, дізнаватись вміст (смислове наповнення) початкового коду. У ході експериментів було розроблено прототип системи перетворення початкового коду в базу знань, описано її архітектурні особливості. У результаті використання програмного прототипу було успішно вилучено знання про початковий код, який перевірявся, і отримані

факти занесено в базу знань, з подальшим включенням неявних аксіом, виявлених системою логічного виведення. Було запропоновано архітектуру клієнт-серверного програмного прототипу для здійснення аналізу і рефакторингу початкового коду, який дозволяє перевикористовувати менеджер бази знань на серверній частині із різними клієнтами, які реалізовуються специфічно до кожної мови програмування. Таким чином досягається можливість мати опис різних компонентів, сервісів програмного комплексу в одній онтології і опрацьовувати її як одне ціле.

Достовірність отриманих результатів забезпечується проведенням аналізу наукових праць учених в області формалізації знань, логічно та послідовно поставленими завданнями, опрацюванням значної кількості аналітичних матеріалів, наукових концепцій щодо аналізу та рефакторингу початкового коду шляхом взаємодії та виконання модифікацій з його представленням. Більше того, вона підтверджується шляхом якісного порівняння запропонованих методів і моделі з іншими системами, призначеними для виконання «нетривіального» (тобто такого, що не зводиться до простого текстового пошуку та заміни символів у коді програми) рефакторингу, а також аналізом показників ефективності витрат ресурсів при здійсненні рефакторингу з використанням запропонованих методів і залученням експертів.

Дане дослідження у майбутньому можна розвивати у кількох напрямках. По-перше, можна досліджувати використання інших способів формалізації знань про початковий код, відмінних від онтології на основі описових логік. Якщо буде доведено, що такі представлення теж ефективно виражають семантику коду, то наступним логічним кроком буде комбінування таких формалізованих представлень для створення бази знань. По-друге, дослідження можна спрямувати на розробку нових методів отримання фактів про початковий код і їх внесення в базу знань. Такими джерелами інформації можуть бути представлення коду, отримані при здійсненні збирання такого коду компілятором чи інтерпретатором, а також іншими обробниками початкового коду. Останній, але

не менш важливий напрям, – запропоновані методи та модель можна інтегрувати із існуючими методами і моделями для рефакторингу (включно з інтегрованими середовищами розробки) для отримання якісно нових результатів і можливостей виконання дій рефакторингу. Зокрема, мова може йти про комбіноване використання методів машинного навчання та запропонованих методів, що оперують формалізованими знаннями про код.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Boston: Addison-Wesley Professional.

Mens, T., & Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering, 2*(30), 126-139.

Murphy-Hill, E., Parnin, C., & Black, A. P. (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering, 1*(38), 5-18.

Dig, D., Manzoor, K., Johnson, R., & Nguyen, T. (2008). Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering, 3*(34), 321-335.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). 2013 International Conference on Software Engineering. *Why don't software developers use static analysis tools to find bugs?* (cc. 672-681). San Francisco: IEEE.

Kerievsky, J. (2004). *Refactoring to Patterns.* Boston: Addison-Wesley.

Harman, M., & Tratt, L. (2007). Pareto optimal search-based refactoring at the design level. *Annual conference on Genetic and evolutionary computation* (cc. 1106-1113). Boston: ACM SIGEVO.

Ouni, A., Kessentini, M., Sahraoui, H., & Hamdi, M. S. (2013). Search-based refactoring: Towards semantics preservation. *International Conference on Automated Software Engineering.* Boston: ACM/IEEE.

Boukadida, H., Kessentini, M., Bechikh, S., & Ben Said, L. (2015). Interactive search-based refactoring: Exploration and evaluation. *Genetic and Evolutionary Computation Conference.* Boston: ACM SIGEVO.

Tsantalis, N., & Chatzigeorgiou, A. (2009). Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering, 3*(35), 347-367.

Otaiby, A., Turki, M. H., & Touir, A. (2012). Refactoring Recommendation System Based on Association Rule Mining for Code Smell Correction. *International Journal of Software Engineering and Its Applications, 2*(6), 17-36.

Silva, D. A., Ramos, R., Valente, M. T., & Anquetil, N. (2016). Does the Modernization of Code Improve Its Quality? An Empirical Study. *Journal of Software: Evolution and Process, 11*(28), 924-950.

Opdyke, W. F. (1992). *Refactoring object-oriented frameworks. Ph.D. thesis.* Urbana-Champaign: University of Illinois.

Leino, K. R. (2010). Using logic as a specification language. *13th International Conference on Formal Engineering Methods.* Durham, UK.

Tokuda, L., & Batory, D. (2001). Automated Software Evolution via Design Pattern Transformations. *Third International Symposium on Automated and Analysis-Driven Debugging.*

Seng, O. S., & Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. *8th annual conference on Genetic and evolutionary computation* (cc. 1909-1916). Seattle: Association for Computing Machinery.

Kesseli, P. (2017). *Semantic Refactorings.* University of Oxford, St Cross College. Oxford: Hillary.

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American, 5*(284), 28-37.

Prud'hommeaux, E., & Seaborne, A. (2008). SPARQL query language for RDF. *W3C Recommendation, 15*.

Baeza-Yates, R., & Ribeiro-Neto, B. (2011). *Modern Information Retrieval.* New York: ACM press.

Nielson, H. R., & Nielson, F. (1992). *Semantics with applications: a formal introduction.* New Jersey: John Wiley & Sons, Inc.

Hitzler, P., & van Harmelen, F. (2010). A reasonable Semantic Web. *Semantic Web, 1, 2*(1), 39-44.

Opdyke, W. F., & Roberts, D. B. (1993). *Extending objects to support multiple interfaces and access control.* New Jersey: Bell Labs.

Xing, Z., & Stroulia, E. (05 November 2007 p.). API-evolution support with DiffCatchUp. *IEEE Transactions on Software Engineering, 33*(12), 818-836.

Murphy-Hill, E., & Black, A. P. (2008). Breaking the barriers to successful refactoring: Observations and tools for extract method. *ACM/IEEE 30th International Conference on Software Engineering.* Leipzig: IEEE.

Beller, M., Gousios, G., & Zaidman, A. (2019). Oops, my tests broke the build: An analysis of Travis CI builds with GitHub. *Empirical Software Engineering, 2*(24), 889-917.

Kim, M., Cai, D., & Kim, S. (2011). An empirical investigation into the role of API-level refactorings during software evolution. *33rd International Conference on Software Engineering* (cc. 141-150). Waikiki: IEEE.

Goetz, B., Bloch, J., Bowbeer, J., Holmes, D., Lea, D., & Peierls, T. (2006). *Java concurrency in practice.* Boston: Addison-Wesley.

Baader, F., Horrocks, I., & Sattler, U. (2005). Description logics as ontology languages for the semantic web. *Mechanizing Mathematical Reasoning*, 228-248.

van Harmelen, F., Patel-Schneider, P. F., & Horrocks, I. (2001). *Reference description of the DAML+OIL ontology markup language.* WWW Consortium.

McGuinness, D. L., & van Harmelen, F. (2004). OWL web ontology language overview. *W3C recommendation, 10*(10).

Motik, B., Patel-Schneider, P. F., & Parsia, B. (2012). *OWL 2 web ontology language structural specification and functional-style syntax (second edition).* W3C recommendation.

Baader, F., Brandt, S., & Lutz, C. (2005). Pushing the EL envelope. *19th international joint conference on Artificial intelligence* (cc. 364-369). San Francisco: Morgan Kaufmann Publishers Inc.

Calvanese, D., Giacomo, G. D., Lembo, D., Lenzerini, M., & Rosati, R. (2007). Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *Journal of Automated reasoning, 3*(39), 385-429.

Motik, B., Grau, B. C., Horrocks, I., Wu, Z., Fokoue, A., & Lutz, C. (2009). *OWL 2 Web Ontology Language Profiles (Second Edition).* W3C Recommendation.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition, 2*(5), 199-220.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., & Patel-Schneider, P. F. (2007). *The description logic handbook: Theory, implementation and applications.* Cambridge university press.

Smith, B., Ceusters, W., Klagges, B., Köhler, J., Kumar, A., Lomax, J., & Rosse, C. (2005). Relations in biomedical ontologies. *Genome biology, 5*(6), R46.

Allemang, D., & Hendler, J. (2011). *Semantic web for the working ontologist: Effective modeling in RDFS and OWL.* Elsevier.

Horrocks, I., Patel-Schneider, P. F., Boley, H., Tabet, S., Grosof, B., & Dean, M. (2004). SWRL: A semantic web rule language combining OWL and RuleML. *W3C Member submission, 21*(79), 31.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., & Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Web Semantics: science, services and agents on the World Wide Web, 2*(5), 51-53.

Glimm, B., Horrocks, I., Motik, B., Stoilos, G., & Wang, Z. (2014). HermiT: An OWL 2 reasoner. *Journal of Automated Reasoning, 3*(53), 245-269.

Tsarkov, D., & Horrocks, I. (2006). FaCT++ description logic reasoner: System description. *Int. Joint Conf. on Automated Reasoning*, (cc. 292-297).

Blackburn, P., de Rijke, M., & Venema, Y. (2001). *Modal logic.* Cambridge University Press.

Motik, B., Shearer, R., & Horrocks, I. (2009). Hypertableau reasoning for description logics. *Journal of Artificial Intelligence Research*(36), 165-228.

Vazirani, V. V. (2003). *Approximation algorithms.* Springer Science & Business Media.

Bryant, R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers, 8*(35), 677–691.

Grau, B., Horrocks, I., Kazakov, Y., & Sattler, U. (2007). Just the right amount: extracting modules from ontologies. *16th international conference on World Wide Web* (cc. 717–726). ACM.

Antoniou, G., & Van Harmelen, F. (2009). *A Semantic Web Primer (2nd ed.).* MIT Press.

O'Connor, M., & Das, A. (2009). SQWRL: a Query Language for OWL. *6th International Workshop on OWL: Experiences and Directions.* Chantilly, VA, USA.

Connor, M., Shankar, R., Tu, S., Parrish, D., & Das, A. (2009). Using a Semantic Wiki to Improve the Consistency and Analytic Completeness of Clinical Problem Lists. *13th World Congress on Medical and Health Informatics.* Brisbane, Australia.

Alcocer, J., Antezana, A., Santos, G., & Bergel, A. (2020). Improving the success rate of applying the extract method refactoring. *Sci. Comput. Program.*, 195.

Kaur, G., & Singh, B. (2017). Improving the quality of software by refactoring. *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, (cc. 185-191).

Saca, M. (2017). Refactoring improving the design of existing code. *37th Central America and Panama Convention* (cc. 1-3). IEEE.

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.

Inoue, K., & Roy, C. K. (2021). *Code Clone Analysis.* Singapore: Springer.

Bug tracking. (без дати). Отримано з Swift: https://swift.org

Swift issues. (10 2022 р.). Отримано з Swift: https://swift.org

Lacerda, G., Petrillo, F., Pimenta, M., & Guéhéneuc, Y. (2020). Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations. *ArXiv*, abs/2004.10777.

Almogahed, A., Omar, M., & Zakaria, N. H. (2022). Refactoring codes to improve software security requirements. *Procedia Computer Science*(204), 108–115.

Kaur, S., & Singh, P. (2019). How does object-oriented code refactoring influence software quality? research landscape and challenges. *Journal of Systems and Software*(157).

Morales, R., Soh, Z., Khomh, F., Antoniol, G., & Chicano, F. (2017). On the use of developers' context for automatic refactoring of software anti-patterns. *Journal of Systems and Software*(128), 236–251.

De Nicola, R., Di Stefano, L., Inverso, O., & Uwimbabazi, A. (2022). Automated replication of tuple spaces via static analysis. *Science of Computer Programming*(223).

Hammad, M., Babur, Ö., Basit, H. A., & van den Brand, M. (2022). Clone-Writer: An effective editor for developing code by using code clones. *Software Impacts*(13).

Al Dallal, J. (2012). Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Information and Software Technology, 10*(54), 1125–1141.

Tkachuk, A., & Bulakh, B. (2022). Usage of formalized knowledge about source code for refactoring actions in Swift. *Technology Audit and Production Reserves, 2(68)*(6), 6–10.

Happel, H.-J., & Seedorf, S. (2006). Applications of ontologies in software engineering. *Workshop on Sematic Web Enabled Software Engineering* (cc. 5-9). ISWC.

Tkachuk, A., & Bulakh, B. (2023). Describing the knowledge about the source code using an ontology. *Infocommunication and Computer Technologies, 5*(1), 114-122.

Ruta, M., Scioscia, F., Di Sciascio, E., & Bilenchi, I. (2016). OWL API for iOS: early implementation and results. *13th OWL: Experiences and Directions Workshop. 10161*, cc. 141–152. Springer.

Ruta, M. S., Gramegna, F., Bilenchi, I., & Di Sciascio, E. (2019). Mini-ME Swift: The First Mobile OWL Reasoner for iOS. *The Semantic Web. ESWC 2019. Lecture Notes in Computer Science. 11503*, cc. 298-313. Springer.

Matentzoglu, N., & Palmisano, I. (2016). *An Introduction to the OWL API*. Отримано з University of Manchester: http://syllabus.cs.manchester.ac.uk/pgt/2019/COMP62342/introduction-owl-api-msc.pdf

ONT-API: an RDF-centric Java library to work with OWL2. (2023). Отримано з GitHub: https://github.com/owlcs/ont-api

Dirsumilli, R. M. (без дати).

Dirsumilli, R., & Mossakowski, T. (2016). RESTful Encapsulation of OWL API. *5th International Conference on Data Management Technologies and Applications*, (cc. 150 - 157).

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.

Atzeni, M., & Atzori, M. (2017). CodeOntology [Data set].

Aguiar, C. Z., Zanetti, F., & Souza, V. E. (2021). Source Code Interoperability based on Ontology. *VII Brazilian Symposium on Information Systems* (cc. 1-8). ACM.

Pinto, G. H., & Kamei, F. (2013). What programmers say about refactoring tools? *Workshop on refactoring tools* (cc. 33-36). ACM.

Catalog of Refactoring. (10 2023 р.). Отримано з Refactoring Guru: https://refactoring.guru/refactoring/catalog

Horrocks, I., Patel-Schneider, P. F., & van Harmelen, F. (2003). From SHIQ and RDF to OWL: the making of a Web Ontology Language. *Web Semantics: Science, Services and Agents on the World Wide Web, 1*(1), 7-26.

Ткачук, А. (2020). *Аналіз і рефакторинг програмного коду з використанням бази знань*. Київ: НТУУ "Київський політехнічний інститут імені Ігоря Сікорського".