Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА

Метод оцінки якості найменування ідентифікаторів коду програмного забезпечення
Назва теми

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДРПЗ.170120.01.10.ПЗ

Виконав студент 2 курсу, група ІПЗм-21-1 _____ В. В. Стьопич
Підпис Ініціали, прізвище

Керівник д-р фіз.-мат. наук, проф. _____ Л. П. Бедратюк
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. техн. наук, доцент _____ Г. І. Радельчук
Науковий ступінь, звання Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

5 грудня 2022 р.

Хмельницький 2022

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Н. Н. Бедратюк
01 09 2022 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ)

Стьопичу Владиславу Валерійовичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Метод оцінки якості найменування ідентифікаторів коду програмного забезпечення

Керівник проєкту (роботи) д-р фіз.-мат. наук, проф. Бедратюк Н. Н.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 01.07.2022 р. № 83
2. Строк подання студентом проєкту (роботи) на кафедру 01.12.2022 р.
3. Вихідні дані до проєкту (роботи) Матеріали переддипломної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1 Аналіз предметної областіта рішень з оцінки якості найменування ідентифікаторів коду програмного забезпечення.
2 Удосконалення методу оцінки якості найменування ідентифікаторів коду програмного забезпечення та розробка алгоритму граматичних шаблонів покращення найменування ідентифікаторів вихідного коду програмного забезпечення
3 Архітектура програмної реалізації
4 Програмна реалізація

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата завдання видав	Підпис, дата завдання прийняв
Антиплагіат	Гурман І. В., доцент	26.11.22	1.12
Нормоконтроль	Радельчук Г. І., доцент		

7. Дата видачі завдання «01» вересня 2022 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітки
1 Вивчення предметної області; формулювання мети та задачдослідження; визначення об'єкта та предмета дослідження;	01.09-10.09.2022	
2 Робота над розділом 1 дипломної роботи аналіз відомих моделей, методів та засобів за темою роботи; визначення методологічних підходів до вирішення задачі; висновки до розділу та постановка задач дослідження	11.09-25.09.2022	
3 Робота над розділом 2 дипломної роботи – розробка моделей,методів та алгоритмів вирішення задачі; висновки до розділу	26.09-10.10.2022	
4 Робота над науковими статтями	11.10-30.10.2022	
5 Робота над розділом 3 дипломної роботи – розробка інформаційної технології вирішення задачі (аналіз вимог до програмного засобу та його проектування, аналіз та вибір засобів реалізації програми); висновки до розділу	11.10-26.10.2022	
6 Робота над розділом 4 дипломної роботи – програмна реалізація спроектованих рішень, результати експериментів та їх аналіз; дослідження ефективності запропонованих рішень; висновки до розділу	27.10-17.11.2022	
7 Попередній захист дипломної роботи	Листопад	
8 Перевірка роботи на наявність плагіату; нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	01.12-04.12.2022	
9 Підготовка до захисту дипломної роботи	з 01.12.2022 р.	

Студент _____ В. В. Стьопич
Підпис Ініціали, прізвище

Керівник проєкту (роботи) _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: «Метод оцінки якості найменування ідентифікаторів коду програмного забезпечення».

Автор роботи: Стьопич Владислав Валерійович.

Керівник роботи: Бедратюк Леонід Петрович.

Пояснювальна записка: 88 с., 24 рис., 3 дод., 39 джерел.

ІДЕНТИФІКАТОР КОДУ, ПРОГРАМНА СИСТЕМА, ЗАСОБИ КОНТРОЛЮ ЯКОСТІ, ПРОГРАМНИЙ ПРОДУКТ, ОЦІНКА ЯКОСТІ НАЙМЕНУВАННЯ ІДЕНТИФІКАТОРА.

Мета роботи – удосконалення методу оцінки якості найменування ідентифікаторів вихідного коду програмних систем та розробка відповідного алгоритму лінгвістичних шаблонів покращення якості.

Предмет – якість найменування ідентифікаторів вихідного коду програмної системи.

Об’єкт – процес створення концепції вимірювання та покращення якості ідентифікаторів.

Виходячи із мети, предмету, об’єкту даної магістерської роботи було виокремлено такі завдання дослідження:

– здійснити аналіз предметної області;

– проаналізувати існуючі методи оцінки якості найменування ідентифікаторів вихідного коду програмної системи;

– здійснити удосконалення методу оцінки якості найменування ідентифікаторів вихідного коду програмних систем;

– розробити алгоритм граматичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи із наданням рекомендацій щодо структури імен.

Наукова новизна:

– удосконалено метод оцінки якості найменування ідентифікаторів вихідного коду програмних систем;

– розроблено алгоритм лінгвістичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи із наданням рекомендацій щодо структури імен.

Практичне значення отриманих результатів. У даній магістерській роботі здійснено спробу встановити нові концепції алгоритму, які допомагають вимірювати та покращувати якість ідентифікаторів. Результатом цієї магістерської роботи є набір інструментів оцінки найменування ідентифікаторів за якості, які інтегруються в робочий процес розробника. Через послідовність емпіричних досліджень сформулювали низку методик та лінгвістичних шаблонів для оцінки якості ідентифікатора імен в коді та надати рекомендації щодо структури імен. Результати даного дослідження можуть покращити роботу програмістів, шляхом їх інтеграції у творчий процес розробки, що в свою чергу забезпечить процес створення та підтримки високої якості найменувань ідентифікаторів у вихідному коді програмних систем.

Підпис

01.12.2022
Дата

ABSTRACT

Master’s thesis: «A method for assessing the quality of naming software code identifiers».

Author: Stopych Vladyslav Valeriyovych.

Head of work: Leonid PetrovychBedratyuk.

Master’s thesis consists of: 88 pages of the general text, 24 graphics, 3 supplements, 39 literature sources.

CODE IDENTIFIER, SOFTWARE SYSTEM, QUALITY CONTROL TOOLS, SOFTWARE PRODUCT, IDENTIFIER QUALITY ASSESSMENT.

The purpose of the work is to improve the method of improving the quality of naming identifiers of the source code of software systems and to develop the corresponding algorithm of linguistic patterns for quality improvement.

The subject is the quality of the naming of identifiers of the source code of the software system.

The object is the process of creating a concept for measuring and improving the quality of identifiers.

Based on the goal, subject, object of this master's thesis, the following research tasks were identified:

– carry out an analysis of the subject area;

– to analyze the existing methods of improving the quality of naming identifiers of the source code of the software system;

– to improve the method of improving the quality of naming identifiers of the source code of software systems;

– develop an algorithm of linguistic patterns to improve the quality of naming identifiers of the source code of the software system with recommendations on the structure of names.

Scientific innovation:

– the method of improving the quality of naming identifiers of the source code of software systems has been improved;

– an algorithm of linguistic patterns to improve the quality of naming identifiers the source code of the software system was developed, with recommendations on the ructure of names.

Practical significance of the obtained results. In this master's thesis, an attempt as made to establish new algorithm concepts that help to measure and improve the ıality of identifiers. The result of this master's thesis is a set of tools for the assessment f naming identifiers and quality, which are integrated into the developer's workflow. hrough a series of empirical studies, a number of techniques and linguistic patterns ave been formulated to evaluate the quality of the identifier of names in code and to rovide recommendations for the structure of names. The results of this study can nprove the work of programmers by integrating them into the creative process of evelopment, which in turn will ensure the process of creating and maintaining high-uality names of identifiers in the source code of software systems.

Signature

01.12.2022
Date

ЗМІСТ

ВСТУП

Питанню якості програмного забезпечення, зокрема програмних систем приділено досить велику увагу. Однак, якість імен ідентифікаторів є важливим та критичним аспектом. Даному питанню присвячені роботи таких вчених як Лавріщєвої К.М., яка займалась питаннями програмної інженерії та якістю програмного забезпечення зокрема; Коцовський В.М. розглядав супровід програмних систем в контексті якості. Підходами до оцінки якості імен ідентифікаторів займались такі науковці як Абебе, Тонелла, Лібліт, Хофмайстер, Лю, Хост, Освольд та багато інших.

Значну частину вихідного коду складають імена ідентифікаторів – унікальні лексичні токени, які надають інформацію про сутності та взаємодію сутностей у коді. Було заявлено, що ідентифікатори складають приблизно 70% символів у програмній системі [15]. Через це їх осмислене іменування має вирішальне значення для розуміння програми.

Імена ідентифікаторів надають зрозумілі людині описи класів, функцій, змінних тощо. Неякісні або неоднозначні імена ідентифікаторів (тобто імена, які неправильно описують поведінку коду, з якою вони пов’язані), змушують розробників витрачати більше часу на роботу над розумінням поведінки коду. Неякісні імена також можуть мати шкідливий вплив на інструменти, які покладаються на підказки природної мови; погіршення якості їхньої продукції та робить їх ненадійними. Крім того, спричинені неправильним тлумаченням коду неякісні та невідповідні назви, можуть призвести до появи проблем з якістю в системі, що обслуговується.

Таким чином, покращене найменування ідентифікаторів підвищує ефективність роботи самого розробника, і в цілому покращує та підвищує якість програмного забезпечення та якісніші засоби аналізу програмного забезпечення.

Мета роботи – удосконалення методу оцінки якості найменування ідентифікаторів вихідного коду програмних систем та розробка відповідного алгоритму лінгвістичних шаблонів покращення якості.

Предмет – якість найменування ідентифікаторів вихідного коду програмної системи.

Об’єкт – процес створення концепції вимірювання та покращення якості ідентифікаторів.

Виходячи із мети, предмету, об’єкту даної магістерської роботи можна виокремити такі завдання дослідження:

– здійснити аналіз предметної області;

– проаналізувати існуючі методи оцінки якості найменування ідентифікаторів вихідного коду програмної системи;

– здійснити удосконалення методу оцінки якості найменування ідентифікаторів вихідного коду програмних систем;

– розробити алгоритм граматичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи із наданням рекомендацій щодо структури імен.

Наукова новизна:

– удосконалено метод оцінки якості найменування ідентифікаторів вихідного коду програмних систем;

– розроблено алгоритм граматичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи із наданням рекомендацій щодо структури імен.

Практичне значення отриманих результатів. У даній магістерській роботі здійснено спробу встановити нові концепції алгоритму, які допомагають вимірювати та покращувати якість ідентифікаторів. Результатом цієї магістерської роботи є набір інструментів оцінки найменування ідентифікаторів та якості, які інтегруються в робочий процес розробника. Через послідовність емпіричних досліджень сформулювали низку методик та лінгвістичних шаблонів для оцінки якості ідентифікатора імена в коді та надати рекомендації щодо структури імен.

Результати даного дослідження можуть покращити роботу програмістів, шляхом їх інтеграції у творчий процес розробки, що в свою чергу забезпечить

процес створення та підтримки високої якості найменувань ідентифікаторів у вихідному коді програмних систем.

Доцільність та ефективність розробки системи теоретично обґрунтована у першому та другому розділах кваліфікаційної роботи опираючись, випливають із результатів емпіричних досліджень.

Теоретичними методами дослідження виступають: аналіз, синтез, абстрагування та порівняння.

Емпіричними методами дослідження виступають: опис та тестування.

За темою кваліфікаційної роботи опубліковані тези «Оцінювання імен ідентифікаторів вихідного програмного коду» на конференції «Актуальні проблеми комп’ютерних наук АПКН-2022».

1 ТЕОРЕТИЧНИЙ ВИКЛАД ДОСЛІДЖУВАНОЇ ПРОБЛЕМИ

1.1 Аналіз предметної області

Незалежно від галузі чи технічної сфери, якість є критичним аспектом будь-якої програмної системи [13] та програмного забезпечення вцілому. Крім того, забезпечення якості програмного забезпечення не є одноразовим завданням; починаючи з моменту впровадження та продовжуючи протягом усього терміну служби системи, розробники виконують технічне обслуговування завдання на їх програмне забезпечення для досягнення функціональних і нефункціональних цілей. Цей наголос на якості програмного забезпечення призвів до того, що організації виділяють від 60% до 80% ресурсів на технічне обслуговування програмного забезпечення [19], що є найдорожчою фазою життєвого циклу розробки програмного забезпечення.

Ключовим елементом підтримки програмного забезпечення є розуміння програми. Розуміння програми – це дії розробників, які читають вихідний код, щоб зрозуміти призначення коду або визначити вимоги, пов'язані з їх діяльністю з обслуговування. До внесення змін до коду, щоб полегшити розвиток програмної системи, розробникам необхідно читати рядки коду у вихідному коді файлу для того, що розуміти поведінку коду. Ясно, що такі питання як погана читабельність і зрозумілість коду впливають не тільки на час, який витрачають розробники при виконанні своїх завдань, але також можуть опосередковано чи напряму впливати на якість оновлень, що виконуються в системі.

Крім того, відмінності у коді (наприклад, навички, досвід) між оригінальним авторським кодом та супроводжуючою документацією також впливають на розуміння. Зі всього часу розробника 58% витрачається на діяльність по розумінню. Тому важливо, щоб розробники створювали вихідний код таким чином, щоб він не перешкоджав читабельності та зрозумілості. Це включає вдосконалення коду, починаючи від змін рівня проектування, таких як зменшення цикломатичної складності [19] до відповідної узгодженості, у формі домовленостей про найменування.

Будучи фундаментальними елементами у вихідному коді, імена ідентифікаторів складають майже 70% символів у кодовій частині програмної системи. Ці назви є лексемами, які однозначно ідентифікують сутності в коді (наприклад, класи, методи, змінні тощо) і відіграють значну роль у розумінні коду. Важливість та необхідність якісних та хороших імен ідентифікаторів визнають як розробники так і наукові кола. Ця важливість відображається в практиках розробки програмного забезпечення, які забезпечують рекомендації, найкращі практики, показники та моделі, щоб допомогти розробникам у іменування ідентифікаторів із загальною метою покращення розуміння коду [2]. Тому розробники повинні приділяти значну увагу створенню назви ідентифікатора, оскільки їхній вибір впливає на час, витрачений на розуміння мети ідентифікатора; а добре побудовані імена можуть покращитися діяльність розуміння аж на 19% [4].

Хоча показники якості, найкращі практики та рекомендації підкреслюють потребу у високоякісному найменуванні ідентифікатора, вони діють лише як теоретичні аспекти для розробників; вони не є формальними механізмами використання адекватних та потрібних імен. Ці механізми не можуть допомогти розробникам використовувати правильні формулювання, рекомендувати лексичну структуру за межами того, що є методично та теоретично правильним (наприклад, правила іменування).

Наприклад, представлений код у лістингу 1, є чітким свідченням недоречного імені ідентифікатора, які можна виявити за допомогою стандартних домовленостей про іменування. Однак визначення якості в імені ідентифікаторів у лістингу 2 не є простим. У цих двох прикладах імена дотримуються певного стилю імен (наприклад, так званий верблюжий регістр), використовують відомий набір термінів і є читабельними.

Однак вони не точно відображають очікувану поведінку ідентифікатора. У лістингу 2 ім'я ідентифікатора в однині, але пов'язаний з ним тип даних є колекцією. Також не зрозумілим є те чи представляє цей ідентифікатор сукупність URL-адрес чи одну URL-адресу. Крім того назва методу передбачає перетворення

даного об’єкта, але метод не повертає значення, показники якості навколо читабельності, як правило, не в змозі зафіксувати покращення читабельності.

```
@Override
protectedboolean fanc_345678_b (int var3 )
{
returnfalse ;
}
```

Лістинг 1 – Приклад назви ідентифікатора низької якості (fanc_345678_b).

```
privateHashSet
includeCrawlingURL;
```

Лістинг 2 – Приклад поганого, але читабельного імені ідентифікатора («includeCrawlingURL»).

Щоб виправити погані імена ідентифікаторів, подібні до тих, що показані в наведених вище прикладах, розробники виконують операцію рефакторингу перейменування імені ідентифікатора, яка або зберігає оригінал значення імені або змінює його. Наприклад, щоб виправити неякісне ім'я ідентифікатора у лістингу 1 розробник виконує операцію рефакторингуRenameMethod, результатом якого є високоякісне ім'я. Таким чином, більшість сучасних підходів, запропонованих дослідницьким співтовариством, покладаються на виявлення та пропонування можливостей перейменування ідентифікатора у вихідному коді, використовуючи методи машинного навчання. Однак, у міру наближення цих значень до існуючого коду, існуючі неякісні та неправильні імена ідентифікаторів у вихідному коді негативно вплинуть на якість згенерованого імені і як наслідок вони можуть навіть призвести до зниження якості системи, що обслуговується.

Таким чином, їх повністю автоматизована природа, що керується великими даними може працювати проти них. Краще розуміння структур найменування з коду поведінки і лінгво-семантичні перспективи можуть допомогти правильно

навчати ці інструменти, щоб не тільки використовувати історичні дані, але також звернути увагу на людське розуміння значення імені, засноване як на лінгвістичній, так і програмно-орієнтованій інформації. Іншими словами, можна використовувати емпіричні дані, отримані людиною, щоб спрямувати ці інструменти на кращі рішення, а не лише на патерни, що отримані на основі алгоритмів.

1.2 Аналіз існуючих рішень до оцінки імен ідентифікаторів

Аналіз відповідної наукової літератури вказує на те, що існує досить великий спектр підходів до оцінки імен ідентифікаторів. Найновішими вважаються дослідження таких вчених як Лю, Абебе, Тонелли, Лібліта, Хофмайстера та інших. Розглянемо їх далі більш детальніше.

Зокрема, дослідник Лю та інші [27] пропонують автоматизований підхід, заснований на методі глибокого навчання для налагодження назви на основі узгодженості між назвою методу та його реалізацією. Доцільним є використання аналізу правил асоціації, щоб визначити дієслова, які можуть бути хорошими кандидатами для використання в назви методів; ця робота зосереджена на співпоширенні слів, щоб знайти будь-які зв’язки, що виникають. Абебе і Тонелла [28] використовують онтологію, яка моделює зв’язки слів у частині програмного забезпечення. У своїх працях вони здійснюють генерацію пропозицій для нових імен ідентифікаторів, використовуючи різні схеми вибору послідовності слів для створення ідентифікатора.

Лібліт та ін. [29] здіснили обговорення іменування в кількох мовах програмування та роблять зауваження щодо впливу природної мови на використання слів у цих мовах. Також вони зосередились на дослідженнях впливу більш інформативних ідентифікаторів на розуміння коду. Їх знахідки показують перевагу описових ідентифікаторів над неописовими.

Хофмайстер та ін. [30] здійснили порівняння розуміння ідентифікаторів, що містять слова, проти ідентифікаторів, що містять літери та/або скорочення. Їх результати показують, що назви ідентифікаторів містять лише слова замість абревіатур або букви, що збільшує швидкість розуміння розробником у середньому на 19%. Дослідження проводились за участі великої кількості програмістів, яких просять описати дванадцять різних функцій. Ці функції використовують три різні рівні ідентифікаторів: окремі літери, абревіатури та повні слова. Результати свідчать, що повні ідентифікатори слів забезпечують найкраще розуміння, хоча були випадки, коли їх не було і відмічалась статистична різниця між повними словами та абревіатурами.

Батлер та ін. [31] розширюють свої попередні праці над ідентифікаторами класів Java, щоб показати, що ідентифікатори методів з недоліками також є (тобто разом з ідентифікаторами класів), пов’язані з низькоякісним кодом відповідно до показників на основі статичного аналізу.

Хост і Освольд [32] розробляють автоматичні правила іменування, використовуючи елементи підпису методу, тобто тип повернення, імена параметрів і типи, а також потік керування. Цю техніку вони називають методом уточнення фрази, яка бере послідовність тегів частини мови (тобто фраз) і конкретизує їх, замінюючи справжні слова. (наприклад, фраза <дієслово>-<прикметник> може бути уточнена на is-empty).

Крім того, вони використовують статичний аналіз для групування імен методів (у формі фраз) разом із поведінкою. Визначають каталог лінгвістичних антипаттернів, які, як виявлено, погіршують якість розуміння коду. Автори показують негативний вплив мовних антипатернів провівши два дослідження з розробниками програмного забезпечення та виявивши, що більшість програмістів сприймати антишаблони як погану практику іменування. У своєму дослідженні показників читабельності Фахурі та ін. [33] показують, що поточні показники можуть бути неефективними для фіксації покращень читабельності; підкреслюючи важливість подальших досліджень якості іменування та того, як імена розвиваються через деякий час.

Граматичні шаблони ідентифікатора. Як показують дослідження науковців іменники, дієслова, прикметники – це три найпоширеніші теги частин мови, які розробники використовують у створенні імен ідентифікаторів. Автори використовують StandfordParser для аналізу тегів частини мови на ім'я ідентифікатора автоматично. Бінклі та ін. [34] досліджують ефективність Стенфордського університету лог-лінійний тег частини мови для імен полів. Завдяки цьому дослідженню автори пропонують чотири правила, засновані на тегах частини мови, для покращення імен полів. Дослідження іменування в множині мови програмування показує, як природна мова впливає на використання слів у цих мовах. Хост та Освольдт [32] досліджують незвичайні назви методів і пропонують набір правил іменування для виявлення проблем у назвах методів. Автори використовують теги частин мови разом із типом повернення, потоком керування та параметрами методу для виявлення порушень імен на основі заздалегідь складеного набору правил.

Перейменування ідентифікатора. Алламаніс та інші [35] представили модель, що має назву «Naturalize», яка використовує статистичну природну мову обробка для аналізу та вивчення стилю (тобто умов кодування) кодової частини та надає пропозиції щодо перейменування. Модель «Naturalize» вивчає синтаксичні обмеження або підграматики ідентифікатора таких імен, що використовують верблюжу букву або підкреслення, і для уніфікації імен, що використовуються в подібних контекстах. Автори також запропонувати нейронну імовірнісну модель мови, щоб запропоновувати описовий, ідіоматичний метод для імен ідентифікаторів автоматично.

Сузукі та ін. [36] представили модель для оцінювання на основі зрозумілості імен методів. У своїх підходах, автори збирають і вивчають назви методів із відкритих систем Java. Як частина у своїй стратегії аналізу автори використовують порогове значення для визначення показника зрозумілості ім’я методу та використання моделі n-грам, щоб надавати пропозиції розробнику. Дослідження за Лю та ін. [27] розглядає рекомендації щодо перейменування на основі попередніх дій розробників виконати на вихідному коді. Крім того,

вивчаючи зв’язок між аргументами та назвами параметрів, автори розробляють підхід до виявлення аномалій іменування та пропонують перейменування розробникам. Також було відмічено, що підхід, заснований на машинному навчанні для рекомендацій щодо імен методів не працює в реалістичних умовах.

Автори також пропонують евристичний підхід, що полягає у використанні методів глибокого навчання для виявлення непослідовних імен методів. Їхній підхід витягує глибокі представлення назв методів. Автори вдосконалюють модель за допомогою великої кількість методів із реальних проектів. Підхід до назви передбачає порівняння перекриття між близькістю імен методів у векторному просторі імен методів і набором імен методів, тіла яких розташовані близько у векторному просторі тіла методу. Також дослідники представили підхід до аналізу та класифікації перейменувань ідентифікаторів. Автори показують вплив правильного найменування на мінімізації зусиль щодо розробки програмного забезпечення та виявили, що 68% розробників вважають доцільною та корисною рекомендацію імен ідентифікаторів [32, 33, 34].

1.3 Огляд методів вирішення проблеми

Лексикосемантичний метод. Лексикосемантика (або лексико-семантика або лексична семантика) – це розділ лінгвістики, який займається вивченням значень слів, наприклад внутрішньої семантичної структури слів і зв'язок між різними значеннями слова. В даному підпункті подаються поняття, шо були використані в в даному досліджені для визначення якості найменування ідентифікатора.

Щоб оцінити рефакторинги перейменування та згрупувати їх у різні види на основі їхньої семантики використовується таксономія рефакторингів, у якій тип сутності, що записує сутність вихідного коду представляє даний ідентифікатор. До прикладу, ідентифікатор може бути назвою типу, класу тощо.

Форма перейменування: лексичні зміни ідентифікатора відображаються в цій категорії та складаються з чотири підкатегорії: прості, складні,

перевпорядкування та форматування. Прості зміни ідентифікатора ті, які просто додають, видаляють або замінюють один термін. Додано, видалено або змінено кілька термінів, що відображається у комплексних змінах. Перевпорядкування відбувається, коли два або більше термінів в ідентифікаторі змінюють позиції (наприклад, NameEmployeeзмінюється на EmployeeName), тоді як зміни форматування відбуваються, коли літера в терміні змінює регістр або додається чи видаляється роздільник (наприклад, підкреслення).

Семантичні зміни: це модифікації значення ідентифікатора в результаті додавання, видалення, зміни термінів (наприклад, використання терміна, який є синонімом або антонімом оригінальному). Щоб визначити, чи було збережено або змінено семантику ідентифікатора, використовуються наступні методики.

Значення ідентифікатора зберігається, якщо виконується одне із наведеного нижче правила:

– під час зміни додано або видалено роздільник;

– зміна розширила абревіатуру;

– зміна згорнула термін у абревіатуру;

– старий термін замінено на новий термін, який є синонімом старого терміну;

– кілька старих термінів було змінено на кілька нових термінів, які є синонімами чи використання або видалення заперечення зберігає значення ідентифікатора (тобто ItemNotVisible стає ItemHidden).

Значення ідентифікатора змінюється, якщо виконується одне з наступного:

– розширення значення – старий термін перейменовано або термін (тобто прикметник чи іменник) було видалено, що узагальнює ідентифікатор (наприклад, GetEmployeeFirstName стає GetEmployeeName);

– звуження значення – старий термін перейменовано на гіпонім самого себе або термін було видалено, що звужує значення ідентифікатора (наприклад, GetEmployeeName стає GetEmployeeFirstName);

– зміна значення (тобто не звужений чи розширений) – коли старий термін змінюється на новий термін, який не пов'язаний зі старим; коли новий термін є

меронімом, голонімом або антонімом старого терміна; або коли кілька термінів змінено і заперечення змінює синонім старого терміна;

– додавання значення – до ідентифікатора було додано один або кілька нових термінів та додавання не входить до однієї з категорії вище (наприклад, вузьке значення).

– видалення значення – один або кілька термінів, вилучених із ідентифікатора та видалення не підпадає під одну із категорій, описаних вище (наприклад, розширене значення).

Метод контекстуалізації перейменування рефакторингу. Коли ім’я ідентифікатора більше не відповідає належному опису ролі ідентифікатора в програмному забезпеченні системи, розробник змінить назву, тобто відбудеться рефакторинг перейменування, який є поширеним типом рефакторингу та вва–жається частиною таксономії Фаулера [13]. Рефакторинг перейменування змінює нефункціональні атрибути програмної системи, тобто назву ідентифікатора.

Під час перейменування ідентифікатора нове ім’я має краще відповідати опису ролі та роботи ідентифікатора поточного стану системи, ніж стара назва. Вивчення рефакторингу перейменування набирає обертів та йому приділяється все більше уваги в наукових дослідженнях. Зрозумілим є факт, що потрібне більш глибоке розуміння використання природної мови у підтримці та розвитку програмного забезпечення.

Багато методів для підтримки розуміння покладаються на аналіз ідентифікаторів. Крім того, багато попередніх досліджень присвячені практикам іменування, шаблонам та покращенню аналізу імен ідентифікаторів. Зокрема низка наукових праць висвітлює дослідити ідею налагодження, оцінки та генерації імен ідентифікаторів [22, 31, 32]. Це має прямий, позитивний вплив на підходи, які синтезують програми, які повинні зрозуміти, як розробники описують елементи коду (наприклад, ідентифікатори імен, коментарі до методів), щоб створити природну мову, прийнятну розробниками (тобто текст, який оптимізує розробник розуміння). Таким чином, аналіз найменування

ідентифікаторів та використання його результатів на практиці є актуальним завданням, особливо якщо їх виконують розробники в реальних сценаріях.

Ідентифікатори перейменовуються розробниками з різних причин. Можна отримати уявлення про те, як розробники вибрали свої слова, чому вони віддають перевагу одним видам слів над іншими та як автоматизувати цей процес шляхом ретельного аналізу рефакторингу перейменування. У цьому підрозділі показано приклади того, як це зробити за допомогою розробника, а також записати в повідомленнях фіксації та операції рефакторингу, щоб одночасно відображати їх варіанти перейменування.

Аналізуючи таке перейменування методу: з setDisableBinLogCache на setEnableReplicationCache, помічено, що значення імені змінилося; розробник змінив назву на зміна вимкнення на увімкнення. Ця зміна відображається в повідомленні про фіксацію, введеному розробником:

«Змінює кешування реплікації на вимкнене за замовчуванням» [37]. Так само перейменування класу з Key на EntityKey демонструє дію звуження значення ідентифікатора. Тобто мета цього перейменування відображається в повідомленні: «Перейменувати ключ на EntityKey» [26].

Розробники також можуть перейменовувати ідентифікатори, щоб: 1) краще представляти наявну функціональність, а не тоді, коли вони змінюють код або звужують його, або 2) дотримуються стандартів найменування чи виправляють правопис або граматичні помилки. Наприклад, тут розробник перейменував клас TestProxyController на ProxyControllerTest шляхом зміни порядку назв термінів на «...фіксовані імена, що відсутні у стандартах». У наступному прикладі розробник зберігає значення методу шляхом перейменування від інактивації до деактивації, використовуючи синонім. Це, знову ж таки, відображено в повідомленні коміту: «Метод перейменування на правильну англійську...» [9], де перейменування на «справжню англійську» вказує на те, що значення не було змінено, але тепер має бути легшим для розуміння.

Нарешті, повідомлення про фіксацію – не єдиний спосіб контекстуалізації рефакторингу перейменування. Зміни у коді навколо назви також допомагає

зрозуміти намір розробника. На жаль, більшість типів змін до коду не є частиною попередньо визначеної таксономії. Тобто важко зрозуміти абстрактну мету індивідуальних змін на рівні домену. Однак, деякі типи коду зміни таксономізовані. Зокрема, рефакторинг – це класифікація змін, внесених у код для конкретної мети; зазвичай для оптимізації нефункціональних атрибутів коду. Ми можемо це побачити на прикладі рефакторингів, які відбуваються безпосередньо перед і відразу після даного перейменування, щоб допомогти зрозуміти, що розробник робив до і після того, як застосував рефакторинг перейменування.

Наприклад, у коміті [30] розробники застосували рефакторинг методу вилучення з наступним коментарем: «використання інфраструктури аналізу Jangaroo», раніше застосування перейменування: з getCompilationsUnit на getCompilationUnit. Це зберігає значення ім'я, але розміщує ім'я в більшій відповідності до його типу, як зазначено в повідомленні коміту для цієї зміни: «Виправлений тип у назві внутрішнього методу» [31].

Іншим прикладом є рефакторинг класу переміщення, коли клас було переміщено з одного пакету в інший [28]. Цей коммітрефакторингу мав такий коментар: «Поступові зміни, деякі рефакторинги пакетів тощо». Крім того, після цього коміту було виконано перейменування: з JsonViewResult на JsonView [47]. Це перейменування розширює значення імені, видаляючи результат, роблячи ідентифікатор більш загальним за значенням. Повідомлення фіксації, пов’язане з перейменуванням, таке:

«Очищено деякі назви файлів для полегшення використання...», тобто розробник, імовірно, проходив через це і перейменування речей після рефакторингу класу переміщення.

На додаток до змін у навколишньому коді, зміна типу даних, пов’язаного з ідентифікатором також може допомогти контекстуалізувати перейменування ідентифікатора. Наприклад, у комміті [30] розробник виконує наступне перейменування змінної: DatesqlDate на Timestamptimestamp із фіксацією повідомлення «вирішує проблему екземпляри java.util.Date та jodatime.Datetime втратили б інформацію про час...» З цього прикладу видно, що причиною

перейменування є виправлення помилки за допомогою структури даних мітки часу замість дати.

1.4 Постановка задачі

В ході проведеного дослідження було поставлено мету – удосконалення методу покращення якості найменування ідентифікаторів вихідного коду програмних систем та розробка відповідного алгоритму лінгвістичних шаблонів покращення якості.

Згідно із метою визначено такі завдання:

– здійснити аналіз предметної області;

– проаналізувати існуючі методи оцінки якості найменування ідентифікаторів вихідного коду програмної системи;

– здійснити удосконалення методу оцінки якості найменування ідентифікаторів вихідного коду програмних систем;

– розробити алгоритм граматичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи із наданням рекомендацій щодо структури імен.

Отже, в даній роботі має бути здійснено аналіз еволюції найменування ідентифікаторів у вихідному коді для виявлення мовних моделей з точки зору семантики із використанням ручних, частково автоматизованих даних які потім можуть використовуватися для навчання та керування повністю автоматизованими техніками для розуміння значущих структур і семантики ідентифікатора імен.

Результати даного дослідження мають на меті допомагати розробникам не лише створювати імена ідентифікаторів, але також краще керувати ідентифікаторами у вихідному коді їхніх проектів.

1.5 Висновки

У першому розділі дипломної роботи магістра проаналізовано предметну область та сучасний стан досліджуваної проблеми.

У підрозділі 1.1 було встановлено, що ключовим елементом підтримки програмного забезпечення є розуміння програми. Розуміння програми – це дії розробників, які читають вихідний код, щоб зрозуміти призначення коду або визначити вимоги, пов'язані з їх діяльністю з обслуговування. До внесення змін до коду, щоб полегшити розвиток програмної системи, розробникам необхідно читати рядки коду у вихідному коді файлу для того, що розуміти поведінку коду. Ясно, що такі питання як погана читабельність і зрозумілість коду впливають не тільки на час, який витрачають розробники при виконанні своїх завдань, але також можуть опосередковано чи напряму впливати на якість оновлень, що виконуються в системі.

У підрозділі 1.2 було здійснено аналіз методів оцінки якості найменувань ідентифікаторів коду програмного забезпечення. Встановлено, що існує досить великий спектр підходів до оцінки імен ідентифікаторів. Найновішими вважаються дослідження таких вчених як Лю, Абебе, Тонелли, Лібліта, Хофмайстера та інших.

У підрозділі 1.3 було здійснено постановку задачі та окреслено шляхи її можливого розв’язання.

В результаті опису першого розділу було встановлено мету роботи, що полягає в удосконаленні методу оцінки якості найменування ідентифікаторів вихідного коду програмних систем та розробка відповідного алгоритму лінгвістичних шаблонів покращення якості. Також було визначено об’єкт та предмет дослідження і окреслено коло завдань, що необхідно вирішити для досягнення поставленої мети.

2 МЕТОД ОЦІНКИ ЯКОСТІ НАЙМЕНУВАННЯ ІДЕНТИФІКАТОРІВ ВИХІДНОГО КОДУ ТА АЛГОРИТМ ГРАМАТИЧНИХ ШАБЛОНІВ ПОКРАЩЕННЯ ЙОГО ЯКОСТІ

2.1 Концептуальна модель процесу оцінки якості імен ідентифікаторів

Більшість ресурсів життєвого циклу програмного забезпечення виділяється на обслуговування програми. Технічне обслуговування значною мірою залежить від розуміння програми, оскільки розробники, зазвичай, витрачають значну частину свого часу на розуміння коду, який вони підтримують перед застосуванням змін, налагодження, документування тощо. Зрозуміло, що полегшення розуміння коду полегшить обслуговування та підвищення продуктивності розробників. Один із основних способів для розробника зрозуміти, що робить частина коду, це розуміння значення та ролі ідентифікаторіву коді.

Як уже зазначалось, ідентифікатори складають приблизно 70% символів у програмній системі. Через це їх осмислене іменування має вирішальне значення для розуміння програми.

Коли ім’я ідентифікатора більше не відповідає належному опису ролі ідентифікатора в програмному забезпеченні системи, розробник може змінити назву. Ця зміна називається рефакторингом перейменування. Перейменувати рефакторинг є поширеним типом рефакторингу і є частиною таксономії Фаулера [13]. Рефакторинг перейменування змінює нефункціональні атрибути програмної системи, тобто найменування ідентифікатора.

Під час перейменування ідентифікатора нове ім’я має краще відповідати опису ролі ідентифікатора поточного стану системи, ніж стара назва. Вивчення рефакторингу перейменування набирає обертів та йому приділяється все більше уваги в дослідженнях. Добре зрозуміло, що потрібно більш глибоке розуміння того, як природна мова використовується для підтримки розуміння та того, як вона розвивається разом із програмним забезпеченням.

Багато методів для підтримки розуміння покладаються на аналіз ідентифікаторів. Крім того, багато попередніх досліджень досліджували практики іменування, шаблони та те, як це робити шляхом покращення аналізу імен ідентифікаторів. Зокрема існує багато практичних досліджень щодо ідеї налагодження, оцінки та генерації імен ідентифікаторів, що має прямий, позитивний вплив на підходи синтезу програм, які повинні зрозуміти, як розробники описують елементи коду (наприклад, ідентифікатори імен, коментарі до методів тощо).

Щоб створити природну мову, прийнятну розробниками (тобто текст, який оптимізує розробник для повного розуміння), потрібно здійснити аналіз практик іменування ідентифікаторів, особливо якщо їх виконують розробники в реальних сценаріях використання.

Для побудови концепуальної моделі необхідно здійснення аналізу еволюції методів, класів та ідентифікаторів із використанням рефакторингу перейменування до набагато більшої кількості систем і поєднати аналіз на основі їх таксономії з фіксацією повідомлення, щоб дослідити, чому розробники певним чином перейменовують ідентифікатори. Також необхідно розуміння того, чому, наприклад, розробник вирішує звузити або узагальнити значення найменування ідентифікатора. Дані дослідницькі питання в основному визначені для вивчення висновків щодо практики перейменування, а також аналізують рефакторинг перейменування, щоб класифікувати перейменування розробника-практика, використовуючи контекст їхніх зусиль щодо розвитку, а також для надання попередніх результатів дослідження.

Для досягнення вказаних цілей необхідно визначити такі основні моменти.

1. Які типи лексичних змін зазвичай застосовуються за допомогою рефакторингу перейменування, а також яка частка перейменувань є лише зміною одного чи кількох термінів в ідентифікаторі, яка пропорція зміни порядку термінів, зміни у множині, зміни в капіталізації, або додавання роздільників. Для цього дослідницького питання потрібно володіти інформацією про те, як виглядає перейменовування. В свою чергу це дасть уявлення про те, наскільки складними є

перейменування, і дасть певну міру того, наскільки (з точки зору складових слів) ідентифікатори мають тенденцію до зміни.

2. Також необхідно визначити які види семантичних змін відбуваються в термінах, що складають ідентифікатори, коли вони перейменовані та які типи змін значення ідентифікатора є найчастішими. Метою цього запитання буде дослідити, як часто значення ідентифікатора розширюється, звужується, збережено, повністю змінено, додано або видалено з програмної системи. Відповідь на ці запитання допоможе визначити типову поведінку перейменування та допомогти надати більш детальне розуміння діяльності перейменування.

3. Які види граматичних змін відбуваються в термінах, що складають ідентифікатори, коли вони перейменовані. Тобто, коли відбувається зміна тегу частини мови для будь-якого окремого терміну в ідентифікаторі. Відповідь на це запитання допомагає мати уявлення про те, що змінено. Зміна частини мови не завжди означає повну зміну терміну; деякі зміни частини мови пов’язані зі зміною відмінювання початкового терміну. Головне, що потрібно зрозуміти, чи може це допомогти у визначенні моменту семантичної зміни.

4. Якою мірою повідомлення фіксації можна використовувати для контекстуалізації іншого типу семантичних змін рефакторингу перейменування. Якщо використовується тематичне моделювання на корпусі commit-повідомлення, згруповані за категорією семантичних змін, чи можна мати розуміння того, які типи діяльності здійснюють розробники, коли вони вносять різні типи семантичних змін.

5. Які тенденції відображаються у способі перейменування ідентифікаторів та чи можна визначити будь-яку діяльність із розробки, яка корелює з різними типами семантичних змін, внесених до ідентифікаторів. Це запитання буде відображати використовує зібрані дані та використовувати його, щоб допомогти зрозуміти, що викликає різні типи семантичних змін, внесених до ідентифікаторів під час еволюції програмного забезпечення.

Результати дослідження допомагають зрозуміти причини та наслідки рефакторингу перейменування.

Зокрема, це стосується результатів, що мають впливна інструменти підтримки, які намагаються зрозуміти поведінку розробника, коли справа доходить до іменування (ідентифікатора). Це в кінцевому підсумку допоможе збільшити прийняття технології, яка підтримує еволюцію імен ідентифікаторів і створення кращих вказівок щодо того, як слід застосовувати та підтримувати перейменування під час обслуговування.

В даному дослідженні використано двоетапний підхід, щоб відповісти на вище поставлені дослідницькі запитання. Початкова фаза складається з пошуку проектів з відкритим вихідним кодом і виявлення операцій рефакторингу, які відбуваються впродовж історії розробки кожного отриманого проекту. Друга фаза передбачає аналіз виявлених операцій перейменування як засіб розуміння типу підходу, які використовують розробники при зміні імен ідентифікаторів. На рисунку 2.1 зображено концептуальну модель потоку кроків, які беруть участь у процесі оцінки якості найменування ідентифікаторів. Нижче наведено деталі кожного етапу.

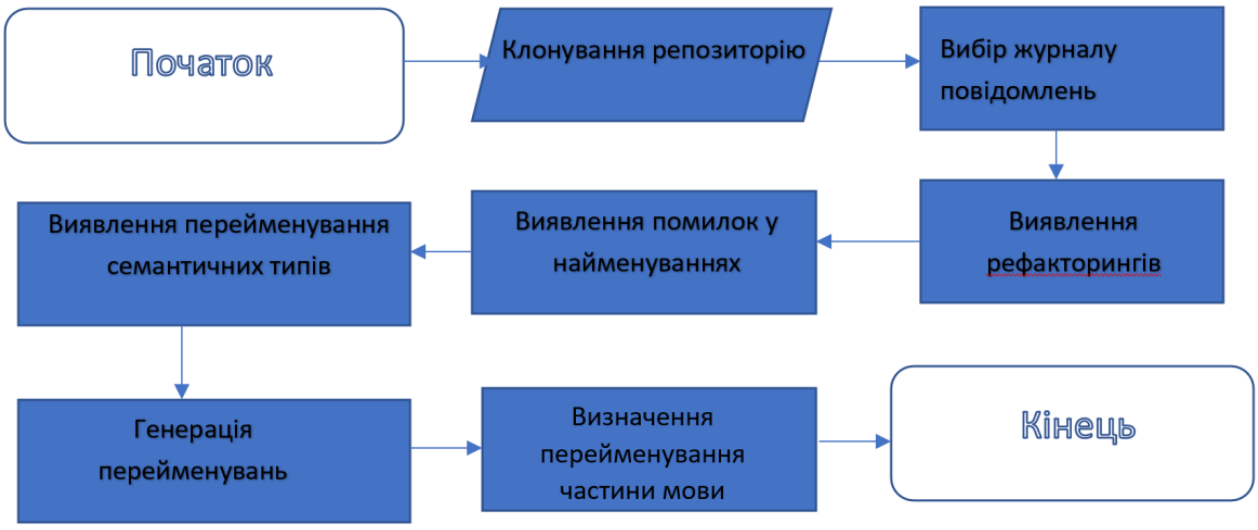


Рисунок 2.1 – Концептуальна модуль процесу оцінки якості найменування ідентифікаторів

Щоб гарантувати, що дане дослідження точно фіксує операції перейменування ідентифікаторів у реальному світі, було вкрай важливо, щоб дослідження базувалося на репрезентативному наборі даних. У цьому

контекстіздійсненого дослідження використовуєтьсясписок проектів, що розміщені у вільному доступі, наприклад у депозитарії GitHub. Для ідентифікації операцій рефакторингу, що виконуються розробниками цих проектів, було використовуємо RefactoringMiner для кожного проекту. Перераховуючи історію комітів кожного проекту, RefactoringMiner може виявити близько 1 000 000 операцій рефакторингу в 3 795 проектах. Із виявлених типів рефакторингу 43,36% операцій рефакторингу пов’язані з операціями перейменування (тобто пакет, клас і перейменування методу). Крім того, розробники зазвичай виконують більше операцій перейменування імен методів у порівнянні з іменами класів або пакетів; з проектами, в середньому містять приблизно 9 пакетів, 47 класів і 122 перейменування методів. Звідси, як не дивно, кількість комітів, пов’язаних із перейменуванням методів, також значно вище. Проте випадки різних типів рефакторингу перейменування в проектах, що містяться в схожому наборі даних, мають подібні розподіли.

Щоб зрозуміти зміни перейменування, внесені розробниками, проводиться таксономія на основі інструментів аналізу оригінальних та перейменованих найменувань ідентифікаторів. Оскільки не має можливості отримати копію з REPENT, потрібно здійснювати аналіз якомога ближче до інструментів та технологій, які доступні в даний момент часу для використання даному дослідженні. Для цього можна використовувати інструментарій NaturalLanguageToolkit (NLTK), який має реалізацію Wordnet, для отримання семантики та частини мови подробиці про імена ідентифікаторів.

Перш ніж виконувати аналіз, виконується попередня обробка оригінальних і нових назв перейменованих ідентифікаторів. Враховуючи, що більшість ідентифікаторів складається з кількох термінів, даний підхід включає розбиття кожної назви на список термінів (тобто токенізацію). Щоб виконати поділ можна використовувати алгоритм розділення Ронін, що реалізований у пакеті Spiral або використовуючи інші засоби.

Семантичний аналіз здійснюється шляхом порівняння кожного терміну з оригіналом та новим ідентифікатором семантичних зв’язків, таких як синоніми,

гіпоніми, гіперніми, антоніми, мероніми, голоніми. Якщо зв’язку не існує, то здійснюється виконання перевірки основи між двома ідентифікаторами та повторного їх порівняння. Для цієї мети доцільно використовувати похідні слова та алгоритми Портера, Ланкастера та Сноубола. Також здійснюється виконання перевірки лематизації за допомогою алгоритму NLTK. Можна використовувати кілька методів формування основи лематизації, щоб спробувати знайти якомога більше співпадінь.

Для кожного виявленого збігу, кожен з яких називається відповідним терміном, також отримується частина мови, що пов’язана з новими та оригінальними термінами. Крім того, використовується їх евристика для визначення семантичних змін. Зокрема, здійснюється ідентифікація перейменування, як зберігати, змінювати, звужувати, розширювати, додавати або видаляти значення від старого до нового ідентифікатора. Також виявлено складність перейменування), перевпорядкування термінів, зміни форматування, доповнення та вилучення термінів.

2.2 Метод оцінки якості найменування ідентифікаторів вихідного коду

Для того, щоб здійснити оцінку якості найменування ідентифікаторів вихідного коду можуть бути використані спеціальні сховища даних, які називаються таблиці ідентифікаторів (таблиці символів).

До складу даних таблиць ідентифікаторів входить певний набір полів. Цим полям відповідає певна кількість різноманітних ідентифікаторів, які можуть міститися у вихідному коді. Кожне таке поле має містити у собі повну інформацію про даний елемент таблиці. Загалом, компілятор має можливість здійснювати свою роботу або з однією або з декількома видами ідентифікаторів. Від того як здійснена реалізація компілятора і встановлюється кількість видів ідентифікаторів. Тобто, можна здійснити організацію роботи різних таблиць

ідентифікаторів для різних модулів вихідного коду програми. Також за тим самим алгоритмом можна працювати і з різними типами елементів вхідної мови.

В розроблюваній таблиці ідентифікаторів для кожного елемента вихідної програми, що зберігається, існує свій склад інформації. Даний в свою чергу, залежить від семантики вхідної мови та, відповідно, типу елемента. До прикладу, в таблицях імен ідентифікаторів може зберігатися інформація такого роду (Таблиця 2.1).

Таблиця 2.1 – Інформація у таблицях ідентифікаторів

Тип	Міститиме інформацію
змінні	ім'я змінної; тип даних змінної; адреса сегменту пам'яті, пов'язаного зі змінною;
константи	ім'я константи (якщо воно є); значення константи; тип даних константи (якщо потрібно);
функції	ім'я функції; кількість та типи формальних аргументів функції; тип результату, що отримується; адресу дзвінка коду функції

Загалом, у прикладі, наведеному у таблиці 2.1 наведено тільки часткова інформація, оскільки кожен компілятор може реалізувати наповненість певних таблиць ідентифікаторів. Крім того, не вся інформація, що зберігається в таблиці ідентифікаторів, заповнюється відразу – він може кілька разів виконувати звернення до даних у таблиці ідентифікаторів на різних фазах компіляції.

Незалежно від реалізації компілятора принцип його роботи з таблицею ідентифікаторів залишається одним і тим же – на різних фазах компіляції компілятор змушений багаторазово звертатися до таблиці для пошуку інформації

та запису нових даних. Як правило, кожен елемент у вихідній програмі однозначно ідентифікується своїм ім'ям. Тому компілятору доводиться часто виконувати пошук необхідного елемента в таблиці ідентифікаторів на його ім'я, тоді як процес заповнення таблиці виконується нечасто – нові ідентифікатори описуються у програмі набагато рідше, ніж використовуються. Звідси можна зробити висновок, що таблиці ідентифікаторів повинні бути організовані таким чином, щоб компілятор мав можливість максимально швидкого пошуку потрібного йому елемента.

Виділяються такі способи побудови та організації таблиць найменувань імен ідентифікаторів:

– сюди входять прості списки – найспростіший спосіб;

– упорядковані списки;

– бінарне дерево – досить ефективний спосіб організації таблиць;

– хеш-адресація з рехешуванням – досить ефективний спосіб;

– хеш-адресація за методом ланцюжків – досить ефективний спосіб;

– комбінація хеш-адресації зі списком або бінарним деревом – один із найефективніших способів організації таблиць.

Для того, щоб здійснити побудову таблиці використовується додавання елементів в тому порядку, як вони надходили. Через це такий спосіб дана таблиця ідентифікаторів являє собою масив інформації чи невпорядкованим списком. Кожне поле даної таблиці буде містити дані про відповідний елемент таблиці.

Щоб знайти у даній таблиці необхідний елемент потрібно послідовно порівнювати кожен елемент таблиці із тим елементом, що потрібно знайти. Це пошук працює до тих пір, поки не знайдеться потрібний елемент. Якщо позначити час, який потрібно для додавання елемента через (X_d), і він не буде залежати від кількості елементів у таблиці (K), то тоді для того, щоб знайти елемент, який позначається через (Xn) у невпорядкованій таблиці, яка містить K елементів потрібно в середньому $P/2$ порівнянь.

Потрібно зазначити, що вказаний спосіб побудови таблиць найменувань ідентифікаторів є малоефективним. Це відбувається тому, що пошук у таблиці

ідентифікаторів є операцією, що здійснюється компілятором найчастіше. В свою чергу, зрозумілим є той факт, що будь-яка програма містить досить велику кількість різноманітних ідентифікаторів.

Якщо здійснювати пошук серед впорядкованих чи відсортованих певним чином елементів за якимось алгоритмом чи правилом елементів, то ефективність такого пошуку буде збільшена в рази. Можливий варіант розміщення елементів таблиці в алфавітному порядку.

Ефективним методом пошуку в упорядкованому списку *K* елементів є бінарний або логарифмічний пошук, який здійснюється наступним чином.

Ідентифікатор, який потрібно знайти, порівнюється з елементом *(K+1)/2* у середині таблиці. Якщо цей елемент не є шуканим, то потрібно переглянути тільки блок елементів, пронумерованих від 1 до *(K+1)/2–1*, або блок елементів від *(K+1)/2+1* до *K* залежно від того, меншим чи більшим є шуканий елемент від того, з яким його порівняли. Далі увесь цей пошук продовжується шляхом повторення над тим блоком, що потрібний. Даний блок менший у два рази. Даний процес триває доти, допоки потрібний елемент не відшукається чи алгоритм не дійде до чергового блоку, що містить один або два елементи (з якими вже можна виконати пряме порівняння елемента, що шукається).

Оскільки з кожним разом кількість блоків, що мають потрібний елемент, скорочується наполовину, то в такому випадку максимальна кількість порівнянь обраховується за формулою (1):

$$1 + \log_2 K \qquad (1)$$

А час пошуку елемента в таблиці найменувань ідентифікаторів можна оцінити за формулою (2)

$$X_n = O \cdot \log_2 K \qquad (2)$$

Якщо припустити, що *K=128,* то видно, що бінарний пошук вимагає щонайменше 8 порівнянь, а пошук у невпорядкованій таблиці – у середньому 64 порівняння. Такий спосіб називають «методом бінарного пошуку», бо за кожним разом кількість розглянутої інформації стає меншою у два рази. Оскільки час, витрачений на пошук потрібного елемента в масиві, має логарифмічну залежність від загальної кількості елементів у ньому, тому даний спосіб називають «логарифмічним».

Даний метод має недолік, а саме вимагає впорядкованості таблиці найменувань ідентифікаторів.

Впорядкованість масиву інформації відіграє пряму роль у пошуку, оскільки час заповнення даного масиву напряму залежить від числа елементів у ньому. Приблизний вигляд таблиці найменувань ідентифікаторів, зазвичай, формується ще до повного її заповнення. Саме тому умова впорядкованості відіграє значну роль та має виконуватись на всіх етапах звернення до неї. Алгоритм прямого упорядкованого включення елементів доцільно використовувати для конструювання даної таблиці самостійно без включення інших методів.

Коли здійснюється додавання в таблицю кожного нового елемента спочатку потрібно визначити місце, де буде розміщуватись новий елемент, а потім виконати перенесення частини інформації в таблиці, якщо елемент додається в самий її кінець.

Таким чином, якщо здійснюється логарифмічний пошук у таблиці найменувань ідентифікаторів, то здійснюється суттєве зменшення часу пошуку потрібного елемента. Це відбувається завдяки збільшення часу на розміщення в таблицю нового елемента. Даний метод є досить ефективним порівняно із методом організації таблиці, при якому елементи є невпорядкованими. Це відбувається тому, що додавання нових елементів до таблиці ідентифікаторів відбувається значно рідше, ніж звернення до них.

Якщо при організації таблиці відмовитись від структури масиву даних, що є безперервним, то час пошуку шуканого елемента в таблиці ідентифікаторів скоротиться. При цьому час, необхідний для її заповнення не збільшиться.

Також доцільно використовувати спосіб побудови таблиць, де сама таблиця найменувань ідентифікаторів визначається у формі бінарного дерева, де вузли є елементами таблиці. Перший елемент, який зустрічається під час заповнення таблиці називається кореневим вузлом. Оскільки, кожна вершина цього дерева може мати не більше двох гілок (не більше двох нижчих вершин), то таке дерево називають бінарним. Щоб краще орієнтуватись визначають дві гілки «праву» та «ліву». Це забезпечує нижчу помилковість.

Вважатимемо, що алгоритм заповнення бінарного дерева працює з потоком вхідних даних, що містить ідентифікатори (у компіляторі цей потік даних породжується в процесі аналізу тексту вихідної програми). Перший ідентифікатор, як уже було сказано, міститься у вершину дерева. Усі подальші ідентифікатори потрапляють у дерево за наступним алгоритмом.

1. Здійснити вибір із вхідного потоку даних чергового ідентифікатора. За відсутності чергового ідентифікатора вважати побудову дерева завершеною.

2. Коренева вершина має стати поточним вузлом дерева.

3. Здійснити порівняння чергового ідентифікатор з ідентифікатором, який розміщується у поточному вузлі дерева.

4. У випадку, якщо наступний ідентифікатор є меншим, то потрібно перейти до наступного, якщо рівний, то надіслати повідомлення про помилку та зупинити виконання алгоритму. При цьому слід зауважити, щоне повинно бути двох однакових ідентифікаторів. Якщо знаходяться два однакові ідентифікатори, то потрібно перейти до пункту 7.

5. У ситуації існування лівої вершини у поточному вузлі, то потрібно зробити її поточним вузлом, а далі здійснити повернення до пункту 3, у протилежному випадку потрібно перейти до пункту 6.

6. На даному етапі необхідно здійснити створення нової вершини. Далі помістити в неї чергове найменування ідентифікатора, а потім зробити цю нову вершину лівою вершиною поточного вузла та повернутися до пункту 1.

7. У випадку існування правої вершина у поточному вузлі існує, то потрібно зробити її поточним вузлом, а далі повернутись до пункту 3. В протилежному випадку здійснити перехід до пункту 8.

8. На даному етапі здійснити створення нової вершину, а також помістити в неї черговий ідентифікатор. Створену вершину зробити правою вершиною поточного вузла, а далі здійснити повернення до пункту 1.

Для прикладу, можна проілюструвати створення бінарного дерева за вказаним алгоритмом. Нехай у на є певна кількість найменувань ідентифікаторів. Розглянемо як приклад послідовність ідентифікаторів F, BC, GA, M22, D1, E, A12. На рисунку 2.2 показано наявний приклад усього етапу формування бінарного дерева для вказаної групи найменувань ідентифікаторів.

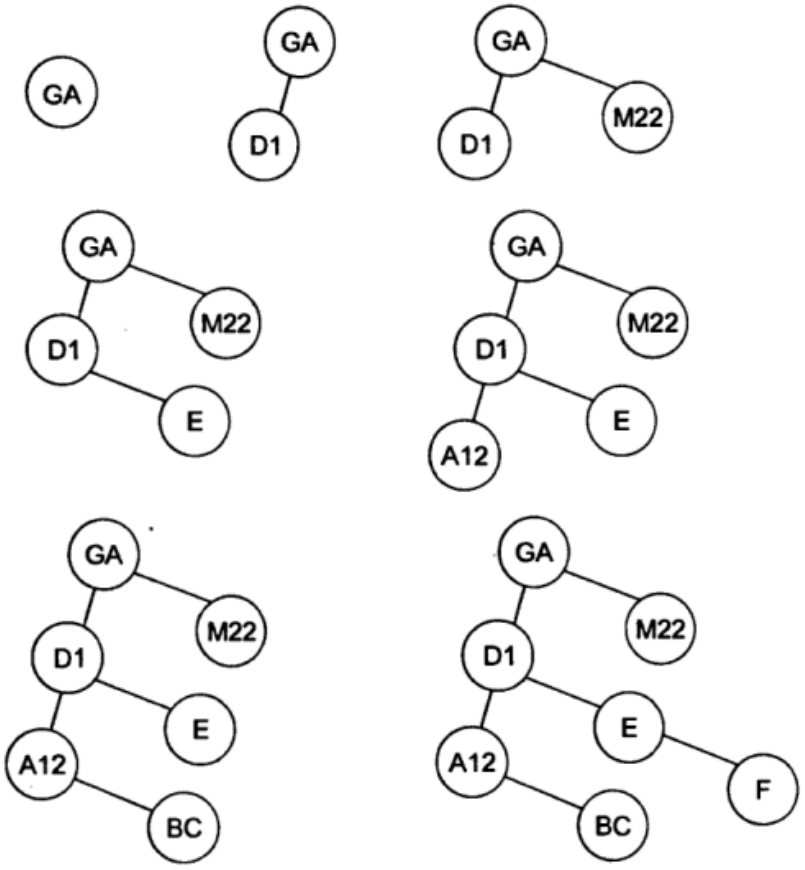


Рисунок 2.2 – Побудоване бінарне дерево

Для пошуку потрібного елемента у дереві можна використовувати алгоритм, який дещо подібний до алгоритму наповнення дерева.

1. На першому етапі необхідно кореневу вершину зробити поточним вузлом бінарного дерева.

2. На даному кроці здійснюється порівняння потрібного ідентифікатор з тим ідентифікатором, який розміщується у поточному вузлі дерева.

4. У випадку, якщо ідентифікатори співпадають, то ідентифікатор вважається знайденим. Отже, тоді алгоритм завершується. У протилежному випадку потрібно здійснити перехід до п’ятого кроку.

5. У випадку, коли наступне найменування ідентифікатора менше, то здійснити перехід до шостого етапу. У протилежному випадку здійснити перехід до сьомого кроку.

6. На даному етапі виявнити чи існує ліва вершина у поточного вузла. Якщо існує, то зробити її поточним вузлом і повернутись до другого кроку. В протилежному випадку, якщо ідентифікатор, що шукається не знайдений, то алгоритм закінчується.

7. У випадку, коли права вершина існує у поточному вузлі, то потрібно зробити її поточним вузлом. Далі здійснити повернення до другого кроку. В протилежному випадку вважати, що шуканий ідентифікатор не знайдений, тому алгоритм завершується.

Як приклад можна навести пошук найменування ідентифікатора A12. Будемо вважати, що дерево повністю сформоване. Для цього використовуємо кореневу вершину, яка стає поточним вузлом. Далі здійснюємо порівняння ідентифікаторів GA та A12. Оскільки ідентифікатор, який шукається менший, то ліва вершина стає поточним вузлом стає D1. На наступному кроці знову здійснюємо порівняння найменувань ідентифікаторів. І як видно, той ідентифікатор, що шукається є меншим. Тому ліва вершина стає поточним вузлом A12. Під час здійснення наступного порівняння буде знайдене потрібне ім'я токену ідентифікатора.

У випадку здійснення пошуку відсутнього найменування ідентифікатора, наприклад, A11, пошук знову буде здійснено від кореневої вершини. Далі відбувається порівняння ідентифікаторі GA та A11. Оскільки, шуканий

ідентифікатор є меншим, то ліва вершина D1 стає поточним вузлом. Далі потрібно порівняти найменування ідентифікаторі. Якщо шуканий ідентифікатор менший, то ліва вершина A12 стає поточним вузлом. Шуканий ідентифікатор менший, але ліва вершина-нащадок у вузла A12 відсутня, тому в даному випадку шуканий ідентифікатор не знайдено.

Для цього способу кількість необхідних порівнянь і форма дерева, що вийшла залежить від порядку надходження ідентифікаторів. Наприклад, якщо в розглянутому вище прикладі замість послідовності ідентифікаторів F, BC, GA, M22, D1, E, A12 взяти якусь іншу будь-яку послідовність, то отримане дерево буде мати зовсім інший вигляд. А якщо, наприклад, як приклад взяти послідовність найменувань ідентифікаторів A, B, C, D, E, F, то в такому випадку дерево виродиться в упорядкований зв'язаний список, що буде односпрямованим.

Дана недосконаліть є недоліком даного способу побудови таблиць найменувань ідентифікаторів. Разом із цим ще одним не зовсім вдалим моментом при побудові дерева є необхідність роботи з динамічним виділенням пам'яті.

Побудоване бінарне дерево буде невиродженим, якщо допустити, що послідовність ідентифікаторів у вихідній програмі є статистично невпорядкованою. В такому випадку буде (*Хд*) – середній час, що виділяється на заповнення дерева. Пошук елемента в даному дереві позначено як (*Хп*). Ці величини можна оцінити наступним чином (3, 4):

$$Xд = N \cdot O \log_2 K \tag{3}$$

$$Xn = O \cdot \log_2 K \tag{4}$$

Отже, можна зробити висновок, що в принципі спосіб побудови таблиць найменувань ідентифікаторів є досить вдалим механізмом. На його основі будуються різноманітні компілятори. Якщо потрібно здійснити побудову для декількох різних дерев, що містять найменування ідентифікаторів різних типів та довжини, то даний спосіб себе повністю оправдовує.

Найкращого результату, що можна досягти шляхом використання різноманітних засобів побудови таблиць встановлюється завдяки логарифмічній залежності часу пошуку, а також часу заповнення таблиці найменувань ідентифікаторів. Але, якщо брати до уваги реальну розробку програмного забезпечення, то у програмних системах буде дуже велика кількість найменувань ідентифікаторів і логарифмічної залежності часу пошуку від їх числа буде замало. Тому доцільно включати такі способи, які реалізуються на основі використання хеш-функцій та хеш-адресації.

Хеш-функцією *F* називається деяке відображення множини вхідних елементів *R* на множину цілих невід'ємних чисел $Z: F(r) = n,\ r\ \in R,\ n \in Z.$

Термін «хеш- функція» походить від англійського терміну «hashfunction» (hash– «заважати», «змішувати», «плутати»).

Область визначення хеш-функції – це множина допустимих вхідних елементів *K*. Підмножина *N* з множини цілих невід'ємних чисел *T*, що містить усі можливі значення, що повертаються функцією *H* є множиною значень хеш-функції *H*.

Хешування це дії, що спрямовані на відображення та визначення хеш-функції у множині значень. Досить часто у наукові літературі замість терміну «хешування» використовуються терміни «рандомізація», «переупорядкування».

Під час роботи із таблицею найменувань ідентифікаторів хеш-функція має на меті виконання відображення найменування ідентифікаторів на множину цілих невід'ємних чисел. Тоді для хеш-функцій областю їх визначення буде множина усіх можливих найменувань ідентифікаторів.

Сутністю використання та роботи хеш-адресації є використанні значення, яке повертається хеш-функцією. Результатом є адреса комірки з деякого масиву даних. В такому випадку розмір масиву даних має відповідати області значень хеш-функції, що використовується. Саме тому при реальному програмуванні і роботі із великими масивами ідентифікаторів розмір доступного адресного простору комп'ютера має бути досить значним і область значень хеш-функцій ніяким чином не повинна його перевищувати. Метод організації таблиць

ідентифікаторів, заснований на використанні хеш адресації, полягає у розміщенні кожного елемента таблиці в комірці, адреса якої повертає хеш-функцію, обчислену для цього елемента. Тоді в ідеальному випадку для розміщення будь-якого елемента в таблиці ідентифікаторів достатньо лише обчислити його хеш-функцію і звернутися до потрібної комірки масиву даних. Для пошуку елемента у таблиці необхідно обчислити хеш-функцію для шуканого елемента та перевірити, чи не є заданий нею осередок масиву порожній (якщо він не порожній – елемент знайдений, якщо порожній – не знайдено).

На рисунку 2.3 проілюстровано метод організації таблиць ідентифікаторів з використанням хеш-адресації. Трьом різним ідентифікаторам *A1*, *A2*, *A3* відповідають на малюнку три значення хеш-функцій *n1*, *n2*, *n3*. У адресах комірки *n1*, *n2*, *n3*, міститься інформація про ідентифікатори *A1*, *A2*, *A3*. При пошуку ідентифікатора *A3* обчислюється значення адреси *n3* і вибираються дані з відповідної комірки таблиці.

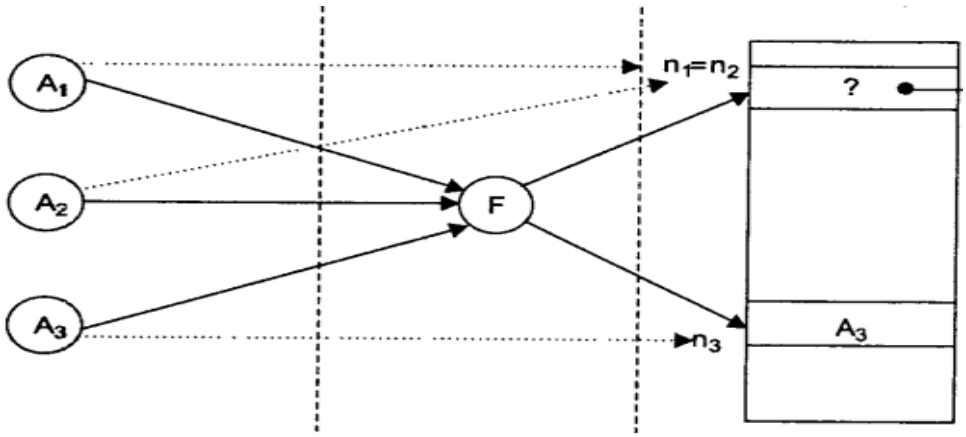


Рисунок 2.3 – Невизначеності при використанні хеш-адресації

Цей спосіб вважається достатньо ефективним, оскільки час розміщення елемента в таблиці і час його пошуку визначаються тільки часом, який витрачається на обчислення хеш-функцій. Цей загальний час є меншим для того, щоб здійснити багаторазові порівняння елементів таблиці імен ідентифікаторів.

Також слід зазначити, що даний спосіб має свої недоліки. По-перше: використання об’єму пам'яті під таблицю ідентифікаторів є неефективним,

оскільки розмір масиву для її зберігання повинен відповідати області значень хеш-функції. Разом з тим в реальних умовах зберігання в таблиці ідентифікаторів він може бути значно меншим. По-друге: потрібно розумно здійснювати вибір хеш-функції.

Хеш-функцій існує дуже багато, і їх багатоваріантність теж досить велика. Для того, щоб отримати результат від виконання хеш-функції, тобто «хешування», потрібно здійснити над ланцюжком символів певні прості арифметичні та логічні операції. Однією із найпростіших для символів хеш-функцій є код внутрішнього подання літер символів. Дану хеш-функцію також використовують і для ланцюжка символів, при цьому здійснюючи вибір у ланцюжку першого символу.

Хеш-функція, запропонована вище, є незадовільною: якщо використовувати цю хеш-функцію, то однозначно будуть виникати нові проблеми, зокрема може бути ситуація, коли двом різним найменуванням ідентифікаторів, що починаються з однієї і тієї ж літери, буде відповідати одне і те ж значення хеш-функції. За таких умов при здійсненні хеш-адресації в одне поле таблиці найменувань ідентифікаторів і за однією й тією ж адресою мають розміщуватись два різні ідентифікатори. А це явно є неможливим. Ситуація, при якій двом або більше ідентифікаторам відповідає одне й те саме значення функції називається колізією. На рисунку 2.3 показано виникнення колізії, тобто різним ідентифікаторам *A1* і *A2* відповідають два значення хеш-функції, що збігаються *n1=n2* [40].

Звісно, якщо виникають колізії, то такі хеш-функції не можуть використовуватись для здійснення хеш-адресації в таблиці найменувань ідентифікаторів. Навіть один випадок отримання колізії у всій множині найменувань ідентифікаторів є забороною на використання даною функцією.

Для того, щоб не виникало ніяких колізій та невизначеностей, потрібно, щоб хеш-функція була взаємно однозначною. В такому випадку будь-яким двом довільним елементам з області визначення хеш-функції завжди відповідатимуть два різних її значення. Якщо розглядати таку можливість у теоретичному аспекті,

то виходить, що таку хеш-функцію для найменувань ідентифікаторів створити можливо. Тоді будуть задіяні для області визначення хеш-функції всі можливі імена ідентифікаторів, для області значень хеш-функції цілі невід'ємні числа, що є нескінченними множинами.

На практиці це зробити досить складно, оскільки в реальності область значень будь-якої хеш-функції має обмеження розміром доступного адресного обсягу. Здійснити організацію взаємно однозначного відображення нескінченної множини на кінцеве неможливо. Навіть якщо врахувати той факт, що довжина рядка ідентифікатора в реальних компіляторах, також зазвичай обмежена – тобто лежить в межах від 32 до 128 символів (це означає, що й область визначення хеш-функції кінцева). Навіть у такому випадку кількість всіх можливих ідентифікаторів все одно дорівнює більшій кількості допустимих адрес у сучасних комп'ютерах. Отже, як бачимо, у реальних практичних умовах, створити взаємно однозначну хеш-функцію неможливо.

Таким чином, неможливо уникнути колізій. Одним із способів вирішення проблеми колізій є метод рехешування (розміщення). Відповідно до цього методу, якщо для елемента A адреса $h(A)$, буде обчислюватись за допомогою хеш-функції, яка демонструє комірку, яка вже зайнята, то тоді є необхідність в обчисленні значення функції $n_1=h_1(A)$. Після чого здійснити перевірку зайнятості поля за адресою n_1. Якщо поле зайняте, то в такому випадку здійснюється обчислення значення $h_2(A)$. Такі дії будуть виконуватись аж до тих пір, допоки не буде знайдено вільної комірки. Як варіант можливий ще збіг чергового значення $h_i(A)$ із $h(A)$. Якщо таблиця найменувань ідентифікаторів вже заповнена, і в ній відсутнє порожнє місце, то в такому випадку виникає оголошення про те, що є помилка розміщення найменування ідентифікатора у таблиці ідентифікаторів. Особливістю такого методу є те, що спочатку таблиця ідентифікаторів має бути заповнена такою інформацією, яка дозволяє розуміти той факт, що її поля порожні, тобто не містять даних. Наприклад, якщо використовуються покажчики для зберігання імен ідентифікаторів, то таблицю треба попередньо заповнити порожніми покажчиками.

Здійснити організацію таблиці найменувань ідентифікаторів можна за таким алгоритмом розміщення елемента.

1. На першому кроці здійснюється обчислення значення хеш-функції $p = s(B)$ нового елемента B.

2. У випадку, якщо поле за адресою n порожнє, то потрібно помістити до неї елемент B після чого здійснити завершення алгоритму. У протилежному випадку при $i:=1$ потрібно здійснити перехід до третього кроку.

3. На наступному етапі необхідно здійснити обчислення $p_i=s_i(B)$. У випадку виявлення порожньої комірки за адресою p_i, то тоді потрібно помістити в неї елемент A. Після цього необхідно здійснити завершення алгоритму. В протилежному випадку перейти до четвертого кроку.

4. При ситуації, коли $p=p_i$, то необхідно повідомити про наявність помилки та після цього здійснити завершення алгоритму. У протилежному випадку при $i:=i+1$ здійснити повернення до третього кроку.

Якщо здійснення організації таблиці найменувань ідентифікаторів відбувається таким чином, то в такому випадку алгоритм пошуку елемента B в даній таблиці найменувань ідентифікаторів такий:

1. Обчислити значення хеш-функції $p=h(B)$ нового елемента B.

2. Якщо поле за адресою p порожнє, то елемент не знайдено, алгоритм завершено, інакше порівняти ім'я елемента в комірці p з іменем елемента A. Якщо вони збігаються, то елемент знайдено та алгоритм завершено, *інакше* $i:=1$ і потрібно перейти до кроку 3.

3. Обчислити $pi= pi(B)$. Якщо осередок за адресою n_i порожній або $n=n_i$, то елемент не знайдено і алгоритм завершено, інакше потрібно порівняти ім'я елемента в комірці p_i з ім'ям шуканого елемента B. Якщо вони збігаються, то елемент знайдено та алгоритм завершено, інакше $i:=i+1$ і повторити пункт 3.

4. У результаті після розміщення в таблиці для пошуку ідентифікатора B_1 буде потрібно 1 порівняння, для B_2 – 2 порівняння, для B_3 – 2 порівняння, для B_4 – 1 порівняння та для B_5 – 5 порівнянь.

Таким чином існує пряма залежність між наповненістю таблиці та кількістю операцій, що необхідні для пошуку чи розміщення в таблиці елемента. Явно видно, що обчислення функції $p_i(B)$ мають здійснюватись однаково на таких етапах як розміщення, а також пошук відповідного елемента. Важливим моментом тут є саме обчислення функції $p_i(B)$.

Найпростішим методом обчислення функції $p_i(B)$ є її організація у вигляді $p_i(B)=(p(B)+p_i) mod\ N_m$, де p_i – деяке число, що обчислюється, а P_m – максимальне кількість елементів у таблиці ідентифікаторів. У свою чергу, найпростішим підходом тут покласти $p_i=i$. Тоді одержуємо формулу $p_i(B)=(p(B)+i)\ mod\ N_m$. В цьому випадку при збігу значень хеш-функції для будь-яких елементів пошук вільного поля в таблиці починається послідовно від поточної позиції, заданої хеш-функцією $p(B)$.

Цей метод не можна визнати особливо успішним, так як при збігу хеш-адрес елементи в таблиці починають групуватися навколо них, що збільшує кількість необхідних порівнянь при пошуку та розміщенні. Середній час пошуку елемента в такій таблиці залежно від кількості операцій порівняння можна оцінити так (5):

$$T_h = O\ ((1 - Lf / 2) / (1 - Lf)) \qquad (5)$$

Тут Lf – (loadfactor) ступінь заповненості таблиці ідентифікаторів – відношення числа зайнятих полів таблиці до максимально допустимого числа елементів у ній: $Lf = N/N_m$.

Розглянемо як приклад ряд послідовних осередків таблиці $n1, n2, n3, n4, n5$ та ряд ідентифікаторів, які треба розмістити в ній: $B1, B2, B3, B4, B5$.

$$s(B_1) = s(B_2) = s(B_5) = p_1,\ s(B_3) = p_2,\ s(B_4) = p_4 \qquad (6)$$

Послідовність розміщення ідентифікаторів в таблиці при використанні найпростішого методу рехешування показана на рисунку 2.4

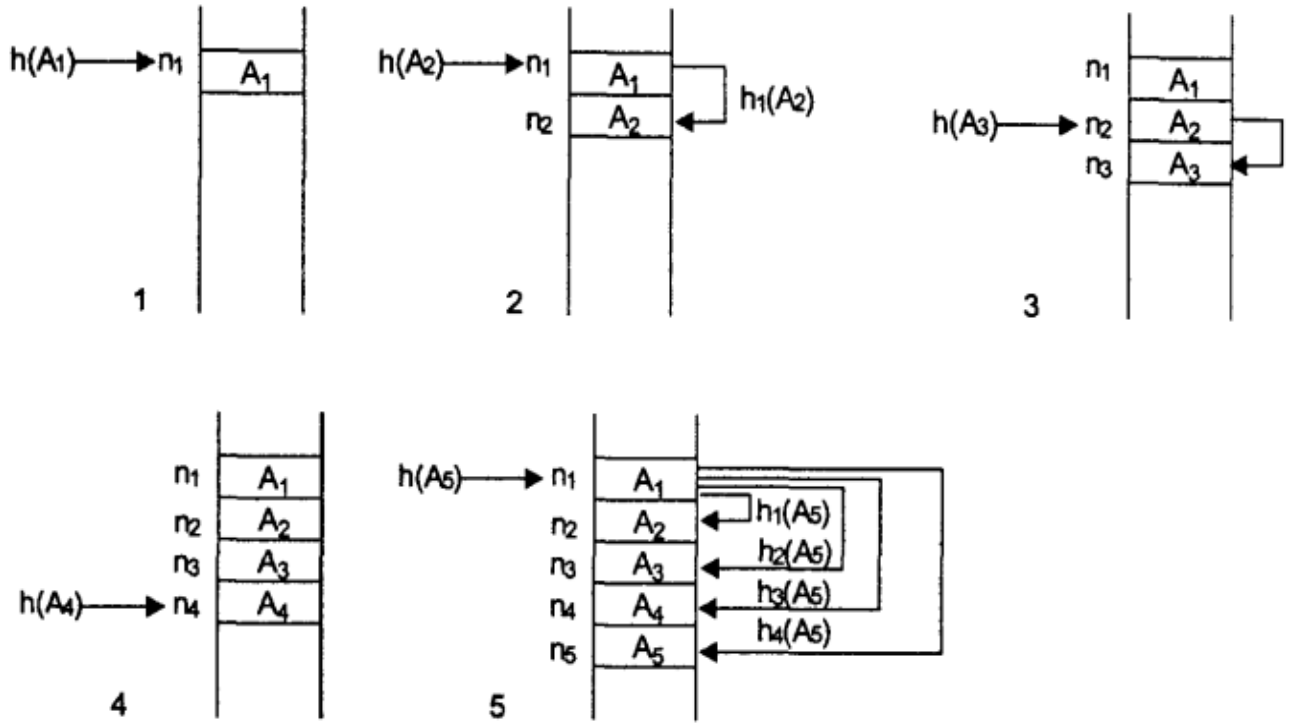


Рисунок 2.4 – Заповнення таблиці ідентифікаторів при використання простого рехешування

Навіть такий примітивний метод рехешування є досить ефективним засобом організації таблиць ідентифікаторів при неповному заповненні таблиці. Маючи, наприклад, навіть заповнену на 90 % таблицю для 1024 ідентифікаторів, у середньому необхідно виконати 5,5 порівнянь для пошуку одного ідентифікатора, у той час як навіть логарифмічний пошук дає в середньому від 9 до 10 порівнянь. Порівняльна ефективність методу буде ще вищою при зростанні числа ідентифікаторів та зниженні заповненості таблиці.

Середній час на розміщення одного елемента в таблиці та на пошук елемента в таблиці можна знизити, якщо застосувати досконаліший спосіб рехешування. Одним із таких методів є використання як p_i для функції (7)

$$s_i(A) = (s(A) + p_i) \bmod N_m \qquad (7)$$

після довжини псевдовипадкових цілих чисел $p_1, p_2, ..., p_k$. При хорошому виборі генератора псевдовипадкових чисел довжина послідовності k буде $k=N_m$. Тоді середній час за позовом одного елемента в таблиці можна оцінити так (8):

$$En = O((1 / Lf) \cdot \log2(1 - Lf)) \quad (8)$$

Існують інші методи організації функцій рехешування $S_i(B)$, засновані на квадратичних обчисленнях або, наприклад, на обчисленні за формулою (9):

$$S_i(A) = (h(A) \cdot i) \bmod N_m, \text{ якщо } N_m \text{ – просте число} \quad (9)$$

У цілому рехешування дозволяє досягти хороших результатів для ефективного пошуку елемента в таблиці (кращих, ніж бінарний) пошук і бінарне дерево), але ефективність методу сильно залежить від заповненості таблиці ідентифікаторів і якості використовуваної хеш-функції – чим рідше виникають колізії, тим вище ефективність методу. Вимога неповного заповнення таблиці веде до неефективне використання обсягу доступної пам'яті.

Неповне заповнення таблиці ідентифікаторів під час застосування хеш-функції веде до неефективного використання всього обсягу пам'яті, доступного компілятору. Причому обсяг пам'яті, що не використовується, буде тим вищим, чим більше інформації зберігається для кожного ідентифікатора. Цього недоліку можна уникнути, якщо доповнити таблицю ідентифікаторів деякою проміжною хеш-таблицею.

У осередках хеш-таблиці може зберігатися або порожнє значення, або значення вказівника на деяку область пам'яті з основної таблиці ідентифікаторів. Тоді хеш-функція обчислює адресу, яким відбувається звернення спочатку до хеш-таблиці, а потім вже через неї за знайденою адресою – до самої таблиці ідентифікаторів. Якщо відповідна комірка таблиці ідентифікаторів порожня, то комірка хеш-таблиці міститиме порожнє значення. Тоді зовсім не обов'язково мати в самій таблиці ідентифікаторів комірку для кожного можливого значення хеш-функції – таблицю можна зробити динамічною так, щоб її обсяг зростав у міру заповнення (спочатку таблиця ідентифікаторів не містить жодного осередку, а всі осередки хеш-таблиці мають порожнє значення).

Такий підхід дозволяє досягти двох позитивних результатів: по-перше, ні необхідності заповнювати порожніми значеннями таблицю ідентифікаторів – це можна зробити тільки для хеш-таблиці; по-друге, кожному ідентифікатору буде відповідати суворо одна комірка в таблиці ідентифікаторів (у ній не буде порожніх клітин, що не використовуються). Порожні осередки в такому випадку будуть тільки в хеш-таблиці, і обсяг пам'яті, що не використовується, не залежатиме від обсягу інформації, що зберігається для кожного ідентифікатора – для кожного значення хеш-функції буде витрачатися тільки пам'ять, необхідна для зберігання одного покажчика на основну таблицю ідентифікаторів.

На основі цієї схеми можна реалізувати ще один спосіб організації таблиць ідентифікаторів за допомогою хеш-функції, званий методом ланцюжків. Для методу ланцюжків у таблицю ідентифікаторів для кожного елемента додається ще одне поле, в якому може міститися посилання на будь-який елемент таблиці. Спочатку це поле завжди порожнє (не вказує ні на що). Також для цього методу необхідно мати одну спеціальну змінну, яка завжди вказує на перший вільний осередок основної таблиці ідентифікаторів.

Спосіб побудови таблиць ідентифікаторів із застосуванням ланцюжків працює наступним чином:

1. У всі поля хеш-таблиці помістити порожнє значення, таблиця ідентифікаторів не повинна містити жодного осередку, змінна *FrPt* (покажчик першого вільного осередку) вказує на початок таблиці ідентифікаторів*; i:=1*.

2. Обчислити значення хеш функції n_i для нового елемента A_i. Якщо комірка хеш-таблиці за адресою n_i порожня, то помістити в неї значення змінної *FrPt* і перейти до кроку 5; інакше перейти до кроку 3.

3. Покласти *j:=1*, вибрати з хеш-таблиці адресу осередку таблиці ідентифікаторів m_j і перейти до кроку 4.

4. Для комірки таблиці ідентифікаторів на адресу m_j перевірити значення поля посилання.

Якщо воно порожнє, то записати адресу зі змінної *FrPt* і перейти до кроку 5; інакше *j:=j+1*, вибрати з поля посилання адресу m_j та повторити крок 4.

5. Додати в таблицю ідентифікаторів новий осередок, записати в неї інформацію для елемента A_i (поле посилання має бути порожнім), у змінну *FrPt* помістити адресу за кінцем доданої комірки. Якщо більше немає ідентифікаторів, які треба розмістити у таблиці, виконання алгоритму закінчено, інакше *i:=i +1* і перейти до кроку 2.

Пошук елемента в таблиці ідентифікаторів, організованої таким чином, будевиконуватися за таким алгоритмом:

1. Обчислити значення хеш-функції n для елемента *A*. Якщо клітинка хеш-таблиці за адресою n порожня, то елемент не знайдений і алгоритм завершений, інакше покласти *j:=1*, вибрати з хеш-таблиці адресу осередку таблиці ідентифікаторів $m_j = n$.

2. Порівняти ім'я елемента в комірці таблиці ідентифікаторів на адресу m_j з ім'ям шуканого елемента *A*. Якщо вони збігаються, то шуканий елемент знайдено і алгоритм завершено, інакше перейти до кроку 3.

3. Перевірити значення поля посилання в осередку таблиці ідентифікаторів на адресу m_j.

Якщо воно порожнє, то елемент, що шукається, не знайдений і алгоритм завершений; інакше *j:=j+1*, ви брати з поля посилання адресу m_j і перейти до пункту 2.

За такої організації таблиць ідентифікаторів у разі виникнення колізії алгоритм розміщує елементи в осередках таблиці, пов'язуючи їх один з одним послідовник, але через поле посилання. При цьому елементи не можуть потрапляти до осередків з адресами, які потім збігатимуться зі значеннями хеш-функції. Таким чином, додаткові колізії не виникають. У результаті таблиці з'являються своєрідні ланцюжки пов'язаних елементів,

Вказаний метод можна вважати досить гарним при здійсненні організації таблиць найменувань ідентифікаторів. Це пояснюється тим, що при обчисленні хеш-функцій хоч і виникають певні колізії, однак вони незначним чином впливають на середній час обчислення хеш-функцій, протягом якого здійснюється розміщення одного елемента, а також пошук певного елемента у таблиці.

2.3 Алгоритм граматичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи

Для встановлення розуміння взаємозв’язку між групами ідентифікаторів пропонується використовувати граматичні шаблони, тобто послідовність тегів частини мови (також відомих як анотації), призначених окремим словам у межах вихідного коду.

Ім'я ідентифікатора відоме як граматичний шаблон. Наприклад, ідентифікатор під назвою GetUserToken призначається граматичний шаблон шляхом розбиття ідентифікатора на три складові слова: Get, Use (Користувач) і Token (маркер). Розділена послідовність (Отримати ім’я працівника) потім передається через тег частини мови для визначення граматичного шаблону: дієслово-іменник-іменник. Цей граматичний зразок є не є унікальним для цього ідентифікатора; його поділяють на багато інших потенційних ідентифікаторів, які використовують подібні умови. Таким чином, граматичний шаблон може бути використаний для з'єднання ідентифікаторів, які містять різні терміни; GetEmployeeName, SetUserId і WriteEmployeeAddress мають однаковий граматичний шаблон, і хоча вони не виражають однакову семантику, їхні граматичні моделі вказують на схожість в їх семантиці. Зокрема, дієслово (get, set, write), яке застосовується до іменника (ім’я, ідентифікатор, адреса), який має визначену роль або контекст (працівник, користувач).

Лексичний аналізатор, що здійснює встановлення розуміння взаємозв’язку між найменування ідентифікаторів здійснює читання потоку символів, які входять у вихідний код. Даний аналізатор здійснює групування цих символів у певні граматичні послідовності (лексеми).

Лексема (логічна одиниця мови) – це структурна одиниця мови, яка складається з елементарних символів мови та не містить у своєму складі інших структурних одиниць мови. Лексемами мов програмування є ідентифікатори, константи, ключові слова мови, знаки операцій тощо.

На вхід лексичного аналізатора надходить текст вихідної програми, а вихідна інформація передається для подальшої обробки компілятором на етапі синтаксичного аналізу та розбору. Для кожної лексеми аналізатор будує вихідний токен (token) виду ім'я_токену, значення_атрибуту.

Він передається наступній фазі, синтаксичному аналізу. Перший компонент токена, імя_токену, є абстрактним символом, що використовується під час синтаксичного аналізу, а другий компонент, значення атрибуту, вказує на запис у таблиці символів, що відповідає даному токену. Інформація із запису в таблиці символів необхідна для семантичного аналізу та генерації коду.

Припустимо, наприклад, що вихідна програма містить інструкцію присвоєння *position = initial + rate * 60.*

Символи у цьому присвоєнні можуть бути згруповані у наступні лексеми та відображені наступні токени, що передаються синтаксичному аналізатору.

1. *position* є лексемою, яка може відображатися в токен *(id,1),* де *id* – абстрактний символ, що позначає ідентифікатор, а 1 вказує запис таблиці символів для позиції. Запис таблиці символів для деякого ідентифікатор зберігає інформацію про нього, таку як його ім'я та тип.

2. Символ присвоєння = є лексемою, яка відображається в токен *(=).*

Оскільки цей токен не вимагає значення атрибуту, другий компонент цього токена опущений. Як ім'я токена може бути використаний будь-який абстрактний символ, наприклад такий, як assign, але для зручності запису як ім'я абстрактного символу можна використовувати саму лексему.

3. *initial* є лексемою, яка відображається в токен *(id 2),* де 2 вказує на запис у таблиці символів для *initial*.

4. + є лексемою, що відображається в токен (+).

5. *rate* – лексема, що відображається в токен *(id, 3),* де 3 вказує на запис у таблиці символів для *rate*.

6. * – лексема, що відображається в токені (*).

7. 60 – лексема, що відображається в токен (number, 4), де 4 вказує на запис таблиці символів для внутрішнього уявлення цілого числа 60.

Прогалини, що поділяють лексеми, лексичним аналізатором відкидаються.

Подання інструкції надання після лексичного аналізу у вигляді послідовності токенів.

При цьому поданні імена токенів =, + і * є абстрактними символи для операторів присвоювання, складання та множення відповідно.

Шаблон (pattern) – це опис виду, який може набувати лексеми токена. У у випадку ключового слова шаблон є просто послідовність символів, які утворюють це ключове слово.

З теоретичної точки зору лексичний аналізатор не є обов'язковим, необхідною частиною компілятора. Його функції можуть виконуватися на етапі синтаксичного аналізу. Однак існує кілька причин, виходячи з яких до складу практично всіх компіляторів включають лексичний аналіз:

– спрощується робота з текстом вихідної програми на етапі синтаксичного аналізу і скорочується обсяг оброблюваної інформації, оскільки лексичний аналізатор структурує вихідний текст програми, що надходить на вхід, і викидає всю незначну інформацію;

– для виділення в тексті та розбору лексем можна застосовувати просту, ефективну і теоретично добре опрацьовану техніку аналізу, у той час як на етапі синтаксичного аналізу конструкцій вихідної мови використовуються достатньо складні алгоритми аналізу;

– сканер відокремлює складний за конструкцією синтаксичний аналізатор від роботи безпосередньо з текстом вихідний програми, структура якого може варіюватися в залежності від версії вхідної мови – за такої конструкції компілятора при переході від однієї версії мови до іншої достатньо лише перебудувати відносно простий сканер.

Функції, що виконуються лексичним аналізатором, та склад лексем, які він виділяє в тексті вихідної програми, можуть змінюватись в залежності від версії компілятор. В основному лексичні аналізатори виконують виняток із тексту вихідної програми коментарів та незначних прогалин, а також виділення лексем

наступних типів: ідентифікаторів, рядкових, символьних та числових констант, ключових (службових) слів вхідної мови.

У більшості компіляторів лексичний та синтаксичний аналізатори – це взаємопов'язані частини. Лексичний розбір вихідного тексту у такому варіанті виконується поетапно так, що синтаксичний аналізатор, виконавши розбір чергової конструкції мови, звертається до сканера за наступною лексемою. При цьому він може повідомити інформацію про те, на яку лексему слід очікувати. У процесі розбирання може навіть відбуватися "відкат назад", щоб виконати аналіз тексту на іншій основі. Надалі будемо виходити з припущення, що всі лексеми можуть бути однозначно виділені сканером на етапі лексичного розбору.

У найпростішому випадку фази лексичного та синтаксичного аналізу можуть виконувати компілятором послідовно. Але для багатьох мов програмування інформації на етапі лексичного аналізу може бути недостатньо для однозначного визначення типу та меж чергової лексеми.

Ілюстрацією такого випадку може бути приклад оператора присвоювання з мови програмування, який має лише одну вірну інтерпретацію (якщо операції розділити пробілами).

Якщо неможливо визначити межі лексем, то лексичний аналіз вихідного текст повинен виконуватися поетапно. Тоді лексичний та синтаксичний аналізатори повинні функціонувати паралельно, по черзі звертаючись одне до одного. Лексичний аналізатор, знайшовши чергову лексему, передає її синтаксичному аналізатору, який намагається виконати аналіз ліченої частини вихідної програми і може або запитати у лексичного аналізатора наступну лексему, або вимагати від нього повернутися на кілька кроків тому і спробувати виділити лексеми з іншими межами.

Очевидно, що паралельна робота лексичного та синтаксичного аналізаторів більш складна у реалізації, ніж їх послідовне виконання. Крім того, така реалізація вимагає більше обчислювальних ресурсів та у загальному випадку більшого часу на аналіз вихідної програми.

Щоб уникнути паралельної роботи лексичного та синтаксичного аналізаторів, розробники компіляторів та мов програмування часто йдуть на розумні обмеження синтаксису вхідної мови. Наприклад, для мови *C++* прийнято, що при виникненні проблем із визначенням меж лексеми завжди вибирається лексема максимально можливої довжини [40]. Для розглянутого вище оператора присвоєння це призводить до того, що під час читання четвертого знаку з двох варіантів лексем лексичний аналізатор вибирає найдовшу. Будь-які неоднозначності при аналізі даного оператора присвоєння можуть бути виключені тільки у разі правильного розміщення прогалин у вихідній програмі [40].

Вид подання інформації після виконання лексичного аналізу повністю залежить від конструкції компілятора. Але в загальному вигляді її можна подати як таблицю лексем, яка в кожному рядку повинна містити інформацію про вид лексеми, її тип і, можливо, значення. Зазвичай така таблиця має два стовпці: перший рядок лексеми, другий – покажчик на інформацію про лексем, може бути включений і третій стовпець – тип лексем. Таблиця лексем містить весь текст вихідної програми, оброблений лексичним аналізатором. До неї входять усі можливі типи лексем, при цьому, будь-яка лексема може в ній зустрічатися будь-яку кількість разів. Таблиця ідентифікаторів містить лише такі типи лексем: ідентифікатори та константи. До неї не потрапляють ключові слова вхідної мови, знаки операцій та роздільники. Кожна лексема у таблиці ідентифікаторів може зустрічатися лише один раз.

У цій таблиці поле «значення» має на увазі якесь кодове значення, яке буде поміщено у підсумкову таблицю лексем. Значення, наведені у прикладі, є умовними. Конкретні кодові значення обираються розробниками під час реалізації власних компіляторів.

Зв'язок між таблицею лексем і таблицею ідентифікаторів відображено у прикладі деяким індексом, наступним після ідентифікатора за потрібним знаком. У реальному компіляторі ця зв'язок визначається його реалізацією.

Лексичний аналізатор має справу з таким об'єктами, як різного роду константи та ідентифікатори (до останніх належать і ключові слова). Мова

констант та ідентифікаторів у більшості випадків є регулярним, тобто, може бути описаний з допомогою регулярних (праволінійних чи ліволінійних) граматик. Розпізнавачами для регулярними мовами є кінцеві автомати (КА). Кінцеві автомати є розпізнавачами; вони просто кажуть «так» чи «ні» для кожної можливої вхідної рядки.

Кінцеві автомати поділяються на два класи.

1. Недетерміновані кінцеві автомати немає обмежень за свої дуги. Символ може бути міткою кількох дуг, що виходять з того самого стани; крім того, одна з можливих міток – порожній рядок.

2. Детерміновані кінцеві автомати для кожного стану та кожного символу вхідного алфавіту мають рівно одну дугу із зазначеним символом, що залишає поточний стан.

Як недетермінований, так і детермінований кінцеві автомати можна подати у вигляді графа переходів, вузли якого представляють стани, а позначені дуги представляють функцію переходів.

Існують правила, за допомогою яких для будь-якої регулярної граматики може бути побудований недетермінований кінцевий автомат, що розпізнає ланцюжки язика, заданого цією граматикою.

Робота автомата виконується за тактами. На кожному черговому такті і автомат, перебуваючи в деякому стані, зчитує черговий символ із вхідної ланцюжка символів та змінює свій стан на, після чого покажчик у ланцюжку вхідних символів пересувається на наступний символ і починається такт продовжується доти, доки ланцюжок вхідних символів не закінчиться. Кінець ланцюжка символів часто позначають спеціальним символом. Вважається також, що перед тактом один автомат знаходиться у початковому стані.

Кажуть, що автомат допускає певний ланцюжок, якщо в результаті виконання всіх тактів роботи над цим ланцюжком він виявиться в певному стані. Мова, яка визначається автоматом, є множиною всіх ланцюжків, що допускаються автоматом. Для аналізу ланцюжка, достатньо подати його на вхід

автомата, виконати всі такти його роботи та визначити, чи перейшов автомат в результаті роботи в одне з заданих кінцевих станів.

Недетермінований автомат незручний для аналізу ланцюжків, оскільки у ньому можуть траплятися стану, допускають неоднозначність, тобто. такі, з яких виходить дві або більше дуги, позначені одним і тим самим символом. Очевидно, що програмування роботи такого автомата – нетривіальне завдання.

Доведено, що будь-який недетермінований автомат може бути перетворений на детермінований так, щоб їхні мови збігалися (кажуть, що автомати еквівалентні).

Після побудови кінцевий детермінований автомат може бути мінімізований, тобто. для нього може бути побудований еквівалентний йому автомат з мінімальним числом станів.

Можна написати функцію, яка відображає функціонування будь-якого детермінованого кінцевого автомата. Щоб запрограмувати таку функцію, достатньо мати змінну, яка б відображала поточний стан автомата, а переходи автомата з одного стану в інший на основі символів вхідного ланцюжка можуть бути збудовані за допомогою операторів вибору. Робота функції повинна продовжуватися до тих пір, поки не буде досягнуто кінця вхідного ланцюжка. Для обчислення результату функції необхідно після її завершення проаналізувати стан автомата. Якщо це один із кінцевих станів, то функція виконана успішно, і вхідний ланцюжок приймається, якщо ні – то вхідний ланцюжок не належить заданій мові.

Загалом завдання сканера дещо ширше, ніж просто перевірка ланцюжка символів лексеми на відповідність її вхідної мови Сканер повинен виконати ті чи інші по запам'ятовування розпізнаної лексеми (занесення їх у таблицю лексем).

Набір дій визначається реалізацією компілятора. Зазвичай ці дії виконуються відразу ж по виявленню кінця лексеми, що розпізнається, тому їх нескладно вставити в відповідні місця програми-сканера.

Друга проблема, яка вже обговорювалася вище, – це виділення меж лексем. Адже у вхідному тексті лексеми не обмежені особливими знаками. Якщо

говорити в термінах програми-сканера, то визначення меж лексем – це виділення тих рядків у загальному потоці вхідних символів, котрим треба виконувати розпізнавання. В загальному у разі ця задача може бути складною, але для найпростіших вхідних мов кордону лексем розпізнаються за заданими термінальними символами. Ці символи – прогалини, знаки операцій, символи коментарів, і навіть роздільники (коми, крапки з комою та інших.).

Набір таких термінальних символів може змінюватись в залежності від вхідної мови.

Важливо, що знаки операцій самі також є лексемами, і не потрібно пропустити їх під час розпізнавання тексту.

Таким чином, алгоритм роботи найпростішого сканера можна описати так:

– переглядається вхідний потік символів програми вихідною мовою до виявлення чергового символу, що обмежує лексему;

– для вибраної частини вхідного потоку виконується функція розпізнавання вхідної лексеми;

– при успішному розпізнаванні інформація про виділену лексему заноситься до таблиці лексем, та алгоритм повертається до першого етапу;

– при неуспішному розпізнаванні видається повідомлення про помилку, а подальші події залежать від реалізації сканера: або його виконання припиняється, або робиться спроба розпізнати наступну лексему (йде повернення до першого етапу алгоритму).

Робота програми-сканера продовжується доти, доки не будуть переглянуті всі символи програми вихідною мовою із вхідного потоку (рисунок 2.5).

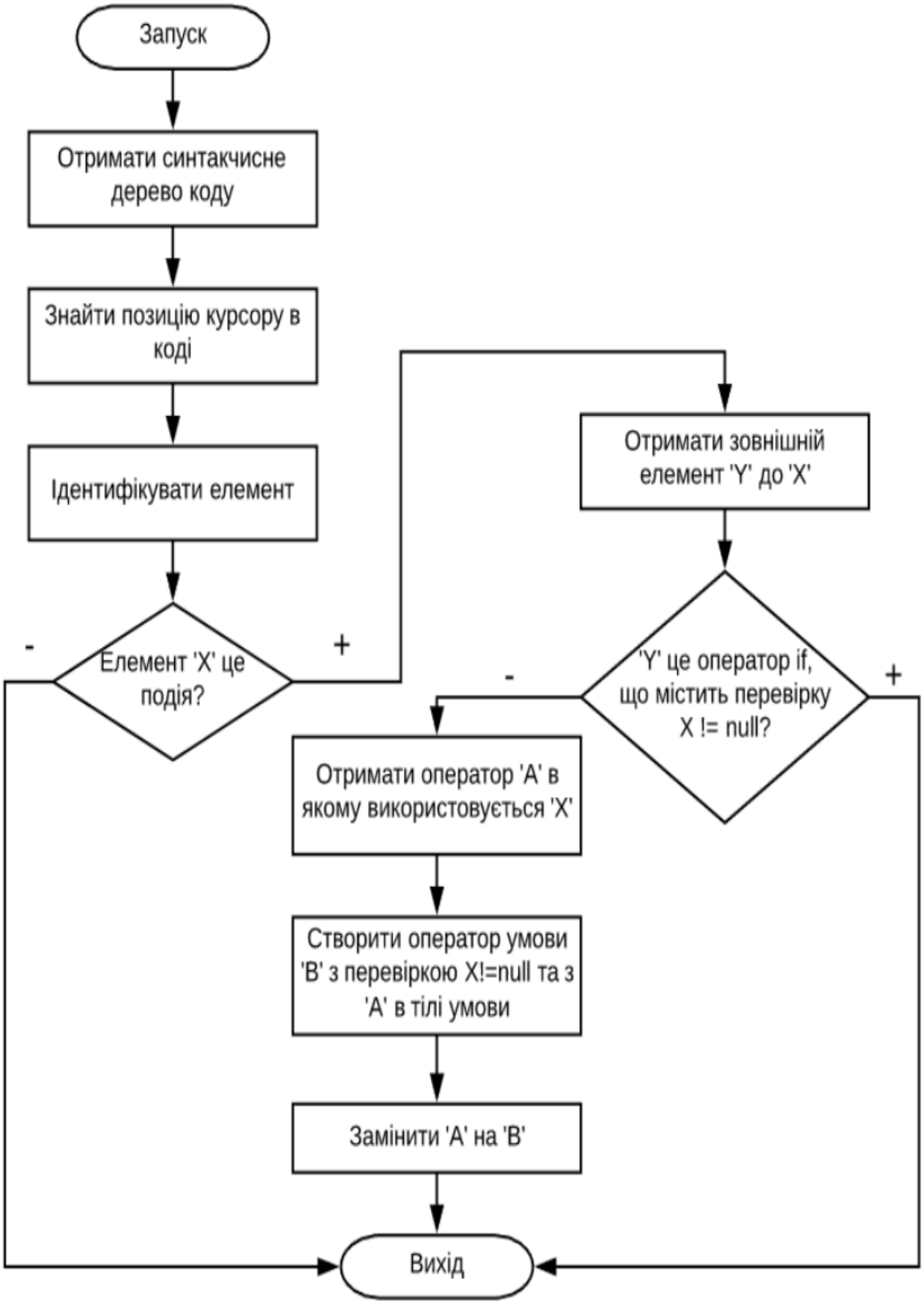


Рисунок 2.5 – Алгоритм створення бінарного дерева заміни найменування ідентифікаторів

На рисунку 2.6 запропоновано алгоритм додавання та заміни найменування ідентифікаторів із переліку граматичних шаблонів. Для початку, як зазвичай визначається синтаксичне дерево коду. Потім знаходиться активна позиція коду та відбувається знаходження ідентифікаторів відповідного елементу. Пропозиція даного рефакторингу буде відображатись тільки для типу подія, яка має перевірений тип, тобто делегат в якого визначений відповідний метод.

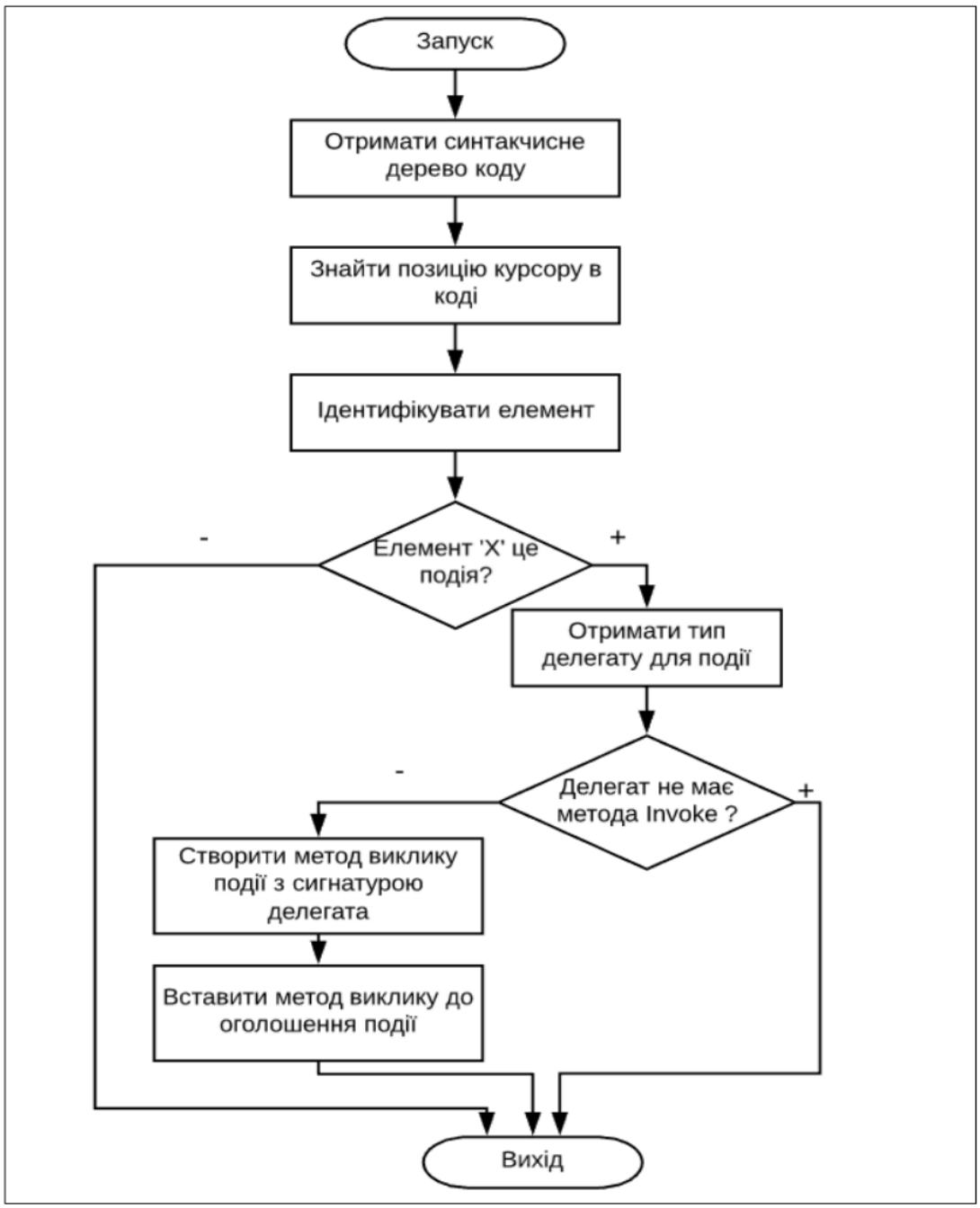


Рисунок 2.6 – Алгоритм додавання та заміни найменування ідентифікаторів із переліку граматичних шаблонів

До фінального етапу відноситься генерація методу виклику події. На цьому етапі здійснюється визначення делегату на основі наданих сигнатур, що можуть безпосередньо використовуватись у створюваному методі виклику події. На етапі генерації та після її закінчення відбувається заміщення того синтаксичного дерева, яке вже існує. Це здійснюється шляхом додавання створеного відповідного методу, що додається перед оголошенням події

2.4 Висновки

Отже, більшість ресурсів життєвого циклу програмного забезпечення виділяється на обслуговування програми. Технічне обслуговування значною мірою залежить від розуміння програми, оскільки розробники зазвичай витрачають значні кошти частину свого часу на розуміння коду, який вони підтримують перед застосуванням змін, налагодження, документування тощо. Зрозуміло, що полегшення розуміння коду полегшить багато обслуговування та підвищення продуктивності розробників. Один із основних способів для розробника зрозуміти, що робить частина коду, – це ідентифікатори в коді.

У другому розділі було здійснено розробку концептуальної моделі процесу оцінки якості найменування ідентифікаторів. Також описано метод оцінки якості найменування ідентифікаторів та здійснено його удосконалення шляхом введення граматичних шаблонів. Суть методу полягає в тому, що спочатку формується відповідна таблиця ідентифікаторів, потім здійснюється пошук елемента, тобто найменування ідентифікатора. Це здійснюється за допомогою методу бінарного пошуку, тобто формується бінарне дерево. Після чого, якщо найменування ідентифікатора не відповідає вимогам, то підключаються граматичні шаблони і відбувається заміна на потрібний.

3 АРХІТЕКТУРА ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Формування та аналіз вимог програмної реалізації системи оцінки якості найменування ідентифікаторів вихідного коду

Перевірка правильності семантики та генерація коду вимагають знання характеристик основних елементів вихідної програми – змінних, констант, функцій та інших лексичних одиниць, що зустрічаються на вхідній мові.

Головною характеристикою будь-якого елемента вихідної програми є його ім'я (ідентифікатор). Найменування кожного елемента повинно бути унікальним. Виділення найменування ідентифікаторів, а також всіх решту елементів вихідного коду програмної системи здійснюється на етапі лексичного аналізу.

Для того, щоб використовувати всі найменування ідентифікаторів, а також їх характеристики потрібно, щоб безпосередньо компілятор мав до них доступ протягом всього процесу компіляції та міг розпізнати та використовувати ці дані на всіх її етапах.

Синтаксичний аналізатор здійснює розпізнавання всього тексту вихідного коду програмної системи і йому не має необхідності встановлювати межі рядка символів, які розпізнаються. Лексичний аналізатор працює навпаки. Він повинен сприймати всю інформацію, що надходить йому на вхід, і або підтвердити її приналежність до вхідної мови, або повідомити про помилку у вихідній програмі.

Разом з тим потрібно зауважити, що синтаксичний аналіз – це не тільки перевірка приналежності певного ланцюжка певній вхідній мові. Завданням даного аналізу є оформлення знайдених синтаксичних конструкцій для того, щоб здійснити надалі генерування тексту тієї програми, що буде у результаті розпізнавання. Синтаксичний аналізатор має містити певну вихідну мову. За допомогою даної вихідної мови здійснюється передача інформації про знайдені розібрані синтаксичні структури наступним етапам компіляції. На даному етапі відбувається перетворення різновиду поточного автомату на перетворювач з магазинною пам'яттю.

Якщо є в наявності граматика вхідної мови, то при розробці синтаксичного аналізатора має бути здійснено певна сукупність формальних перетворень над цією граматикою. В свою чергу це полегшить побудову системи оцінки якості найменування ідентифікатора (СОЯІ). Після цього має бути перевірка отриманої граматики на відповідність одному із відомих класі КС-мов, для яких існують лінійні СОЯІ. У випадку, якщо такий клас існує і він знайшовся, то можна починати побудову СОЯІ. У випадку, якщо виокремлено кілька таких класів, то потрібно обирати шлях найпростішої із найкращими характеристиками. У ситуації відсутності чи неможливості знаходження такого класу КС-мов, то потрібно здійснити виконання деяких перетворень над граматикою. Ці маніпуляції мають призвести до відповідності із якимось відомим класом. Навіть, якщо є очевидним той факт, що граматика підпадає під якийсь із відомих класів КС, доцільним є перевірка всіх класів з метою знаходження якнайкращого. Тоді є можливість здійснити побудову СОЯІ із гарними характеристиками.

Важливо зазначити, абсолютно для всіх класів КС-граматик виділяють досить важливе обмеження, що є принциповим. Суть даного обмеження полягає у тому, що якщо є неможливість перетворення будь-якої КС-граматики на той вид, що задовільняє вимогам певних класів КС-граматик, то потрібно довести, що такого перетворення немає. Загалом, потрібно виходити до вимог у кожному конкретному випадку і все ж таки шукати відповідні перетворення для знаходження якнайкращого результату. Потрібно також зауважити, що великий набір класі КС-граматик сприяє позитивному результату, а отже у кінцевому варіанті якісній побудові СОЯІ.

Якщо ж усі маніпуляції із пошуком здійснені, а відповідний клас КС-мов так і не вдалось знайти, то в такому випадку здійснюється розробка універсальної системи оцінювання якості найменувань ідентифікаторів.

Слід також зауважити, що в такому випадку характеристики універсального СОЯІ будуть дещо не такими якісними на відміну від лінійного СОЯІ. Найкращим варіантом буде досягнення квадратичної залежності часу роботи СОЯІ від довжини вхідного ланцюжка. В реальних умовах такий варіант

фактично виключається, оскільки задача всіх сучасних компіляторів здійснювати роботу швидко та оперативно. Саме тому вони побудовані на основі лінійних СОЯІ. В протилежному випадку час їхньої роботи був би дуже великим.

Існують також випадки, при яких одна граматика може відноситись до кількох класів граматики одночасно. І ці граматики дозволяють створення лінійних СОЯІ. В таких ситуаціях потрібно вибрати найоптимальніший клас граматики для якнайкращого результату реалізації.

Відповісти на це питання не завжди легко, оскільки можуть бути побудовані два принципово різних СОЯІ, алгоритми роботи яких неспівставні. У насамперед йдеться саме про висхідних і низхідних СОЯІ: в основі перших лежить алгоритм підбору альтернатив, основу других – алгоритм «зсув-згортка».

На питання про те, який СОЯІ – низхідний чи висхідний – вибрати для побудови синтаксичного аналізатора немає однозначної відповіді. Цю проблему необхідно вирішувати, спираючись на додаткову інформацію про те, як будуть використані або яким чином будуть опрацьовані результати роботи СОЯІ.

Бажання використати простіший клас граматик для побудови СОЯІ може вимагати якихось маніпуляцій із заданою граматикою, необхідні її перетворення до необхідного класу. При цьому часто граматика стає неприродною та малозрозумілою, що надалі ускладнює її використання для генерації результуючого коду. Тому буває зручним використовувати вихідну граматику такою, якою вона є, не прагнучи перетворити її до простішого класу.

Отже, під час дослідження та на підтвердження теоретичних викладок, потрібно написати програму, яка виконує лексичний аналіз вхідного тексту відповідно до завдання, породжує таблицю лексем і виконує синтаксичний розбір тексту за заданою граматикою з побудовою дерева розбору. Текст на вхідній мові задається у вигляді символьного (текстового) файлу.

За наявності у вхідному файлі тексту, що відповідає заданій мові, програма повинна будувати та відображати дерево синтаксичного аналізу. Якщо ж текст у вхідний файл містить помилки (лексичні або синтаксичні), програма повинна

видавати повідомлення про наявність помилок у вхідному тексті та коректно завершувати своє виконання.

Потрібно здійснити розбиття програми на три складові:

– здійснити аналіз;

– рекомендація граматичного шаблону, після чого здійснюється побудова ланцюжка виведення;

– видалення непотрібного найменування ідентифікатора.

Дана система оцінки якості найменування ідентифікатора повинна виділяти в тексті коду ідентифікатори мови та замінювати їх на термінальний символ граматики. Отриманий після лексичного аналізу ланцюжок повинен розглядатися в другій частині програми у відповідності до алгоритму розбору. При невдалому завершенні алгоритму видається повідомлення про помилку, при вдалому – будується ланцюжок виведення. Після побудови ланцюжка виведення на його основі будується дерево розбору, в якому символи послідовно замінюються на потрібні ідентифікатори із запропонованих граматичних шаблонів.

3.2 Розробка архітектури програмної реалізації

Реалізований як інструмент командного рядка чи консолі, даний програмний засіб інтегрується з деякими добре відомими бібліотеками та інструментами з відкритим кодом для аналізу вихідного коду для виявлення порушень імені ідентифікатора.

На рисунку 3.1 показано концептуальну архітектуру розроблюваного програмного забезпечення. Загалом, програмна система складається з наступних трьох рівнів:

– платформи;

– модулів;

– інтерфейсу.

Даний плагін може використовувати добре відомі інструменти та бібліотеки, що використовуються для природної мови та статичного аналізу, включаючи Spiral, NLTK, Wordnet, Стенфордське тегування POS і srcML тощо.

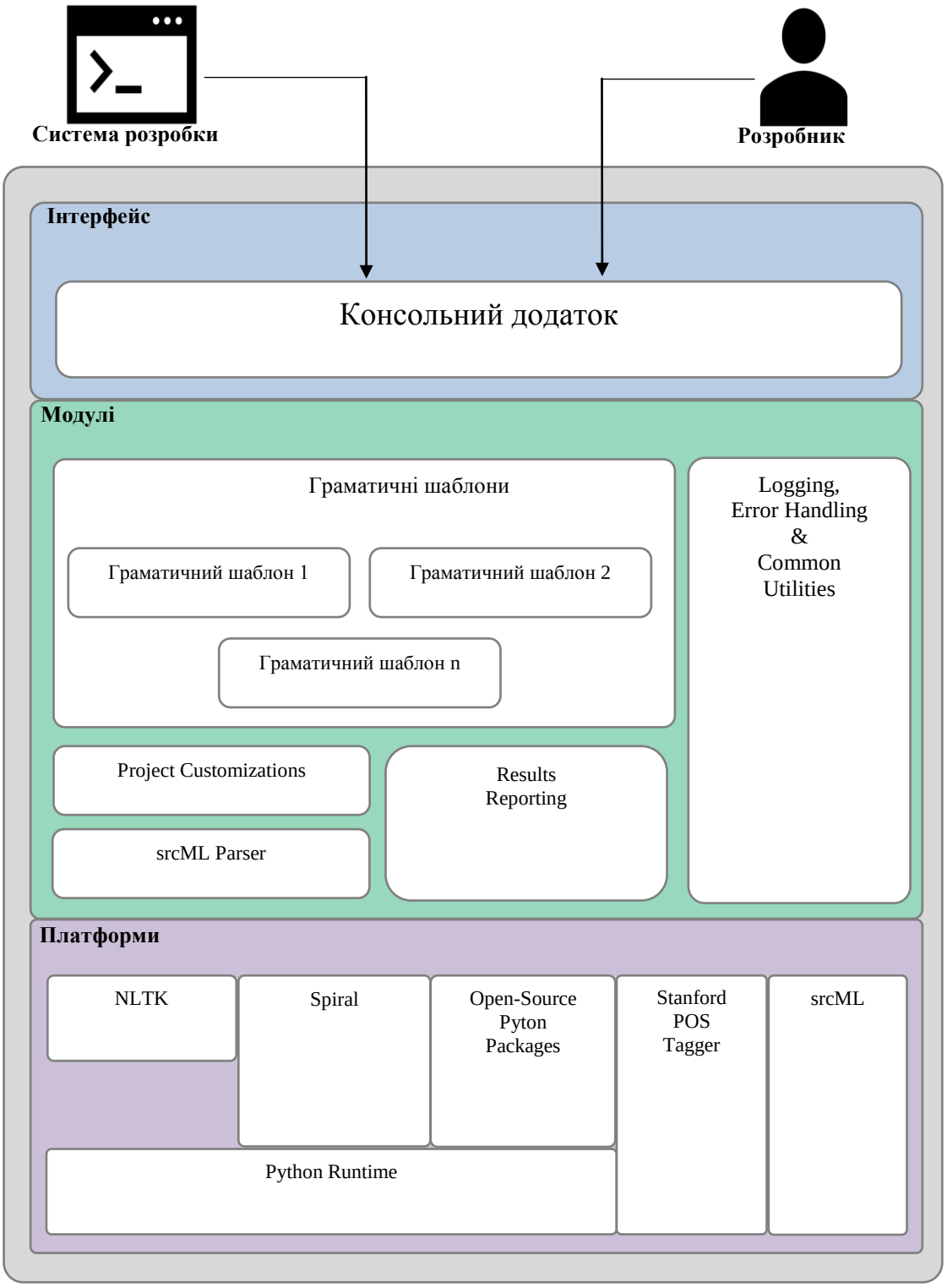


Рисунок 3.1 – Архітектура розроблюваного програмного забезпечення

4. ПРОГРАМНА РЕАЛІЗАЦІЯ

4.1 Вибір інструментарію та середовища реалізації

Оскільки задача, що поставлена у даному дослідженні полягає у розробці програмного забезпечення із надання порад розробникам щодо оцінки якості найменування ідентифікаторів та її покращення у режимі реального часу, то даною розробкою буде плагін, що розробляється у системі на основі відкритого сирцевого коду.

Існує досить широкий спектр інструментарію із відкритим вихідним кодом, для розробки плагінів. Аналіз та огляд декотрих представлено нижче.

Jenkins – сервер автоматизації з відкритим вихідним кодом, побудований на JavaVirtualMachine (JVM), що підтримує тисячі плагінів для розробки, розгортання та автоматизації програмних проектів. Вони є лідером у галузі автоматизації серверів з відкритим вихідним кодом. Інші компанії, які надають сервери автоматизації з відкритим вихідним кодом є TeamCity, CircleCI, Хадсон, Тревіс CI, AtlassianBamboo.

В рамках сучасних методів розробки програмного забезпечення, таких як ліве тестування та безперервна інтеграція, розробники, тестувальники та дизайнери використовують Jenkins для створення, розгортання та автоматизації своїх тестів. Згодом, у міру того, як цикли розробки та тестування ставали коротшими і частіше через технічний прогрес та акцент на цифровому досвіді, стало очевидно, що для того, щоб випустити якісні програмні функції, команди загалом мають стати більш гнучкими. Це означає, що тестування в сучасних умовах «зміщується» вліво і відбувається раніше, як правило, після написаного початкового коду. Дане правило «тест рано і часто» особливо важливе у безперервній інтеграції та безперервній доставці (CI/CD), де мета полягає в тому, щоб мати можливість послідовно та постійно оновлювати інструментарій.

Перевага безперервної інтеграції полягає в тому, що вся робота та код розробки можуть бути інтегровані в процес раніше, допомагаючи швидше та ефективніше виявляти та виправляти помилки чи баги. Чим швидше розробник

зможе знайти і виправити помилки програмного забезпечення, тим більша ймовірність того, що йому не доведеться виправляти це у виробництві, де це може займати більше часу та коштувати дорого.

Щоб розширити безперервну інтеграцію, команди використовують безперервні методи доставки для автоматизації та керування циклами випуску програмного забезпечення замість автоматизації процесу тестування програмного забезпечення. Крім того, групи можуть використовувати безперервне розгортання, що суттєво усуває необхідність практично у будь-якій участі людини. Усі зміни, які відбуваються через етапи розробки програмного забезпечення, автоматично вивільняються у виробництво.

Безперервна інтеграція (CI) та безперервна доставка (CD) уособлюють культуру, набір принципів роботи та набір практик, які дозволяють групам розробників додатків частіше та надійно вносити зміни до коду. Це відомо як конвеєр CI/CD. Це також найкращий метод гнучкої методології, оскільки вона дозволяє групам розробників програмного забезпечення зосередитися на вирішенні бізнес-передумов, якості коду та безпеки в міру автоматизування кроків розгортання.

Існує різниця між CI та CD, не зважаючи на злиття двох термінів, а саме: безперервна інтеграція, безперервна доставка, тестованість.

Безперервна інтеграція – це шлях до координації коду до бази коду. Практично у всіх сучасних ситуаціях безперервну інтеграцію завершено з використанням етапів, яким структуруватися явно не так просто. Виконання безперервної інтеграції породжує необхідність використання призначених для цього інструментів.

Безперервна доставка, з іншого боку, по суті встановлює автоматизацію всього циклу тестування програмного забезпечення та намагається видалити якнайбільше ручної роботи, яка дозволяє розробникам виділяти час для роботи над іншими проектами та звільняє від керування всіма окремими кроками.

Переваги CI/CD:

– зміни коду менш складні і мають менш ненавмисні результати. Відключення дефіциту є менш складним та швидким. Проміжне тестування зменшується за рахунок зменшення середнього часу виявлення помилок;

– тестованість покращується через менші, явні зміни. Ці маленькі зміни дозволяють проводити все більш точні позитивні та негативні тести.

IDEAL – це багатомовна платформа для аналізу імен ідентифікаторів. Вона залежить від контексту створюваного програмного коду, тестування, а також підтримки. Це дозволяє працювати із конкретною конфігураціє проекту та базується на srcML, що дозволяє підтримувати декілька мов програмування (зокрема, Java та C#). IDEAL доступний у відкритому доступі інструмент для полегшення оновлення та підтримки, що дозволяє її широко використовувати розробникам та дослідникам.

Інструмент з відкритим кодом IDEAL виявляє та повідомляє про порушення в іменах ідентифікаторів для використання кількох мов програмування методи статичного аналізу. На додаток до визначення ідентифікаторів, які виявляють проблеми з іменуванням у вихідний код, IDEAL також надає необхідну інформацію для кожного повідомленого порушення, щоб можна вжити відповідних заходів для вирішення проблеми. Актуальним є використання IDEAL розробниками у створенні та підтримці високоякісних імен ідентифікаторів у своїх проектах, а також у дослідженнях для вивчення розповсюдження та ефекту різних поганих практик присвоєння імен у цій галузі.

4.2 Програмна реалізація та тестування системи управління якістю розробки програмних продуктів

Результати проведених досліджень показують закономірності в тому, як розробники створюють найменування ідентифікаторів і те, що вони вважають сильним або якісним іменем. Як показують дослідження, автоматичне визначення

значення термінів (або слів) в ідентифікаторі є складним. Проте, граматичний зразок імені надає більш здійсненний механізм для аналізу якості найменування. До перевірки результатів даного дослідження на основі розробленої архітектури було створено відповідне програмне забезпечення, а саме плагін який включає евристики для допомоги розробникам в оцінці якості найменування ідентифікаторів вихідного коду та надає рекомендації по зміні та виправленню неправильних імен ідентифікаторів в режимі реального часу.

Плагін використовує налаштування конкретної частини мови для визначення граматичної моделі вибраного ідентифікатора, враховуючи граматичні шаблони. Ці евристики включають дійсні граматичні моделі та зв’язок ідентифікатора з навколишнім кодом (наприклад, тип даних), щоб визначити, чи потрібно розробнику змінити найменування ідентифікатора. Якщо ім’я поганої якості, інструмент рекомендує альтернативний граматичний шаблон, включаючи виділення термінів у назві, які слід видалити. Плагін також пояснює запропоновану рекомендацію.

На рисунку 4.1 показано скріншот плагіна, що використовується. Якщо ім’я ідентифікатора порушує правило іменування, плагін підкреслює назву зеленою хвилястою лінією. Крім того, після натискання вибору найменування, користувацький інтерфейс плагіна оновлюється, щоб показати поточний граматичний шаблон імені та рекомендований граматичний шаблон, якщо такий є. Терміни в назві, які слід видалити, відображаються червоним кольором шрифта, тоді як теги частини мови, які слід додати до назви, відображаються зеленим шрифтом. Крім того, плагін також надає приклад і пояснення запропонованої рекомендації. На рисунку 4.2 видно, що плагін також надає розробникам звітність по всіх порушеннях найменувань ідентифікаторів у вихідному файлі програмного забезпечення.

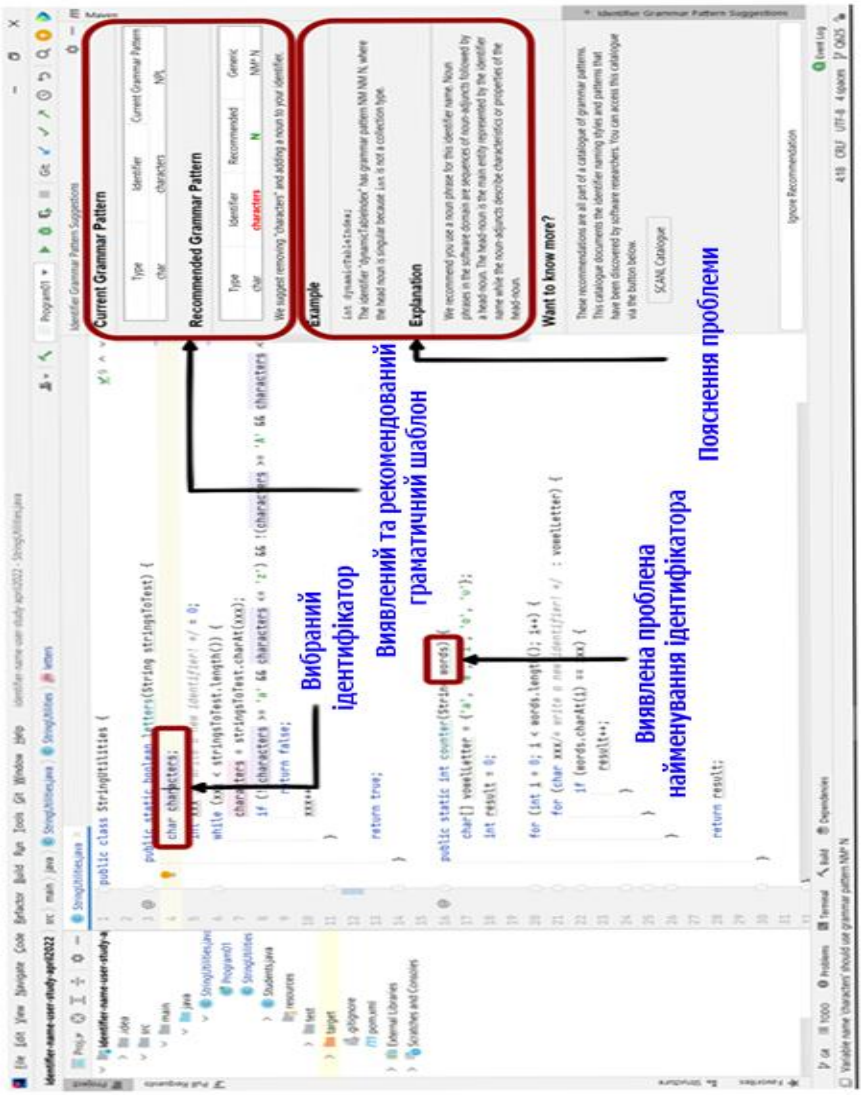


Рисунок 4.1 – Скріншот плагіна, що використовується

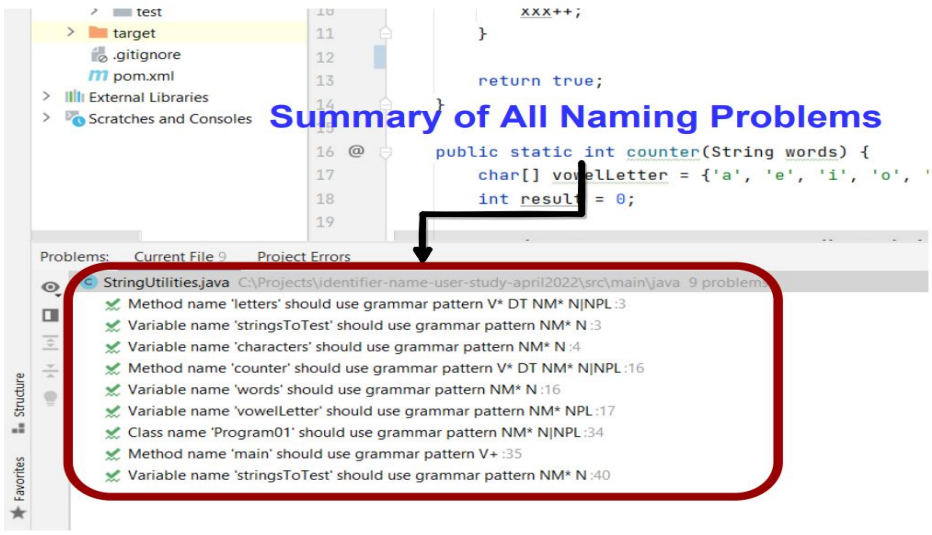


Рисунок 4.2 – Звіт помилок у найменуваннях ідентифікаторів

На рисунку 4.3 наведено загальний огляд архітектури оцінки імені та рекомендації підключати. Плагін складається з чотирьох основних частин: (1) Оцінка граматичного шаблону, (2) Граматична рекомендація щодо шаблону, (3) Приклади та пояснення та (4) Виключення ідентифікатора. Плагін використовує тег програмного коду для створення тегів частини мови під кожен термін в найменуваннях ідентифікаторів. Даний інструментарій доступний в режимі реального часу, оскільки під час здійснення оцінки якості найменування ідентифікаторів здійснюється правильність певного набору ідентифікаторів. У випадку виявлення неправильного найменування ідентифікаторів здійснюється його заміна на граматичний шаблон. Невірне найменування видаляється.

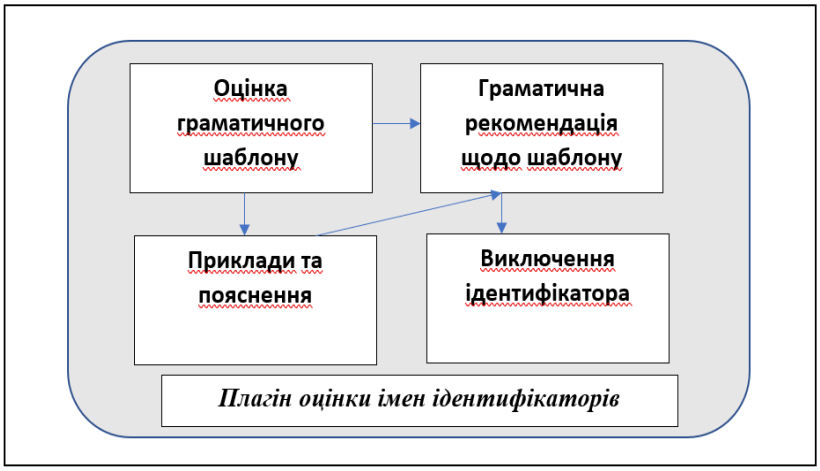


Рисунок 4.3 – Структура плагіна

Щоб зрозуміти ефективність даного плагіну у правильному виявленні порушень найменування ідентифікаторів, можна перевірити його за допомогою двох типів оціночної діяльності. Це можна здійснити шляхом здійснення аналізу популярних open-source систем з використанням плагіну і вручну перевірити результати виявлення статистично значущі зразок. Далі відбувається оцінювання плагіну на використаному наборі даних, щоб оцінити його шляхом порівняння результатів виявлення неправильних найменувань.

Плагін може аналізувати системи, реалізовані будь-якою мовою, що підтримується srcML. Однак наразі його було оцінено лише за допомогою Java та

C#. Таким чином, було обрано дві популярні системи з відкритим кодом для кожної з цих мов програмування.

Для кожної з чотирьох систем було проаналізовано випадкову стратифіковану статистично значущу (тобто, рівень довіри 95% та довірчий інтервал 10%) набір виявлених порушень для кожної категорії.

Загалом можна вручну здійснити перевірку порушення імен у чотирьох системах. Як результат здійснено розбивку кількості випадків порушення для кожної категорії. Як частина процесу ручного аналізу та щоб пом'якшити упередженість, конкретні випадки порушення, які були суб'єктивними та інколи містили посилання на літературу (сіру та рецензовану) не приймались до уваги, щоб допомогти в процесі прийняття рішень. Загалом, плагін повідомляє про середню точність 75,27%, з 14 із 19 порушень типів, що повідомляють про точність понад 50%. Крім того, навіть якщо плагін підтримує налаштування для кожного проекту (наприклад, визначення типів і термінів спеціальних колекційних даних), дана стратегія не використовувала цю функцію, щоб підтримувати узгодженість у виявленні порушень використовуваних чотирьох систем з відкритим кодом.

Дана архітектура може сприяти хорошим результатам у виявленні всіх порушень (показник точності понад 80%). Це антишаблони, де ідентифікатор також робить або містить більше, ніж потрібно. У більшості випадків програмна система може точно обробити повернення чи тип даних ідентифікатора для визначення порушень. Проте є й такі порушення для програмної системи, що складно аналізувати і, отже, призводять до великої кількості помилкових спрацьовувань. Ручний аналіз цих хибнопозитивних випадків показує закономірності, які в більшості випадків є через те, що плагін повідомляє про них як про проблеми. По-перше, оскільки розробники використовують спеціальні дані чи типи повернення для найменувань ідентифікаторів у своєму коді, програмна система не може визначити їх цільове призначення. Наприклад, «EnvVars» – це спеціальний тип, створений розробником для зберігання колекції певних елементів. Розробник повертає цей тип у методі під назвою

«getEnvironmentVariables2». Оскільки плагін не знає, що «EnvVars» є типом на основі колекції, він позначає це як порушення, оскільки метод має повертатись як колекція (тобто ця назва методу отримання містить термін у множині – «Змінні»).

Якщо здійснити адекватне налаштування для обробки користувацьких типів, кількість помилкових спрацьовувань значно зменшиться.

Наступним плюсом у роботі даного плагіна є аналіз лексичних зв'язків між словами; зокрема, щодо антонімів. Хоча дана система правильно розпізнає антоніми, у контексті загалом те, як ці терміни використовуються в назві ідентифікатора чи коментарі, не розглядається, що призводить до помилкових спрацьовувань. Крім того, також спостерігається така тенденція, що звички чи конвенції щодо найменувань також спричиняють появу антонімів. Наприклад, розглянемо метод «GetCompletionResult» із поверненням типу «CompletionResult». Програмна система визначає, що «Отримати» і «Результат» є антонімами, які є лексично дійсними, але є хибно позитивними через правила іменування. Подібно до останнього виклику, у контексті використання термінів трансформації та умовних термінів є причиною повідомлення про велику кількість помилкових спрацьовувань. Хоча плагін правильно визначає ці терміни з вихідного коду, як розробник використовує термін у назві чи коментарі, зараз є проблемою.

Загалом, ручний перегляд вихідного коду також дозволив помітити інші неправильні чи погані найменування ідентифікаторів на практиці, які можуть бути майбутніми мовними антипатернами. Наприклад, загальні терміни «дані» і «результат» є суб'єктивними. Якщо використовується як частина назви ідентифікатора, невідомо, чи ідентифікатор обробляє один предмет, чи колекцію предметів. Так само використання типу var (наприклад, у C#) і «об'єкт» також не вказує на тип даних, які обробляє ідентифікатор. В ідеалі, щоб передати своє призначення чи поведінку ідентифікатора правильно, розробники повинні бути конкретними у встановленні найменувань ідентифікаторів та типів даних у всіх можливих випадках.

4.3 Висновки

У розділі описано програмну реалізацію системи управління якістю розробки. На основі проведеного дослідження, зроблених висновків та результатів було розроблено програмне забезпечення у вигляді плагіна. Дане програмне забезпечення дозволяє здійснювати оцінку якості найменування ідентифікаторів вихідного коду, що працює на основі заміни граматичних шаблонів.

ВИСНОВКИ

Отже, підсумовуючи, слід зазначити, такі аспекти проблеми, що досліджувались в даній роботі.

Незалежно від галузі чи технічної сфери, якість є критичним аспектом будь-якої програмної системи. Крім того, забезпечення якості програмного забезпечення не є одноразовим завданням; починаючи з моменту впровадження та продовжуючи протягом усього терміну служби системи, розробники виконують технічне обслуговування завдання на їх програмне забезпечення для досягнення функціональних і нефункціональних цілей. Цей наголос на якості програмного забезпечення призвів до того, що організації виділяють від 60% до 80% ресурсів на технічне обслуговування програмного забезпечення, що є найдорожчою фазою життєвого циклу розробки програмного забезпечення.

У першому розділі здійснено аналіз існуючих методів та практик щодо оцінки якості найменування ідентифікаторів вихідного коду. Встановлено, що розробка та розуміння програми є попередником усіх завдань обслуговування програмного забезпечення і тому важливо, щоб розробник розумів код, який він буде змінювати. Тому збереження внутрішньої якості коду протягом усього терміну розробки та супроводу програмного забезпечення має першочергове значення. Як фундаментальні елементи в вихідний код, імена ідентифікаторів складають у середньому майже 70% символів у програмному забезпеченні кодової бази системи і відіграють значну роль у розумінні коду. Низька якість ідентифікаторів може заважати розробникам зрозуміти код; в свою чергу вдалопідібрані найменування ідентифікаторів суттєво покращують розуміння діяльності приблизно на 19%.

Однак, не зважаючи на усе це, розробники все ще дуже мало дотримуються певних правил у створенні високоякісних найменувань ідентифікаторів.

У другому розділі здійснено огляд та удосконалення методу оцінки найменування ідентифікаторів вихідного коду, що полягає у застосуванні рефакторингу найменувань, що реалізовані табличним методом. На основі цього

методу розроблено алгоритм покращення найменування ідентифікаторів вихідного коду на основі граматичних шаблонів.

У третьому розділі здійснено розробку архітектури програмного забезпечення для реалізації проведеного дослідження, а саме подано концептуальну архітектуру розроблюваного програмного забезпечення. Загалом, програмна система складається з трьох рівнів: платформи; модулів; інтерфейсу. Також проаналізовано існуючий інструментарій для розробки даного програмного забезпечення на основі відкритого програмного коду.

У четвертому розділі здійснено програмну реалізацію плагіну, що дозволяє оцінювати якість найменування ідентифікаторів програмного вихідного коду та реалізувати рефакторинг на основі алгоритму граматичних шаблонів найменування ідентифікаторів. Даний плагін може використовувати добре відомі інструменти та бібліотеки, що використовуються для природної мови та статичного аналізу. Загалом, програмна система складається з трьох рівнів: платформи; модулів; інтерфейсу.

Практичне значення отриманих результатів. У даній магістерській роботі здійснено спробу встановити нові концепції алгоритму, які допомагають вимірювати та покращувати якість ідентифікаторів. Результатом цієї магістерскої роботи є набір інструментів оцінки якості найменування ідентифікаторів, які інтегруються в робочий процес розробника. Через послідовність емпіричних досліджень сформулювали низку методик та лінгвістичних шаблонів для оцінки якості ідентифікатора імена в коді та надати рекомендації щодо структури імен. Результати даного дослідження можуть покращити роботу програмістів, шляхом їх інтеграції у творчий процес розробки, що в свою чергу забезпечить процес створення та підтримки високої якості найменувань ідентифікаторів у вихідному коді програмних систем.

Результатом дослідження є проектування системи інструментів, створених на основі емпіричних наукових даних, які, при інтеграції в робочий процес розробника оцінює якість існуючих імен ідентифікаторів і надає рекомендації щодо високоякісних імен і структури імен. Каркас складається з плагіна та

інструмента командного рядка. Плагін надає розробникам можливість працювати в реальному часі рекомендації щодо типу слів (наприклад, дієслово, іменник) і структури, для яких слід використовувати розробникам якісні заміни імен. Офлайн-інструмент, який підтримує інтеграцію в систему збірки, звіти про мовніантипатерни (тобто відхилення від загальноприйнятих практик лексичного найменування) у вихідному коді.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Коцовський В.М. Супровід програмних систем: Методичний посібник для студентів спеціальності «Інженерія програмного забезпечення» / В. М. Коцовський. – Ужгород: Видавництво УжНУ «Говерла», 2016. – 52 с.

2. Benjamin Floyd, Tyler Santander, and Westley Weimer. 2017. Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise. In Proceedings of the International Conference on Software Engineering (ICSE). 175–186.

3. Якість програмного забезпечення та тестування: базовий курс. Навчальний посібник / За ред. Крепич С.Я., Співак І.Я. – Тернопіль: ФОП Паляниця В.А., 2020. – 478 с.

4. Erin Treacy Solovey, Daniel Afergan, Evan M. Peck, Samuel W. Hincks, and Robert J. K. Jacob. 2015. Designing Implicit Interfaces for Physiological Computing. ACM Transactions on Computer-Human Interaction 21, 6 (2015), 1–27 с.

5. ISO/IEC 9126-1:2001. Software engineering – Software product quality – Part 1: Quality model.

6. Arif, A., Rana, Z.A.: Refactoring of code to remove technical debt and reduce maintenance effort. In: 2020 14th International Conference on Open Source Systems and Technologies, ICOSST 2020. Institute of Electrical and Electronics Engineers Inc. (12 2020).

7. Білас О.Є. Якість программного забезпечення та тестування. Навчальний посібник / О.Є. Білас – Львів: Видавництво Львівської політехніки, 2011. – 216 с.

8. Моргун І.А. Метод експертної оцінки якості програмного забезпечення / І.А. Моргун // Матеріали міжнародної науково-практичної конференції аспірантів і студентів «Інженерія програмного забезпечення», №2(6), 2011. С. 33-37.

9. ISO/IEC TR 9126-2:2003 Software engineering – Product quality – Part2: External metrics.

10. Thomas Fritz, Andrew Begel, Sebastian C Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using Psycho-physiological Measures to Assess Task Difficulty in Software Development. In Proceedings of the International Conference on Software Engineering (ICSE). 402–413.

11. Загальносистемні принципи та етапи створення програм. Життєвий цикл програмного виробу. [Електронний ресурс] – URL: http://lib.mdpu.org.ua/e-book/vstup/L7.htm.

12. Моделі життєвого циклу, принципи і методології розробки програмного забезпечення [Електронний ресурс] – URL: https://evergreens.com.ua/ua/articles/software-development-metodologies.html.

13. Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2009. Relating Identifier Naming Flaws and Code Quality: An Empirical Study. In Proceedings of the Working Conference on Reverse Engineering (WCRE). 31–35 с.

14. Алексенко О.В. Технології програмування та створення програмних продуктів: конспект лекцій / О.В. Алексенко. – Суми: Сумський державний університет, 2013. 133 с.

15. Принципи встановлення вимог розробки системи [Електронний ресурс] – URL: http://org2.knuba.edu.ua/pluginfile.php/28593/mod_resource/content/1/.

16. Agile-маніфест розробки программного забезпечення [Електронний ресурс] – URL: https://agilemanifesto.org/iso/uk/manifesto.html/.

17. Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), pages 286–2861, 2018.

18. Test Deliverables in Software Testing – Detailed Explanation [Electronic resource] – URL: https://www.softwaretestingmaterial.com/testdeliverables/.

19. J. Hofmeister, J. Siegmund, and D. V. Holt. Shorter identifier names take longer to comprehend. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 217–227, 2017.

20. Johnson, J., Lubo, S., Yedla, N., Aponte, J., Sharif, B.: An empirical study assessing source code readability in comprehension. In: Proceedings – 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019. pp. 513–523. IEEE (9 2019).

21. Авраменко А.С. Тестування програмного забезпечення. Навчальний посібник / А. С. Авраменко, В. С. Авраменко, Г. В. Косенюк. – Черкаси: ЧНУ імені Богдана Хмельницького, 2017. – 284 с.

22. M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley Signature Series (Fowler). Pearson Education, 2018.

23. Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In Proceedings of the International Conference on Software Engineering (ICSE). 378-389.

24. Kristien Ooms, Lien Dupont, Lieselot Lapon, and Stanislav Popelka. 2015. Accuracy and precision of fixation locations recorded with the low-cost Eye Tribe tracker in different experimental setups. Journal of eye movement research 8, 1 (2015).

25. Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heuiseok Lim. 2017. Mining biometric data to predict programmer expertise and task difficulty. Cluster Computing (2017), 1-11.

26. Fowler, M., Beck, K.: Refactoring Improving the Design of Existing Code. Addison Wesley, 2nd edn. (2018).

27. Kui Liu, Dongsun Kim, Tegawend´e F. Bissyand´e, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In Proceedings of the 40th International Conference on Software Engineering, ICSE 2019, New York, NY, USA, 2019. ACM.

28. S. L. Abebe and P. Tonella. Automated identifier completion and replacement. In 2013 17th European Conference on Software Maintenance and Reengineering, pages 263–272, March 2013.

29. Ben Liblit, Andrew Begel, and Eve Sweetser. Cognitive perspectives on the role of naming in computer programs. In In Proc. of the 18th Annual Psychology of Programming Workshop, 2006.

30. J. Hofmeister, J. Siegmund, and D. V. Holt. Shorter identifier names take longer to comprehend. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 217-227, 2017.

31. Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the influence of identifier names on code quality: An empirical study. In Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, pages 156-165. IEEE, 2010.

32. Einar W. Host and Bjarte M. Ostvold. Debugging method names. In Sophia Drossopoulou, editor, ECOOP 2009 – Object-Oriented Programming, pages 294–317, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

33. Sarah Fakhoury, Devjeet Roy, Sk. Adnan Hassan, and Venera Arnaoudova. Improving source code readability: Theory and practice. In Proceedings of the 27th International Conference on Program Comprehension, ICPC '19, pages 2–12, Piscataway, NJ, USA, 2019. IEEE Press.

34. Dave Binkley, Matthew Hearn, and Dawn Lawrie. Improving identifier informativeness using part of speech information. In Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, page 203–206, New York, NY, USA, 2011. Association for Computing Machinery.

35. Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014.

36. Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. An approach for evaluating and suggesting method names using n-gram models. In Proceedings of the 22nd International Conference on Program Comprehension, ICPC 2014, page 271–274, New York, NY, USA, 2014. Association for Computing Machinery.

37. db/src/main/java/com/psddev/dari/db/sqldatabase.java.URL: https://github.com/ perfectsense/dari/commit/88e6556.

38. Типове положення про спеціальне навчання, інструктажі та перевірку знань з питань пожежної безпеки на підприємствах, в установах та організаціях України [Електронний ресурс]. – Режим доступу: https://zakon.rada.gov.ua/laws/show/z0308-94#Text.

39. Motivation and Efficiency of Quality Management Systems Implementation: A Study of Lithuanian Organizations [Електронний ресурс]. – Режим доступу: https://www.researchgate.net/publication/264857878_Motivation_and_Efficiency_of_Quality_Management_Systems_Implementation_A_Study_of_Lithuanian_Organizations.

40. Студентський репозитарій. [Електронний ресурс] URL: https://studfile.net/preview/6305127/page:2. https://studfile.net/preview/6305131/.

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД ОСНОВНИХ МОДУЛІВ

Клас ClassBodyEvaluator.java

```
public class StreamInterceptor<T extends Message> implements
TransportFrameDecoder.Interceptor {

private final MessageHandler<T> handler;

private final String streamId;

private final long byteCount;

private final StreamCallback callback;

private long bytesRead;

public StreamInterceptor(

MessageHandler<T> handler,

String streamId,

long byteCount,

StreamCallback callback) {

this.handler = handler;

this.streamId = streamId;

this.byteCount = byteCount;

this.callback = callback;

this.bytesRead = 0;

}

@Override

public void exceptionCaught(Throwable cause) throws Exception {

deactivateStream();

callback.onFailure(streamId, cause);

}

@Override

public void channelInactive() throws Exception {

deactivateStream();

callback.onFailure(streamId, new ClosedChannelException());
```

```
}

private void deactivateStream() {

if (handler instanceof TransportResponseHandler) {

// we only have to do this for TransportResponseHandler as it exposes
numOutstandingFetches

// (there is no extra cleanup that needs to happen)

((TransportResponseHandler) handler).deactivateStream();

}

}

@Override

public boolean handle(ByteBuf buf) throws Exception {

int toRead = (int) Math.min(buf.readableBytes(), byteCount - bytesRead);

ByteBuffer nioBuffer = buf.readSlice(toRead).nioBuffer();

int available = nioBuffer.remaining();

callback.onData(streamId, nioBuffer);

bytesRead += available;

if (bytesRead > byteCount) {

RuntimeException re = new IllegalStateException(String.format(

"Read too many bytes? Expected %d, but read %d.", byteCount, bytesRead));

callback.onFailure(streamId, re);

deactivateStream();

throw re;

} else if (bytesRead == byteCount) {

deactivateStream();

callback.onComplete(streamId);

}

return bytesRead != byteCount;

}

}
```

Клас ClassBodyEvaluator.java

```
public class ClassBodyEvaluator extends Cookable implements IClassBodyEvaluator {

private final SimpleCompiler sc = new SimpleCompiler();

private String[] defaultImports = new String[0];

private String className = IClassBodyEvaluator.DEFAULT_CLASS_NAME;

@Nullable private Class<?> extendedType;
```

```
private Class<?>[] implementedTypes = new Class[0];

@Nullable private Class<?> result; // null=uncooked

@Override public void

setClassName(String className) { this.className = className; }

@Override public void

setDefaultImports(String... defaultImports) { this.defaultImports =
defaultImports.clone(); }

@Override public String[]

getDefaultImports() { return this.defaultImports.clone(); }

@Override public void

setExtendedClass(@Nullable Class<?> extendedType) { this.extendedType =
extendedType; }

@Deprecated @Override public void

setExtendedType(@Nullable Class<?> extendedClass) {
this.setExtendedClass(extendedClass); }

@Override public void

setImplementedInterfaces(Class<?>[] implementedTypes) { this.implementedTypes =
implementedTypes; }

@Deprecated @Override public void

setImplementedTypes(Class<?>[] implementedInterfaces) {
this.setImplementedInterfaces(implementedInterfaces); }


@Override public void

setParentClassLoader(@Nullable ClassLoader parentClassLoader) {
this.sc.setParentClassLoader(parentClassLoader); }

@Override public void

setDebuggingInformation(boolean debugSource, boolean debugLines, boolean
debugVars) {

this.sc.setDebuggingInformation(debugSource, debugLines, debugVars);

}

@Override public void

setSourceVersion(int version) { this.sc.setSourceVersion(version); }

@Override public void

setTargetVersion(int version) { this.sc.setTargetVersion(version); }

@Override public void

setCompileErrorHandler(@Nullable ErrorHandler compileErrorHandler) {

this.sc.setCompileErrorHandler(compileErrorHandler);
```

```
}

@Override public void

setWarningHandler(@Nullable WarningHandler warningHandler) {
this.sc.setWarningHandler(warningHandler); }


@Override public void

cook(@Nullable String fileName, Reader r) throws CompileException, IOException {

if (!r.markSupported()) r = new BufferedReader(r);

this.cook(fileName, ClassBodyEvaluator.parseImportDeclarations(r), r);

}

protected void

cook(@Nullable String fileName, String[] imports, Reader r) throws
CompileException, IOException {

{

StringWriter sw1 = new StringWriter();

{

PrintWriter pw = new PrintWriter(sw1);

String packageName;

String simpleClassName;

{

int idx = this.className.lastIndexOf('.');

if (idx == -1) {

packageName = "";

simpleClassName = this.className;

} else

{

packageName = this.className.substring(0, idx);

simpleClassName = this.className.substring(idx + 1);

}

}

if (!packageName.isEmpty()) {

pw.print("package ");

pw.print(packageName);

pw.println(";");

}

for (String defaultImport : this.defaultImports) {

pw.print("import ");
```

```
pw.print(defaultImport);
pw.println(";");
}
if (!r.markSupported()) r = new BufferedReader(r);
for (String importT : imports) {
pw.print("import ");
pw.print(importT);
pw.println(";");
}

// Print the class declaration.
pw.print("public class ");
pw.print(simpleClassName);

{
Class<?> oet = this.extendedType;
if (oet != null) {
pw.print(" extends ");
pw.print(oet.getCanonicalName());
}
}

if (this.implementedTypes.length > 0) {
pw.print(" implements ");
pw.print(this.implementedTypes[0].getCanonicalName());
for (int i = 1; i < this.implementedTypes.length; ++i) {
pw.print(", ");
pw.print(this.implementedTypes[i].getCanonicalName());
}
}
pw.println(" {");
pw.close();
}

StringWriter sw2 = new StringWriter();
{
PrintWriter pw = new PrintWriter(sw2);
pw.println("}");
pw.close();
}
```

```
r = Readers.concat(
new StringReader(sw1.toString()),
this.newFileName(fileName, r),
new StringReader(sw2.toString())
);
}

this.sc.cook(fileName, r);

try {

this.result = this.sc.getClassLoader().loadClass(this.className);
} catch (ClassNotFoundException cnfe) {
throw new IOException(cnfe);
}
}

@Override public Class<?>
getClazz() {
assert this.result != null;
return this.result;
}

@Override public Map<String, byte[]>
getBytecodes() { return this.sc.getBytecodes(); }

protected Reader
newFileName(@Nullable final String fileName, Reader reader) {
return Readers.onFirstChar(reader, new Runnable() {
@Override public void run() { ClassBodyEvaluator.this.sc.addOffset(fileName); }
});
}

protected static String[]
parseImportDeclarations(Reader r) throws IOException {
final CharBuffer cb = CharBuffer.allocate(10000);
r.mark(cb.limit());
r.read(cb);

((Buffer) cb).rewind();

List<String> imports = new ArrayList<>();
int afterLastImport = 0;
```

```
for (Matcher matcher = ClassBodyEvaluator.IMPORT_STATEMENT_PATTERN.matcher(cb);
matcher.find();) {

imports.add(matcher.group(1));

afterLastImport = matcher.end();

}

r.reset();

r.skip(afterLastImport);

return imports.toArray(new String[imports.size()]);

}

private static final Pattern IMPORT_STATEMENT_PATTERN = Pattern.compile(

"\\bimport\\s+"

+ "("

+ "(?:static\\s+)?"

+
"[\\p{javaLowerCase}\\p{javaUpperCase}_\\$][\\p{javaLowerCase}\\p{javaUpperCase}\\
d_\\$]*"

+
"(?:\\.[\\p{javaLowerCase}\\p{javaUpperCase}_\\$][\\p{javaLowerCase}\\p{javaUpperC
ase}\\d_\\$]*)*"

+ "(?:\\.\\*)?"

+ ");"

);


@Override public Object

createInstance(Reader reader) throws CompileException, IOException {

this.cook(reader);

try {

return this.getClazz().getConstructor().newInstance();

} catch (InstantiationException ie) {

throw new CompileException((

"Class is abstract, an interface, an array class, a primitive type, or void; "

+ "or has no zero-parameter constructor"

), null, ie);

} catch (Exception e) {

throw new CompileException("Instantiating \"" + this.getClazz().getCanonicalName()
+ "\"", null, e);

}

}

}
```

Клас ClassBodyEvaluator.java

```
public class Compiler extends AbstractCompiler {

private Collection<String> compilerOptions = new ArrayList<>();

private final JavaCompiler compiler;

public
Compiler() {
JavaCompiler c = ToolProvider.getSystemJavaCompiler();
if (c == null) {
throw new RuntimeException(
"JDK Java compiler not available - probably you're running a JRE, not a JDK",
null
);
}

this.compiler = c;
}
public
Compiler(JavaCompiler compiler) { this.compiler = compiler; }

@Override public void
setVerbose(boolean verbose) {}


public void
setCompilerOptions(String[] compilerOptions) { this.compilerOptions =
Arrays.asList(compilerOptions); }

@Override public void
compile(final Resource[] sourceResources) throws CompileException, IOException {
this.compile(sourceResources, null);
}

public void
compile(final Resource[] sourceResources, @Nullable SortedSet<Location> offsets)
throws CompileException, IOException {

List<String> options = new ArrayList<>(this.compilerOptions);


{
List<String> l = new ArrayList<>();
if (this.debugLines) l.add("lines");
```

```
if (this.debugSource) l.add("source");
if (this.debugVars) l.add("vars");
if (l.isEmpty()) l.add("none");

Iterator<String> it = l.iterator();
String o = "-g:" + it.next();
while (it.hasNext()) o += "," + it.next();

options.add(o);
}

{
if (this.sourceVersion != -1) {
options.add("-source");
options.add(Integer.toString(this.sourceVersion));
}
if (this.targetVersion != -1) {
options.add("-target");
options.add(Integer.toString(this.targetVersion));
}
}

{
File[] bcp = this.bootClassPath;
if (bcp != null) {
options.add("-bootclasspath");
options.add(Compiler.filesToPath(bcp));
}
}

options.add("-classpath");
options.add(Compiler.filesToPath(this.classPath));

Compiler.compile(
this.compiler,
options,
this.sourceFinder,
this.sourceCharset,
this.classFileFinder,
this.classFileCreator,
sourceResources,
```

```
this.compileErrorHandler,
this.warningHandler,
offsets
);
}

static void
compile(
JavaCompiler compiler,
List<String> options,
ResourceFinder sourceFinder,
Charset sourceFileCharset,
ResourceFinder classFileFinder,
ResourceCreator classFileCreator,
Resource[] sourceFiles,
@Nullable ErrorHandler compileErrorHandler,
@Nullable WarningHandler warningHandler,
@Nullable SortedSet<Location> offsets
) throws CompileException, IOException {

Collection<JavaFileObject> sourceFileObjects = new ArrayList<>();
for (int i = 0; i < sourceFiles.length; i++) {
Resource sourceResource = sourceFiles[i];

String fn = sourceResource.getFileName();
String className = fn.substring(fn.lastIndexOf(File.separatorChar) +
1).replace('/', '.');
if (className.endsWith(".java")) className = className.substring(0,
className.length() - 5);
sourceFileObjects.add(JavaFileObjects.fromResource(
sourceResource,
className,
Kind.SOURCE,
sourceFileCharset
));
}

final JavaFileManager fileManager = Compiler.getJavaFileManager(
compiler,
sourceFinder,
sourceFileCharset,
classFileFinder,
```

```
classFileCreator
);
try {
Compiler.compile(
compiler,
options,
sourceFileObjects,
fileManager,
compileErrorHandler,
warningHandler,
offsets
);
} finally {
fileManager.close();
}
}
private static JavaFileManager
getJavaFileManager(
JavaCompiler compiler,
ResourceFinder sourceFileFinder,
Charset sourceFileCharset,
ResourceFinder classFileFinder,
ResourceCreator classFileCreator
) {

JavaFileManager jfm = compiler.getStandardFileManager(null, null, null);
jfm = JavaFileManagers.fromResourceCreator(
jfm,
StandardLocation.CLASS_OUTPUT,
Kind.CLASS,
classFileCreator,
Charset.defaultCharset()
);

jfm = JavaFileManagers.fromResourceFinder(
jfm,
StandardLocation.CLASS_PATH,
Kind.CLASS,
classFileFinder,
```

```
Charset.defaultCharset() // irrelevant
);

jfm = JavaFileManagers.fromResourceFinder(
jfm,
StandardLocation.SOURCE_PATH,
Kind.SOURCE,
sourceFileFinder,
sourceFileCharset
);

return jfm;
}

static void
compile(
JavaCompiler compiler,
List<String> options,
Collection<JavaFileObject> sourceFileObjects,
JavaFileManager fileManager,
@Nullable final ErrorHandler compileErrorHandler,
@Nullable final WarningHandler warningHandler,
@Nullable final SortedSet<Location> offsets
) throws CompileException, IOException {

fileManager = (JavaFileManager) ApiLog.logMethodInvocations(fileManager);

final int[] compileErrorCount = new int[1];

final DiagnosticListener<JavaFileObject> dl = new
DiagnosticListener<JavaFileObject>() {

@Override public void
report(@Nullable Diagnostic<? extends JavaFileObject> diagnostic) {
assert diagnostic != null;

JavaFileObject source = diagnostic.getSource();
Location loc = new Location(
( // fileName
source == null ? null :
source instanceof ResourceJavaFileObject ? ((ResourceJavaFileObject)
source).getResourceFileName() :
source.getName()
),
```

```
(short) diagnostic.getLineNumber(),
(short) diagnostic.getColumnNumber()
);
if (offsets != null) {
SortedSet<Location> hs = offsets.headSet(loc);
if (!hs.isEmpty()) {
Location co = hs.last();
loc = new Location(
co.getFileName(),
loc.getLineNumber() - co.getLineNumber() + 1,
(
loc.getLineNumber() == co.getLineNumber()
? loc.getColumnNumber() - co.getColumnNumber() + 1
: loc.getColumnNumber()
)
);
}
}
String message = diagnostic.getMessage(null) + " (" + diagnostic.getCode() + ")";
try {
switch (diagnostic.getKind()) {
case ERROR:
compileErrorCount[0]++;
if (compileErrorHandler == null) throw new CompileException(message, loc);
compileErrorHandler.handleError(diagnostic.toString(), loc);
break;
case MANDATORY_WARNING:
case WARNING:
if (warningHandler != null) warningHandler.handleWarning(null, message, loc);
break;
case NOTE:
case OTHER:
default:
break;
}
```

```
} catch (CompileException ce) {
throw new RuntimeException(ce);
}
}
};

// Run the compiler.
try {
if (!compiler.getTask(
null, // out
fileManager, // fileManager
dl, // diagnosticListener
options, // options
null, // classes
sourceFileObjects // compilationUnits
).call() || compileErrorCount[0] > 0) {
throw new CompileException("Compilation failed with " + compileErrorCount[0] + "
errors", null);
}
} catch (RuntimeException rte) {

for (Throwable t = rte.getCause(); t != null; t = t.getCause()) {
if (t instanceof CompileException) {
throw (CompileException) t; // SUPPRESS CHECKSTYLE AvoidHidingCause
}
if (t instanceof IOException) {
throw (IOException) t; // SUPPRESS CHECKSTYLE AvoidHidingCause
}
}
throw rte;
}
}

private static String
filesToPath(File[] files) {
StringBuilder sb = new StringBuilder();
for (File cpe : files) {
if (sb.length() > 0) sb.append(File.pathSeparatorChar);
sb.append(cpe.getPath());
}
return sb.toString();
```

ДОДАТОК Б

(обов’язковий)

КОПІЯ НАУКОВОЇ ПУБЛІКАЦІЇ

УДК 004.9

Стьопич В.В.

Хмельницький національний університет

ОЦІНЮВАННЯ ІМЕН ІДЕНТИФІКАТОРІВ ВИХІДНОГО ПРОГРАМНОГО КОДУ

Розглянуто задачу оцінювання імен ідентифікаторів коду, що дозволяють здійснювати правильне оформлення програмного коду згідно правил та стандартів. Здійснено аналіз доцільності оцінки якості найменування ідентифікаторів коду програмного забезпечення в сучасних реаліях роботи розробника.

The task of evaluating the names of the code identifiers, which allow for the correct design of the software code according to the rules and standards, is considered. An analysis of the feasibility of assessing the quality of the naming of software code identifiers in the modern realities of the developer's work was carried out.

Значну частину вихідного коду складають імена ідентифікаторів – унікальні лексичні токени, які надають інформацію про сутності та взаємодію сутностей у коді. Імена ідентифікаторів надають зрозумілі людині описи класів, функцій, змінних тощо [1]. Неякісні або неоднозначні імена ідентифікаторів (тобто імена, які неправильно описують поведінку коду, з якою вони пов'язані), змушують розробників витрачати набагато більше часу аби зрозуміти поведінку коду. Погані та неякісні найменування також можуть мати шкідливий вплив на інструменти, які покладаються на підказки природної мови, що впливає на погіршення якості та робить їх ненадійними. Крім того, спричинені неправильним тлумаченням коду погані та невідповідні назви, можуть призвести до появи проблем з якістю в системі, що обслуговується.

Вміння правильно оформляти програмний код – одна з професійних здібностей розробника програмного забезпечення. Важливо не тільки знати та вміти застосовувати конструкції мови програмування, писати ефективні за часом програми, використовувати алгоритмічні прийоми та стандартні шаблони програмування, а й «упаковувати» все це в коректну та ефективну текстову оболонку. Уніфікація та раціональна структура коду покращують читабельність та зрозумілість програм, прискорюють тестування та налагодження, спрощують взаєморозуміння та взаємодію у групі розробників.

Таким чином, дотримання єдиних правил оформлення робиться не так для надання текстам програм красивого зовнішнього вигляду, як для підвищення продуктивності розробки та якості програмного продукту. Для різних мов програмування існує різна історія формування та прийняття правил оформлення вихідного коду. У більшості випадків єдиних загальносвітових вимог та рекомендацій не існує, але складаються певні традиції та неформальні угоди у спільнотах розробників, авторами книг та статей використовуються загальноприйняті правила кодування [1].

Ключовим елементом підтримки програмного забезпечення є розуміння програми. Розуміння програми це дії розробників, які читають вихідний код, щоб зрозуміти призначення коду або визначити вимоги, пов'язані з їх діяльністю з обслуговування. До внесення змін до коду щоб полегшити розвиток програмної системи, розробникам необхідно читати рядки коду у вихідному коді файлу для того, що розуміти поведінку коду. Ясно, що такі питання як погана читабельність і зрозумілість коду впливають не тільки на час, який витрачають розробники при виконанні своїх завдань, але також можуть опосередковано чи напряму впливати на якість оновлень, що виконуються в системі.

Крім того, відмінності у коді (наприклад, навички, досвід) між оригінальним авторським кодом і супроводжуючою документацією також впливають на розуміння. Зі всього часу розробника 58% витрачається на діяльність по розумінню. Тому важливо аби розробники створили вихідний код таким чином, щоб він не перешкоджав читабельності та зрозумілості. Це включає вдосконалення коду, починаючи від змін рівня проектування, таких як зменшення цикломатичної складності [2, 3] до відповідної узгодженості, у формі домовленостей про найменування.

Будучи фундаментальними елементами у вихідному коді, імена ідентифікаторів складають майже 70% символів у кодовій частині програмної системи. Ці назви є лексемами, які однозначно ідентифікують сутності в коді (наприклад, класи, методи, змінні тощо) і відіграють значну роль у розумінні коду. Важливість та необхідність якісних та хороших імен ідентифікаторів визнають як розробники так і наукові кола. Ця важливість відображається в теоретичних аспектах розробки програмного забезпечення, які забезпечують рекомендації, найкращі практики, показники та моделі, щоб допомогти розробникам в іменуванні ідентифікаторів із загальною метою покращення розуміння коду [2]. Тому розробники повинні приділяти значну увагу створенню назви ідентифікатора, оскільки їхній вибір впливає на час, витрачений на розуміння мети ідентифікатора, а добре побудовані імена можуть покращити діяльність розуміння аж на 19% [4].

Хоча показники якості, найкращі практики та рекомендації підкреслюють потребу у високоякісному ідентифікаторі найменування, вони діють лише як теоретичні аспекти для розробників; вони не є формальними механізмами використання адекватних та потрібних імен. Ці механізми не можуть допомогти розробникам використовувати правильні формулювання, рекомендувати лексичну структуру за межами того, що є методично та теоретично правильним (наприклад, правила іменування).

Отже, оцінка найменування ідентифікаторів програмного коду згідно правил та стандартів має ключове значення при розробці програмного забезпечення. Подальші дослідження спрямовані на вдосконалення алгоритму та методу.

Перелік посилань

1. Коцовський В.М. Супровід програмних систем: Методичний посібник для студентів спеціальності «Інженерія програмного забезпечення» / В. М. Коцовський. – Ужгород: Видавництво УжНУ «Говерла», 2016. – 52 с.
2. Benjamin Floyd, Tyler Santander, and Westley Weimer. 2017. Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise. In Proceedings of the International Conference on Software Engineering (ICSE). 175-186.
3. Якість програмного забезпечення та тестування: базовий курс. Навчальний посібник / За ред. Крепич С.Я., Співак І.Я. – Тернопіль: ФОП Паляниця В.А., 2020. – 478 с.
4. Erin Treacy Solovey, Daniel Afergan, Evan M. Peck, Samuel W. Hincks, and Robert J. K. Jacob. 2015. Designing Implicit Interfaces for Physiological Computing. ACM Transactions on Computer-Human Interaction 21, 6 (2015), 1–27.
5. ISO/IEC 9126-1:2001. Software engineering – Software product quality – Part 1: Quality model.

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Метод оцінки якості найменування ідентифікаторів коду програмного забезпечення

Автор роботи:
студент групи ІПЗм-21-1 Стьопич В.В.
Керівник роботи:
д. фіз-мат. наук, проф, Бедратюк Л. П.

Актуальність

Значну частину вихідного коду складають найменування (імена) ідентифікаторів – унікальні лексичні токени, які надають інформацію про сутності та взаємодію сутностей у коді. Імена ідентифікаторів надають зрозумілі людині описи класів, функцій, змінних тощо. Неякісні або неоднозначні імена ідентифікаторів (тобто імена, які неправильно описують поведінку коду, з якою вони пов'язані), змушують розробників витрачати більше часу на роботу над розумінням поведінки коду. Неякісні імена також можуть мати шкідливий вплив на інструменти, які покладаються на підказки природної мови; погіршення якості їхньої продукції та робить їх ненадійними. Крім того, спричинені неправильним тлумаченням коду неякісні та невідповідні назви, можуть призвести до появи проблем з якістю в системі, що обслуговується.

Таким чином, покращене найменування ідентифікаторів підвищує ефективність роботи самого розробника, і в цілому покращує та підвищує якість програмного забезпечення та якісніші засоби аналізу програмного забезпечення.

Провівши грунтовний аналіз останніх досліджень та публікацій було виявлено та вивчено підходи таких вітчизняних та зарубіжних науковців:

•Лавріщєва К.М., яка займалась питаннями програмної інженерії та якістю програмного забезпечення зокрема;

•Коцовський В.М. розглядав супровід програмних систем в контексті якості.

•Лю та інші пропонують автоматизований підхід, заснований на методі глибокого навчання для налагодження назви на основі узгодженості між назвою методу та його реалізацією.

•Лібліт та ін. здійснили обговорення іменування в кількох мовах програмування та роблять зауваження щодо впливу природної мови на використання слів у цих мовах.

•Хофмайстер та ін. здійснили порівняння розуміння ідентифікаторів, що містять слова, проти ідентифікаторів, що містять літери та/або скорочення.

•Хост і Освольд розробляють автоматичні правила іменування, використовуючи елементи підпису методу, тобто тип повернення, імена параметрів і типи, а також потік керування.

•Батлер та ін. розширюють свої попередні праці над ідентифікаторами класів Java, щоб показати, що ідентифікатори методів з недоліками також є (тобто разом з ідентифікаторами класів), пов'язані з низькоякісним кодом відповідно до показників на основі статичного аналізу.

Мета: удосконалення методу оцінки якості найменування ідентифікаторів вихідного коду програмних систем та розробка відповідного алгоритму граматичних шаблонів покращення якості.

Об'єкт дослідження: процес створення концепції вимірювання та покращення якості ідентифікаторів.

Предмет дослідження: якість найменування ідентифікаторів вихідного коду програмної системи.

Задачі дослідження:

1. Здійснити аналіз предметної області.
2. Проаналізувати існуючі методи оцінки якості найменування ідентифікаторів вихідного коду програмної системи.
3. Здійснити удосконалення методу оцінки якості найменування ідентифікаторів вихідного коду програмних систем.
4. Розробити алгоритм граматичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи із наданням рекомендацій щодо структури імен.

Наукова новизна

1. Удосконалено метод оцінки якості найменування ідентифікаторів вихідного коду програмних системи.
2. Розроблено алгоритм граматичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи із наданням рекомендацій щодо структури імен.
3. На основі запропонованого підходу здійснено розробку програмного забезпечення для реалізації алгоритму лінгвістичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи.

Практична значимість

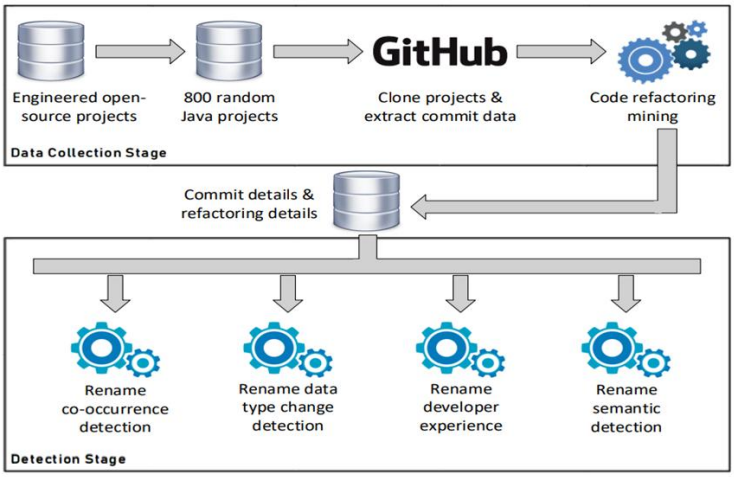
Практична значимість отриманих результатів полягає у тому, що отримані результати магістерської роботи можуть:

- допомогти розробникам не лише створювати якісні імена ідентифікаторів, але також краще керувати ідентифікаторами у вихідному коді їхніх проектів;
- бути використаними для підвищення якості програмного коду в цілому.

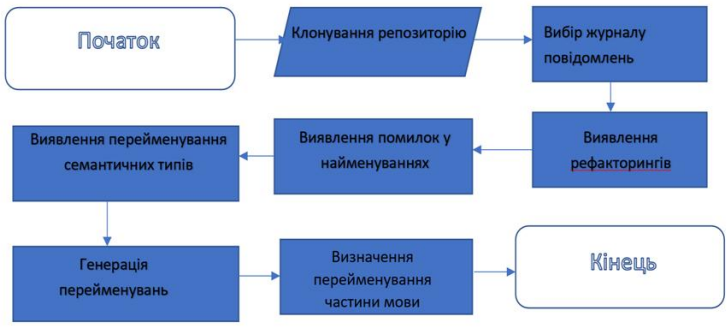
Публікації за темою роботи

За темою дипломної роботи опубліковано тези доповіді конференції АПКН-2022 Хмельницького національного університету «Оцінювання імен ідентифікаторів вихідного програмного коду»

Проведення дослідження науковцями



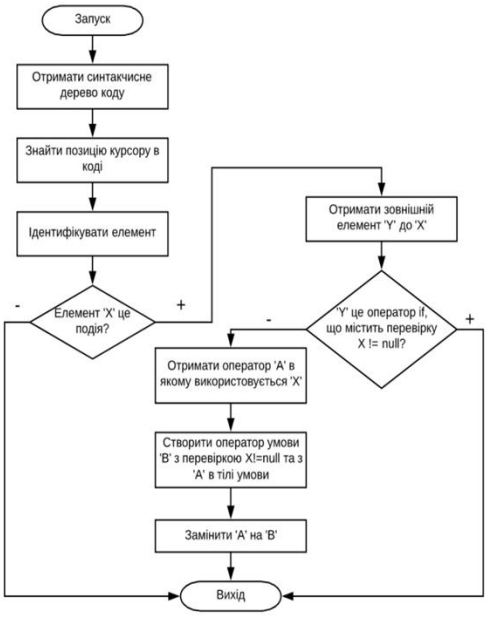
Концептуальна модель



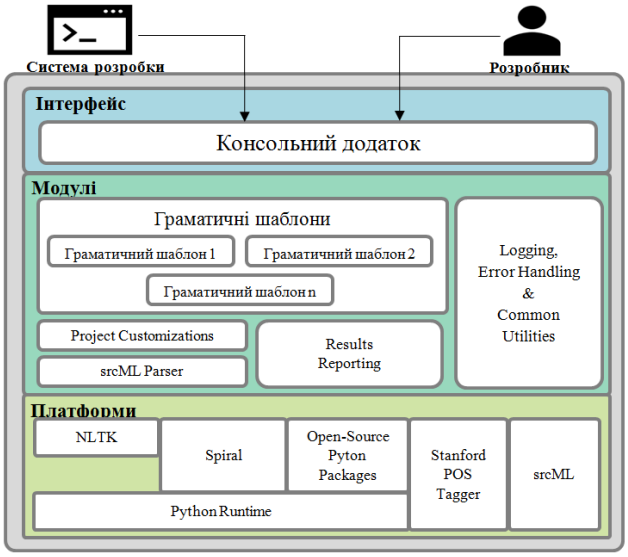
Метод оцінки якості найменування ідентифікаторів

Метод оцінки якості найменування ідентифікаторів полягає у тому, що спочатку формується відповідна таблиця ідентифікаторів, потім здійснюється пошук елемента, тобто найменування ідентифікатора Це здійснюється за допомогою методу бінарного пошуку, тобто формується бінарне дерево. Після чого, якщо найменування ідентифікатора не відповідає вимогам, то підключаються граматичні шаблони і відбувається заміна на потрібний.

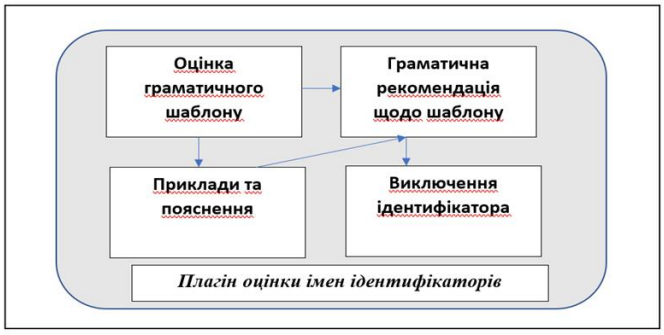
Алгоритм граматичних шаблонів



Архітектура програмного забезпечення



Результати роботи



Висновки:

1. Здійснено аналіз предметної області.
2. Проаналізовано існуючі методи оцінки якості найменування ідентифікаторів вихідного коду програмної системи.
3. Здійснено удосконалення методу оцінки якості найменування ідентифікаторів вихідного коду програмних систем.
4. Розроблено алгоритм граматичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи із наданням рекомендацій щодо структури імен.

Дякую за увагу!

Завідувачу кафедри інженерії програмного
забезпечення проф. Бедратюку Л. П.
здобувача вищої освіти
Стьопича В. В.
факультет ФІТ, 2 курс, група ІПЗм-21-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

29.11.2022 р.
дата

підпис

Mon Dec 05 11:19:12 EET 2022, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 19.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 10%**

	Документ		Сумарний збіг по Базі Даних	
ID: 108937 Назва: КРМ на тему: «Метод оцінки якості найменування ідентифікаторів коду програмного забезпечення» Додано в БД: 2022-12-05 Автора: Стьопич В.В. Керівники: Бедратюк Л.П. Консультанти: Опоненти:	Символи	Лексеми	Символи	Лексеми
	121389	903	24528 (20%)	168 (19%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
107861	Назва: Звіт з переддипломної практики Додано в БД: 2022-10-05 Автора: В.В. Стьопич Керівники: Бедратюк Л. П Консультанти: Опоненти:	22934 (19.0%)	158 (17.0%)

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1013180187

Дата перевірки:
05.12.2022 08:34:51 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
05.12.2022 08:35:36 EET

ID користувача:
100005589

Назва документа: **Стьопич В.В., магістерська**

Кількість сторінок: **76** Кількість слів: **16272** Кількість символів: **128222** Розмір файлу: **6.64 MB** ID файлу: **1012944933**

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

5.01%
Схожість

Найбільша схожість: **2.32%** з джерелом з Бібліотеки (ID файлу: **1012913624**)

2.22% Джерела з Інтернету	54	Сторінка 78
3.34% Джерела з Бібліотеки	125	Сторінка 78

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	46
Підозріле форматування	63 сторінки

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Метод оцінки якості найменування ідентифікаторів коду програмного забезпечення»
Автор: Стьопич Владислав Валерійович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Бедратюк Леонід Петрович, доктор фіз.-мат. наук, професор
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) у тексті дипломної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, назвах розділів/підрозділів тощо та в назвах публікацій у переліку джерел посилання;

2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посилання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає 5,01% і адресується до 54 джерел з Інтернету та 125 джерел із бібліотеки, що, з урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь дипломної роботи.

Керівник — Л. П. Бедратюк

Гарант ОП — О. М. Яшина

Завідувач кафедри — Л. П. Бедратюк

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

Магістр Стьопич Владислав Валерійович

Тема Метод оцінки якості найменування ідентифікаторів коду програмного забезпечення

Спеціальність 121 « Інженерія програмного забезпечення»

Обсяг дипломної роботи:

Кількість сторінок записки 88.

1.Короткий зміст ДР та прийнятих рішень У дипломній роботі проаналізовано існуючі методи оцінки якості найменування ідентифікаторів вихідного коду програмної системи. Здійснено удосконалення методу оцінки якості найменування ідентифікаторів вихідного коду програмних систем. На основі розробленої моделі імплементовано алгоритм граматичних шаблонів покращення якості найменування ідентифікаторів вихідного коду програмної системи із наданням рекомендацій щодо структури імен.

2. Висновок про відповідність ДР поставленому завданню Дипломна робота освітнього ступеня «магістр» у повній мірі відповідає поставленому завданню як у теоретичній, так і в практичній її частині.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У вступі обгрунтовується актуальність теми роботи, формулюються цілі і завдання дослідження, описується наукова новизна та практична значимість отриманих результатів. У першому розділі охарактеризовано структуру предметної області та аналіз існуючих рішень до оцінки імен ідентифікаторів. Виконана розгорнута постановка задачі. У другому розділі описано концептуальну модель процесу оцінки якості імен ідентифікаторів. У третьому розділі аналізовано та сформовано вимоги до програмного забезпечення. Наведені проектні рішення, що дають змогу реалізувати технічні вимоги. У четвертому розділі розглянуто питання, що стосуються реалізації програмної системи на основі прийнятих проектних рішень, а також її технічні та технологічні характеристики. Також проведено емпіричне дослідження, спрямоване на доведення працездатності розробленої системи та її функціональної придатності.

4. Позитивні сторони роботи Дипломна робота покликана вирішити дійсно важливу проблему контролю якості. Було запропоновано покращення методу оцінки якості найменувань ідентифікаторів за допомогою підключення граматичних шаблонів, що збільшило якість розроблюваного програмного забезпечення.

5. Негативні сторони роботи Роботі не вистачає більш наглядного зображення результатів роботи програмного забезпечення з розробленим алгоритмом оцінки найменувань ідентифікаторів з використанням граматичних шаблонів.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми дипломної роботи з дотриманням вимог стандартів. Пояснювальна записка відповідає вимогам стандартів до її оформлення

7. Відгук про роботу в цілому В цілому дипломна робота заслуговує позитивної оцінки. Весь матеріал дипломної роботи структурований, чіткий та послідовний. Усі розділи роботи є послідовними та логічними, що дозволяє чітко розуміти викладений матеріал у рамках тематики дипломної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті за основу для вирішення поставленої задачі.

8. Інші зауваження

9. Оцінка дипломної роботи Розглянувши позитивні та негативні сторони представленої дипломної роботи, можна зробити висновок, що вона заслуговує оцінки «добре».

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи)
Чешун Віктор Миколайович
к.т.н., доцент
доцент кафедри кібербезпеки

"2" 12 2022 р. (підпис)