DOI 10.31891/2307-5732-2021-297-3-39-45
УДК 004.056.5:621

Н. І. ПРАВОРСЬКА, О. В. БАРМАК, Д. М. МЕДЗАТИЙ
Хмельницький національний університет
Т. В. ШЕСТАКЕВИЧ
Львівська політехніка

ПРОЦЕС ВИЯВЛЕННЯ БЛОКІВ З ПОВТОРАМИ І НАДЛИШКОВІСТЮ ПРИ ВИКОРИСТАННІ МОВНО-НЕЗАЛЕЖНОГО ІНКРЕМЕНТНОГО ДЕТЕКТОРА

Для уникнення порушень нормального функціонування розробленого програмного забезпечення, спричиненого помилками, навіть коли розробкою займаються професіонали, використовується ряд автоматизованих інструментів, які дають змогу проводити оцінювання програмного коду. Для виявлення помилок, які з'являються через дублювання блоків виконуваного коду, зазвичай застосовують різноманітні детектори. Важливість при розробці подібних детекторів полягає в тому, щоб продукт не був залежним від мови програмування та мав нескладний алгоритм знаходження клонованих блоків коду. В основі підходу мовно-незалежного детектора повторів покладено метод, який базується на використанні індексу клону. Він представляє собою глобальну структуру даних, яка нагадує типовий інвертований індекс. За основу такого підходу береться текст, тобто метод стає базою для досліджень незалежних від мови.

Ключові слова: програмний код, мовно-незалежний детектор, інкрементний підхід, робочий процес, індекс повторення, індекс клону, хеш-функція, хеш-значення, коміт, репозиторій.

N. I. PRAVORSKA, O. B. BARMAC, D. M. MEDZATIY
Khmelnytsky National University
T. V. SHESTAKEVYCH
Lviv Polytechnic

THE PROCESS OF DETECTING BLOCKS WITH REPETITIONS AND EXCESS BUILDING USING A LANGUAGE-INDEPENDENT INCREASE DETECTOR

To avoid malfunctions of the developed software caused by errors, even when developed by professionals, a number of automated tools are used, which allow to evaluate the software code. A variety of detectors are commonly used to detect errors that occur due to duplicate blocks of executable code. The importance of developing such detectors is that the product is not dependent on the programming language and has a simple algorithm for finding cloned blocks of code. The approach of the language-independent repetition detector is based on a method based on the use of the clone index. It is a global data structure that resembles a typical inverted index. This approach is based on the text, ie the method becomes the basis for research independent of language. In recent years, additional methods have become increasingly popular, which analyze the source and executable code at a smaller level, and there are attempts to avoid unnecessary recalculations, by transferring information between versions.

Reviewing the research presented in the works of scientists dealing with this problem, it was decided to propose an approach to improve methods for detecting repetitions and redundancy of program code based on language-independent incremental repetition detector (MNIDP). Most additional research is based on tree-like and graphical methods, ie they are strictly dependent on the programming language. The solution in the MNIDP campaign is to take the text as a basis, ie the method becomes the basis for research independent of language. This technique is not strictly language-independent, but due to the fact that the tokenization stage will be included, with the help of minor adjustments the desired result has been achieved. This provides a detailed analysis of the internal composition (namely, elements) of the detector and explanations of the work at different stages of the detection process.

Keywords: program code, language-independent detector, incremental approach, workflow, iteration index, clone index, hash function, hash value, commit, repository.

Постановка проблеми

Порушення нормального режиму функціонування розробленого програмного забезпечення іноді можуть виникнути при допущенні помилок навіть, коли цим займаються фахівці професіонали. Незважаючи на це, правки можна внести на любому з етапів життєвого циклу програмного продукту. Однак, якщо цей процес буде тривати до останніх етапів розробки ПЗ, то витрати як самої розробки, так і супроводу (а саме фінансові і часові) можуть стати дуже великими. При цьому треба враховувати, що при експлуатації помилки у програмному забезпеченні, яке використовується в деяких важливих людських сферах (наприклад, медицині, транспорті та тому подібне) можуть становити небезпеку життю та здоров'ю людини. Тому розробники, зазвичай, при розробці програмних продуктів активно використовують інструменти, спроможні проаналізувати код та виявити дефекти на ранніх стадіях життєвого циклу.

При розборі робіт, які є на сьогодні актуальними в питаннях аналізу виконуваного коду, спостерігається наступна тенденція, проводиться розбиття аналізу на статичний та динамічний. Кожний з вище згаданих підходів має як свої переваги, так і недоліки. При статичному аналізі треба використовувати всі можливі шляхи виконання та всі значення змінних. Таким чином, статичній аналіз має змогу виявити дефекти на відміну від динамічного аналізатора, навіть, якщо такий використовувався доволі довгий час. Тобто динамічний аналізатор може виявляти подібне лише в тому разі, якщо виконуваний шлях пройшов через точку дефекту, при деяких значеннях змінних. На відміну від динамічних аналізаторів статичним, зазвичай, не треба інформації про вхідні дані. Окрім цього, перші потребують використання емуляторів чи апаратури, якій притаманна архітектура виконуваного коду, який підпадає під аналіз.

Левова частка сучасних методів аналізу виконуваного коду, використовують звичайний спосіб в підході виявлення блоків з повторами та надлишковостями. Однак, в останні роки, все більше набувають

популярності додаткові методи, які проводять аналіз вихідного та виконуваного коду на більш дрібному рівні, причому є намагання уникнути зайвих перерахунків, завдяки передачі інформації між версіями. Незважаючи на підходи, методи класифікують за представленням вихідного коду на такі основні категорії:

– текстові підходи;
– підходи, основані на токенах;
– синтаксичні підходи;
– семантичні підходи;
– підходи, засновані на поведінці програми.

На відміну від досліджень, які використовують інструменти повного системного аналізу, розробок, що пропонують додаткові методи є небагато. Переважно вони зосереджуються на підходах, основою яких становлять токени, дерева або графіки. Це буде вимагати від синтаксичного аналізатора або парсера проводити побудову необхідних структур даних спираючись на відповідну мову (мови) програмування системи. Фактично з самого початку більшість з них не мають намір підтримувати незалежність від мови програмування. Таким чином вони розробляють детектори, які з одного боку спроможні виявляти більш складні типи клонованого коду (блоків коду з повторами та надлишковістю), але з іншого потребують налаштування для конкретної мови програмування.

Аналіз останніх джерел

Натомість, вперше була запропонована інкрементна методика виявлення клонів авторами роботи Гоуд та Кошке [1]. Для представлення вихідного коду було рекомендовано задіяти деревоподібну техніку, яка використовує глобальну структуру даних узагальненого суфіксного дерева (УСД). Подібна техніка на базі дерева ClemanX пропонувалася у праці Нгуена та ін. [2]. Тут кожний вихідний файл представляється як АСД (абстрактне синтаксичне дерево). Для забезпечення співставлення подібності кожне піддерево такого АСД, додатково представлялося характеристичним вектором.

Згодом було запропоновано додаткове виявлення блоків з повторами і надлишковістю коду у праці Хіго та ін. [3] на основі ГЗП (графів залежності програми). В базі даних проводиться збереження ГЗП кожного методу кожного оновленого файлу, який побудовано на етапі аналізу. На наступному кроці виявлення, після ручного введення користувачем запускається вибірка відповідних графів ЗП з бази даних, які є основою для виявлення клонів. Інший алгоритм, який базується на основі індексу, запропонований в роботі Хаммела та ін. [4]. Основною структурою даних, яка використовується, виступає індекс клону. Він представляє собою глобальну структуру даних, яка нагадує типовий інвертований індекс. Цей метод, хоча і використовує лексичний аналізатор для перетворення вихідного коду в токени, але серед всієї літератури з цього питання є найближчим до незалежного від мови інкрементного підходу.

Проводячи огляд досліджень, викладених в роботі Хаммела та ін. [4] було вирішено запропонувати підхід удосконалення методів виявлення повторів та надлишковості програмного коду на основі мовно-незалежного інкрементного детектора повторів (МНІДП). Більшість додаткових досліджень, за основу мають деревоподібні та графічні методи, тобто вони суворо залежать від мови програмування. Рішенням в поході МНІДП – є взяття за основу тексту, тобто метод стає базою для досліджень незалежних від мови. Ця методика не являється сугубо мовно-незалежною, але через те, що буде включений етап токенізації, за допомогою незначних корегувань досягнуто потрібного результату. Вирішено провести деякі зміни, які забезпечили мовну незалежність і того, як вони відобразилися в запропонованому детекторі МНІДП. При цьому надається деталізація аналізу внутрішнього складу (а саме, елементів) детектора та пояснень роботи на різних етапах процесу виявлення.

Огляд мовно-незалежного інкрементного детектора повторів. Проводиться поділ інкрементного методу даного дослідження на два основних робочих процеси. В свою чергу кожний буде включати в себе додатково ряд підпроцесів. Проміжне представлення пов'язується з першим процесом і зберігається для повторного використання в версіях проекту. Таке об'єднання (сектор, pool) буде прийматися за індекс повторення (Clone Index – індекс клону). Весь проект програмного забезпечення використовується цим процесом в якості вхідних даних. Запуск його відбувається одноразово на початку всього конвеєра виявлення повторів (клонованих блоків коду). Другий процес, який посилається на логіку, проводить запуск фактичної процедури виявлення повторів та надлишковості коду та виводить виявлені клони. Другий процес буде запущено після оновлення основної бази з кодами. В реальному налаштуванні це відбувається після того, як новий запис (коміт, commit) буде розміщено в репозиторій керування версіями.

Процес створення індексу дублювання. У робочому процесі, коли за його допомогою детектор повторень створює індекс повторення (Індекс клону), вирізняють два кроки. Процес представлено на рис. 1.

Рис. 1. Підкроки робочого процесу створення «Індексу клону»

Спершу на етапі попередньої обробки подається кожний з файлів програмного проекту, який піддався аналізуванню. Відбувається зміна коду (наприклад, прибираються зайві пусті рядки) та визначення ступеню деталізації порівняння. На наступному етапі проводиться групування операторів модифікованого вихідного коду в послідовності, основою яких виступає попередньо визначений параметр конфігурації. Він визначає розмір групування та потім відбувається хешування груп. В «Індексі клону» зберігаються отримані хеші разом з додатковими метаданими (ім’я файлу та індекс оператора в файлі).

Під час даного процесу не виконується жодної логіки виявлення повторів та надлишковості для визначення початкового стану бази кодів, що стосується дублювання. Тобто, всіх існуючих блоків повторень у кодовій базі системи поки що немає. Однак, це те, чого можна досягти проводячи запит «Індексу клону» з кожним файлом початкової бази кодів. Цей крок не буде застосований, оскільки не вимагається досліджувати еволюцію клонів, а зацікавленість була тільки в дублікатах коду, які були знищені або створені в кожній новій версії.

Послідовність кроків робочого процесу. Ініціювання робочого процесу відбувається кожного разу, при відправлені наступного нового збереження до репозиторія основаного на контролі версій, де розміщена відповідна програмна система. В контексті даного дослідження, так як запропонований МНІДП не інтегрований з відповідною платформою керування версіями (наприклад, в якості плагіну), буде вручну змодельована процедура. Запуск такої процедури в іншому випадку проходив в реальних умовах автоматично. Виконання проводиться за допомогою файлу конфігурації JSON, який вказує оновлені файли, разом з типом відповідного оновлення для кожного збереження (commit), який буде аналізуватися. У типовому коміті міститься інформація, на основі якої відбувається генерація подібного файлу конфігурації. На рис. 2 представлено лістинг файлу конфігурації, призначеного для аналізу двох комітів.

```
1  {
2    "commits": [
3    {
4      "id": "cb8f645e0f",
5      "changes": [
6        { "type": "M", "filename": "lib/plugins/loader.py" }
7      ]
8    },
9    {
10     "id": "564907d8ac",
11     "changes": [
12       { "type": "A", "filename": "fragments/test_refactor.yml" },
13       { "type": "D", "filename": "fragments/arch_linux.json" },
14       { "type": "R", "filename": [
15         "test/facts/system/distribution/__init__.py",
16         "test/facts/system/__init__.py" ]
17       }
18     ]
19   }]
20 }
```

Рис. 2. Приклад файлу конфігурації JSON, який показує змінені файли за кожним комітом

В кожному коміті містяться файли, які були модифіковані (M), додані (A), знищені (D) або перейменовані (R). В більшості підетапи інкрементного робочого процесу нагадують такий же процес створення індексу. Фактично ідентичними являються етапи попередньої обробки і генерації хешу. В даному разі зв’язані операції будуть застосовані не до всієї кодової бази, а тільки до визначених файлів.

На наступному етапі проводиться порівняння генерованих хешів з хешами, які зберігаються в постійному «Індексі клону» під час процесу, який на рис. 3 вказаний як «Виявлення клону». Дублювання буде виявлене, якщо два хеші збігаються.

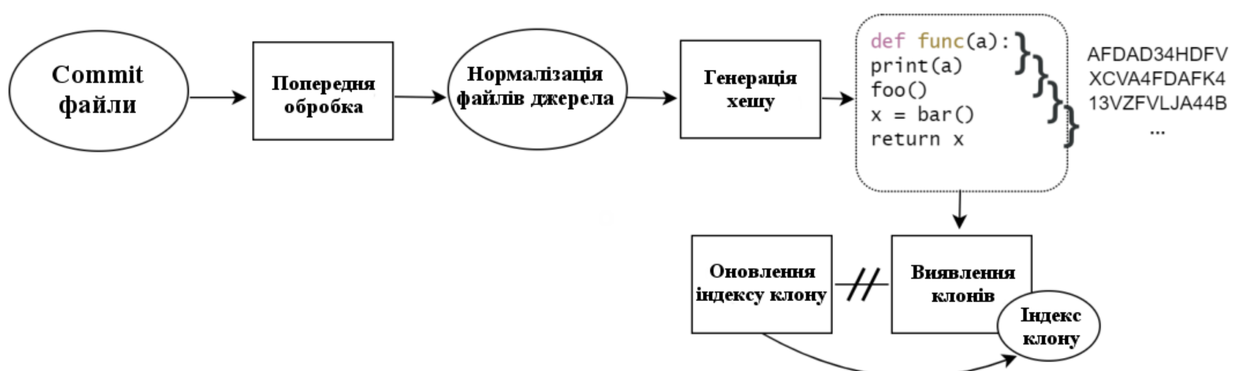

Рис. 3. Підетапи інкрементного робочого процесу

Через те, що відбувалася генерація хеш-значень шляхом хешування фіксованого числа операторів коду, наприклад N, то всі виявлені повтори (клони) будуть мати довжину N. Звідси випливає, що потрібне застосування додаткової логіки для виявлення і розширення блоків з повторами та надлишковістю до їх максимальної довжини.

Як видно з рис. 3 на останніх кроках необхідне оновлення «Індексу клону». Це дасть змогу провести на наступній ітерації порівняння хешів, які були створені файлами в майбутньому коміті з тими хешами, що збереглися в оновленому «Індексі клону». Ця операція буде виконуватися одночасно з виявленням в програмному коді блоків з повторами і надлишковістю.

Проведення попередньої обробки вихідного коду. У підході Хаммела та ін. [4] пропонується застосувати нормалізацію, яка на етапі попередньої обробки вихідного коду буде містити токенизацію. Виконується такий крок для виявлення клонів 2 типу (синтаксично ідентичних фрагментів коду, які можуть відрізнятися за типами, значеннями даних, іменами регістрів, варіаціями ідентифікаторів, літералів, коментарів), шляхом перетворення коду у послідовності лексем. На рис. 4 представлено ілюстрацію, яка введена в початкове дослідження.

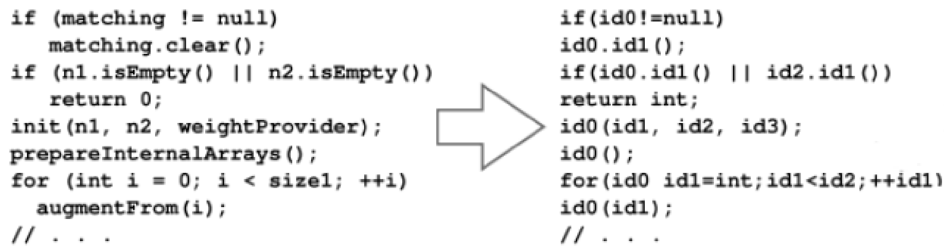

Рис. 4. Етап нормалізації, застосований в дослідженні Хаммела та ін. [4]

Однак для перетворення в токени елементів вихідного коду, таких як ідентифікатори, літерали, тощо, необхідний синтаксичний аналізатор (парсер). При цьому мова всього процесу тоді стає специфічною, через те, що для різних мов програмування потрібні різні синтаксичні аналізатори. Задача пошуку або створення синтаксичного аналізатора ускладнюється і для менш популярних мов програмування. Однак підтримка набору парсерів та використання відповідного, заснованого на основній мові (чи мовах) проекту, не є метою проведеного дослідження.

Спираючись на ці факти, в дослідженні не проводиться токенізація вихідного коду, а відбувається застосування тільки основних кроків попередньої обробки. Ці кроки не будуть впливати на функцію запропонованого МНІДП. Тобто можливість виявлення клонів 2 типу буде виключена. В дослідженні вся основна увага буде приділена точним клонам. Тому, підетапами попередньої обробки, які будуть застосовані для аналізу вихідного коду є видалення:

– початкових та кінцевих пробілів;
– подвійних пробілів;
– порожніх рядків.

Відмітимо, що видалятися коментарі не будуть, оскільки стилів коментарів у різних мовах програмування багато і їх видалення буде вимагати додаткової роботи. Синтаксичний аналізатор для видалення коментарів не обов'язковий, але все одно необхідно буде виконати сегментацію системи по компонентах різних мов для автоматизації процесу ідентифікації та видалення коментарів.

В роботі Паскарелла та ін. [5] проводиться вивчення коментування вихідного коду програм з відкритим кодом та промислових програм на базі мови Java. Спираючись на проведені дослідження виявлено, що в таких програмах існує низький відсоток співвідношення коду до коментаря. Так для систем з відкритим кодом показник знаходиться в межах 6,3–12,1 %, в промислових програмах ці дані менші і становлять 0,1–2,5 %. Великих розходжень у висновках з приводу видалення коментарів не очікується. Тому треба притримуватися простих кроків попередньої обробки, які згадувалися вище. Приклад їх використання в реальному вихідному коді представлено на рис. 5.

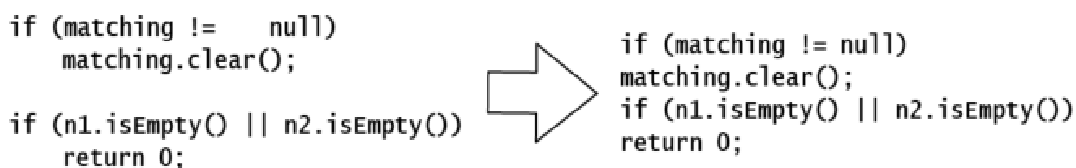

Рис. 5. Основні етапи попередньої розробки, застосовані в нашому дослідженні

Представлення вихідного коду. Для представленого детектора виявлення повторів та надлишковості програмного коду, основою виступає текст. Мається на увазі, що відсутні будь-які спеціалізовані перетворення, які застосовувалися до необробленого вихідного коду. Однак до складу проміжної інформації, яка зберігається і буде використана повторно підчас кожної редакції коду, прості попередньо

оброблені оператори не входять. Фактичною інформацією, яка збережена в «Індексі клоні», являються хеш-значення разом з метаданими. Отримуються подібні хеш-значення шляхом хешування блоків, складених з фіксованої кількості операторів.

Основою виступає ковзне вікно, де відбувається хешування один за одним послідовних блоків, маючих розмір CHUNK_SIZE, починаючи з діапазону [0, CHUNK_SIZE - 1] до [LINES_COUNT-CHUNK_SIZE, LINES_COUNT - 1]. На рис. 6, представлено приклад даного процесу із заздалегідь визначеним CHUNK_SIZE встановленим в 2. Мінімальна довжина клону неминуче визначається вибором значення для CHUNK_SIZE.

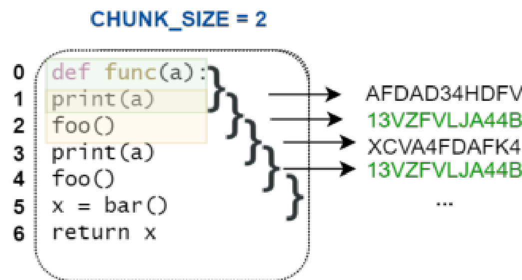

Рис. 6. Ковзне вікно хешування на основі CHUNK_SIZE

Потім в «Індексі клону» зберігається хеш-значення для кожного хешованого блоку. Там же міститься додаткова метаінформація, яка буде використана під час фази фактичного виявлення блоків з повторами і надлишковістю. До її складу входить:

Ім'я файлу, тобто файлу, де розміщено хешований блок.

Оператор індексу.

Рядок початку блоку.

Рядок кінця блоку.

Зазначимо, що «Індекс клону» можна зберігати, як в пам'яті, так і в реальній базі даних. В дослідженні збереження інформації проводиться в пам'яті, хоча в реальних умовах потрібна буде фактична база даних.

Виявлення повторів та надлишковості у програмному коді. При виникненні кожної нової зміни коду або коміті, буде ініціюватися процес виявлення повторів та надлишковості. По-перше детектор має провести обробку кожного файлу, який включено до даного коміту, і на основі попередньо визначеного CHUNK_SIZE продовжує хешування послідовних блоків коду. По-друге відбувається порівняння згенерованих хешів з тими хешами, які збережені в «Індексі клону». На виявлення повторень (клонованих частин коду) буде вказувати будь-яка пара співпадаючих значень хешу, який складається рівно з кількості рядків CHUNK_SIZE. Але цього буває недостатньо, оскільки іноді блок з повторюваним кодом може складатися з більшої кількості рядків, чим визначено CHUNK_SIZE. Відтак, детектору треба скористатися додатковою логікою, при запуску якої відбудеться дослідження, чи є можливість додатково розширити мінімальний ідентифікований клон. Саме в дослідженні Хаммела та ін. [4] аналітично представлені і пояснюються деталі цього процесу. Однак, на високому рівні буде працювати визначення того, чи будуть перекриватися окремі екземпляри клонів. Наприклад, якщо блок з повтореннями коду складається з п'яти рядків, починаючи з індексу 0 до 4, інший блок в цьому ж файлі знаходиться між рядками з індексами 1 і 5, то автоматично це буде означати, що подібні фрагменти перекриваються і окремі клони можуть об'єднуватися в один, який буде займати рядки від 0 до 5. Важливо, що даний процес об'єднання може відбуватися тільки через природу подібного детектора, спроможного спеціально фокусуватися на однакових блоках коду. Невірним виявляється припущення у разі виявлення нечітких клонів на базі оцінок.

Під час процесу виявлення повторів (клонів) та надлишковості, у фрагментах коду відбувається оновлення «Індексу клону» та підготовка його до наступного разу, коли буде запущений інкрементний покроковий процес, при настінні нового коміту. Зокрема, в залежності від типу змін, проводиться обробка пакетами змін файлів, які включені в коміт. При цьому відбуваються наступні дії:

1) проходить обробка пакету, відповідного видаленню. Створюється запит «Індексу клону» з записами індексів файлів, які було видалено. Це дозволить визначити, які з клонів були видалені, та прибрати виявлені з індексу. Подібна обробка видалень дуже важлива в першу чергу через те, що в іншому разі (наприклад, спочатку проходила обробка пакету оновлених фалів) відбулося б порівняння відповідних записів індексу з застарілим «Індексом клону»;

2) іде обробка оновлених файлів, які відповідають перейменуванню. Це аналогічно обробці видалення. Проходить виявлення повторів в коді і знищуються записи, що відповідають старим назвам файлів. Потім знову їх додають, доповнюючи новими записами з відповідними їм оновленими іменами файлів;

3) наступний крок – це оновлені файли, тобто, це розглядається як знищення з подальшим створенням. Як у випадку видалення, спочатку іде запит індексу з неоновленими версіями файлів, для виявлення клонів, які були знищені. Потім проводиться видалення застарілих записів з індексу. Згодом генерується індекс запису для оновлених версій файлів та знову створюється запит індексу, для знаходження повторів коду (клонів), які були додані. В фіналі відбувається оновлення індексу новими записами;

4) у простому випадку новостворених файлів іде генерація відповідних записів індексу, виявляються повтори в коді і в кінці проводиться оновлення індексу з додаванням записів до «Індекс клону».

Вихідні дані. Такими даними для детектора являються необроблені текстові логи (текстові файли, в яких зберігається інформація про відвідування, параметрах відвідувань якогось сайту і помилки, які виникали на цьому), які містять вказівки на виявлені повтори (клони) та надлишковості коду. Більш докладно, у логах включено: (1) ім'я файлу для кожного екземпляра клону, (2) початковий індекс кожного клону у попередньо обробленому файлі, (3) початок та кінець рядків, (4) кількість блоків коду довжиною CHUNK_SIZE, які сприяли остаточному максимальному дублюванню. На рис. 7 представлено цей лістинг, рис. 8–9 демонструє лістинги відповідних файлів кінцевого результату.

```
1 (.../project/Test.java|0|0-6) -
2 (.../project/Test2.java|1|1-7) - 2
```

Рис. 7. Приклад лістингу вихідних даних детектора

```
0 import java.lang.*;
1 import java.io.*;
2 class Test {
3     public static void main(
      String []args) {
4         System.out.println("Hello
       world");
5     }
6 }
```

Рис. 8. Лістинг Test.java

```
0 import java.util.*;
1 import java.lang.*;
2 import java.io.*;
3 class Test {
4     public static void main(
      String []args) {
5         System.out.println("Hello
       world");
6     }
7 }
```

Рис. 9. Лістинг Test2.java

Детектором в даному прикладі було виявлено повтори коду між файлами Test.java та Test2.java. В 1-му екземпляр клонованого блоку знаходиться між рядками 0–6, тоді як у 2-му можна побачити його у рядках 1–7. Важливо звернути увагу на те, що спостерігається відповідність індексів рядкам, після проведення попередньої обробки файлу. Це значить, що наприклад, якщо в файлі Test.java порожній рядок містився під індексом 0 – блок з повторами був переміщений у рядки від 1 до 7 початкового файлу – екземпляр коду з повторами (клон) все одно буде виявлено в рядках від 0 до 6, оскільки під час фази попередньої обробки відбулося видалення порожнього рядка. Детектором буде виведена кількість блоків коду, які посприяли отриманню повтору. В даному прикладі вибраний CHUNK_SIZE був призначений рівним 6. В результаті цифра 2 підтверджує те, що відбулося використання двох блоків коду довжиною 6, які перекривають один одного. Це призвело до виявлення повторів та надлишковості (клону) довжиною 7.

Висновки

Для отримання представлення про продуктивність МНІДП, його запускають для п’яти програмних систем, проводячи вимірювання вимог до часу та пам’яті. Тобто, для кожної системи відбувається аналіз серії з п’ятдесяти комітів та виміри часу і пам’яті, які будуть потрібні для процесу створення індексу. Також проходить вимірювання середнього часу, потрібного для обробки п’ятдесяти комітів під час кроків інкрементного процесу. Для спостереження за поведінкою МНІДП по відношенню до виявлених повторів та надлишковості у програмному коді, постає потреба у проведенні додаткових експериментів. Зокрема, відбувається аналіз десяти останніх комітів для кожної з систем, існуючих в наборі даних і виявлення тих повторів (клонів), які додавалися та знищувалися.

В ході експериментів відбувається порівняння ефективності запропонованого у дослідженні підходу МНІДП та сучасного традиційного підходу SIG для виявлення повторів та надлишковості в програмному коді. Остаточна мета – це перевірка покращення, яке має забезпечувати інкрементний підхід, в тому разі, коли процес виявлення повторюється регулярно для кожного нового перегляду програмного проекту.

Література

1. Nils Göde and Rainer Koschke. Incremental clone detection. In 2009 13th European Conference on Software Maintenance and Reengineering, pages 219–228. IEEE, 2009.

2. Tung Thanh Nguyen, Hoan Anh Nguyen, Jafar M Al-Kofahi, Nam H Pham, and Tien N Nguyen. Scalable and incremental clone detection for evolving software. In 2009 IEEE International Conference on Software Maintenance, pages 491–494. IEEE, 2009.

3. Yoshiki Higo, Ueda Yasushi, Minoru Nishino, and Shinji Kusumoto. Incremental code clone detection: A pdg-based approach. In 2011 18th Working Conference on Reverse Engineering, pages 3–12. IEEE, 2011.

4. Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. Indexbased code clone detection: incremental, distributed, scalable. In 2010 IEEE International Conference on Software Maintenance, pages 1–9. IEEE, 2010.

5. Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. Classifying code comments in java software systems. Empirical Software Engineering, 24(3):1499–1537, 2019.

Рецензія/Peer review : 13.05.2021 р. Надрукована/Printed :30.06.2021 р.