DOI: https://doi.org/10.36910/6775-2524-0560-2022-47-13
УДК 004.05
Коломоєць Геннадій Павлович, к.ф.-м.н., доцент,
https://orcid.org/0000-0003-3667-0316
Запорізький національний університет, м. Запоріжжя, Україна

ДОСЛІДЖЕННЯ ТЕСТОВОГО ПОКРИТТЯ НА РІВНІ БАЙТ-КОДУ JAVA

Коломоєць Г.П. Дослідження тестового покриття на рівні байт-коду Java. Кількісною характеристикою оцінки обсягу тестування програмних продуктів, рішення про завершення тестування і, врешті, визначення якості продукту та готовності його до дистрибуції є ступінь тестового покриття. Покриття коду тестами є підтипом тестового покриття і воно найчастіше визначається інженерами-програмістами. Ми можемо вимірювати ступінь покриття коду тестами за декількома критеріями, такими як покриття методу, покриття рядків, покриття операторів, покриття гілок, покриття умов гілок тощо. При оцінці результатів вимірювання покриття популярними інструментами, наприклад JaCoCo Java Code Coverage Library, за критеріями операторів, гілок та умов гілок результати є неочевидними внаслідок того, що у таких випадках ступені покриття визначаються на рівні байт-коду. Стаття наводить детальний аналіз визначення ступенів покриття на рівні байт-коду за вказаними критеріями при модульному тестуванні типових методів з комбінованою умовою з логічним оператором AND (&) у першому випадку та з оператором швидкої оцінки AND (&&) у другому випадку. Отриманий байт-код методів вивчався з відтворенням станів стеку операндів до та після кожної команди, що дозволило правильно інтерпретувати переходи у програмі. Було з'ясовано, що при визначенні ступеню покриття за критеріями гілок та умовами гілок враховувались виключно інструкції умовних переходів, причому кількість гілок дорівнювалася подвоєній кількості таких інструкцій. Також були визначені ступені покриття коду тестами за критеріями операторів та гілок для обох методів при різних наборах вхідних даних. Було визначено, що внаслідок визначення ступенів покриття саме на рівні байт-коду, для 100% покриття методу з комбінованою умовою з логічним оператором AND (&) достатньо два тестових прогони, у той час як для того ж методу, але з комбінованою умовою з оператором швидкої оцінки AND (&&) необхідні, як мінімум, три прогони.
Ключові слова: покриття коду тестами, покриття за критерієм операторів, покриття за критерієм гілок, байт-код Java, стек операндів, інструкція переходу, JaCoCo Library.

Kolomoiets H. Research of test coverage at the Java byte-code level. The test coverage is a quantitative evaluation of the software testing scope, of the decision to complete testing and, finally, of the product quality and its readiness for distribution. The code coverage is a subtype of the test coverage and it most often used by software engineers. We can measure the code coverage by several criteria such as method coverage, line coverage, statement coverage, branch coverage, branch condition coverage etc. While evaluating the measurement results of the statement coverage, branch coverage and branch condition coverage, received by popular tools, e.g. JaCoCo Java Code Coverage Library, these results are not obvious because in such cases the coverage is determined at the byte-code level. The paper provides an analysis of the code coverage calculation at the byte-code level in the unit testing of typical methods with a combined condition determined by the logical operator AND (&) in the first case and the short-circuit operator AND (&&) in the second case. We studied the obtained byte-code of the methods with the reproduction of the JVM operand stack states before and after each instruction, which allowed to interpret correctly the transitions in the program. We found for the branch and branch condition coverage only conditional transition instructions were taken into account and the number of branches was equal to twice their number. We also measured the statement coverage and branch coverage for both methods with different input data and several test runs. It was determined that for 100% code coverage of the method with a combined condition determined by the logical operator AND (&) we need at least two test runs, while for the same method, but with a combined condition specified by the short-circuit operator AND (&&) requires at least three test runs.
Keywords: code coverage, statement coverage, branch coverage, Java byte-code, operand stack, transition instruction, JaCoCo Library.

Вступ та постановка завдання. Забезпечення якості програмного забезпечення ґрунтується на оцінках його характеристик якості та результатах його тестування. Характерною рисою технологій забезпечення якості програмного забезпечення є використання кількісних оцінок – метрик для визначення поточного рівня якості програмного забезпечення та прийняття рішень щодо подальшого удосконалення продукту та його випуску. І хоча для оцінки якості програмного забезпечення використовують велику кількість метрик [1], оцінка тестування, найчастіше, виконується вимірюванням однієї метрики – тестового покриття [2].

Зауважимо, що поняття тестового покриття (Test Coverage) та покриття коду тестами (Code Coverage) часто ототожнюють або вважають рівноправними підвидами покриття, перше з яких базується на методиках "чорної скриньки" (найчастіше згадується покриття функціональних вимог), а друге – на методиках "білої скриньки" [3]. Насправді, стандарт ISO/IEC/IEEE 29119-4 Тестування програмного забезпечення – Методики тестування (Software and systems engineering – Software testing – Test Techniques) [4] зазначає покриття коду тестами як підмножину тестового покриття. У цій роботі ми зосередимось на дослідженні покриття коду тестами, а саме на вивченні алгоритмів їх визначення популярним інструментом JaCoCo Java Code Coverage Library (надалі JaCoCo Library) [5].

Аналіз останніх досліджень і публікацій. Тестове покриття – це метрика, яка визначає ступінь покриття тестами тестових елементів програмного забезпечення, що тестується, та застосовується для оцінки якості програмного забезпечення. Стандарт ISO/IEC/IEEE 29119-4 [4] у розділі "Вимірювання тестового покриття" (Test Coverage Metrics) надає загальну формулу розрахунку значення тестового покриття, а також наводить рекомендації щодо його розрахунку для кожної зі стандартизованих методик тестування. Розрахунок значення ступеню покриття виконується за формулою:

$$Coverage = \frac{N}{T} \times 100\%$$

де N – кількість тестових елементів, покритих тестами для певної методики тестування,

T – загальна кількість тестових елементів, визначених для тестування певною методикою.

Існують декілька критеріїв покриття коду тестами, основними з них є [2]:

- покриття методів/функцій (Method/Function Coverage) – визначає частку від загальної кількості методів/функцій програми, які були викликані хоча б один раз;
- покриття рядків/операторів коду (Line/Statement Coverage, C0) – визначає частку від загальної кількості рядків/операторів коду, які були виконані хоча б один раз;
- покриття гілок (Branch Coverage, C1) – визначає частку виконаних від загальної кількості усіх можливих шляхів, що визначаються операторами, які надають декілька можливих шляхів виконання програми (оператори `if else`, `switch`, оператори циклів, оператори `break`, `continue`, `return` тощо);
- покриття рішень (Decision Coverage, C2) – визначає частку значень операторів прийняття рішень (операторів `if else`, `switch`, операторів циклів), які вони приймали при тестуванні, від загальної кількості усіх можливих значень операторів прийняття рішень;
- покриття умов гілок (Branch Condition Coverage) – визначає частку значень операторів прийняття рішень (операторів `if else`, `switch`, операторів циклів) у комбінованому операторі прийняття рішення, наприклад, `if(A | B & C)`, які вони приймали при тестуванні, від загальної кількості усіх можливих значень усіх операторів прийняття рішень.

Існують програмні інструменти вимірювання покриття коду тестами, які відрізняються, зокрема, кількістю підтримуваних критеріїв покриття, які вони можуть вимірювати [6-8]. Частина з наведених критеріїв, наприклад, покриття методів/функцій, покриття рядків коду, досить легко вимірюється відповідними інструментами, але якщо взяти, наприклад, покриття операторів, то виникає питання, як врахувати можливість написання одного оператора у декількох рядках або можливість написання декількох операторів у одному рядку. Ця проблема вирішується вимірюванням кількості операторів на нижчому аніж вихідний код рівні – для Java на рівні байт-коду [6].

Постановка завдання. Метою цієї роботи було дослідження алгоритмів визначення ступеню покриття коду тестами за різними критеріями на прикладі JaCoCo Library – популярного вільно розповсюджуваного інструменту з відкритим кодом, що забезпечує вимірювання покриття коду програм на Java за максимальною кількістю критеріїв.

Викладення основного матеріалу дослідження. Для вимірювання покриття коду тестами був створений Java клас з єдиним методом:

```
public class OneMethClass {
    int foo(int x, int y) {
        int z = 0;
        if ((x > 0) & (y > 0)) {
            z = x;
        } else {
            z = y;
        }
        return z;
    }
}
```

для якого був розроблений JUnit 5 тестовий метод:

```
    @Test
    public void testFoo() {
        OneMethClass instance = new OneMethClass();
        int x = 1;
```

```
        int y = 2;
        int expResult = 1;
        int result = instance.foo(x, y);
        assertEquals(expResult, result);
    }
```

При виконанні тесту з вказаними вхідними параметрами (він буде виконаний успішно) буде викликаний метод класу `OneMethClass` і покриття *на рівні методів* буде дорівнювати 100%. Оскільки при зазначених вхідних даних тіло оператора `if` буде виконане, а тіло оператора `else` – ні, покриття методу, який складається з 6 рядків, *на рівні рядків* буде дорівнювати 5/6 ≈ 83% (рядок **`z = y;`** виконаний не буде).

Як зазначалося вище, покриття на рівні операторів визначається на рівні байт-коду, який можна отримати дизасемблюванням створеного компілятором JDK 17 файлу класу за допомогою, наприклад, утиліти JDK javap [9]. В наведеному нижче байт-коді методу додана послідовна нумерація інструкцій та коментарі зі станом стеку операндів після виконання інструкції та поясненнями. Чорним кольором виділені оператори, які виконуються тестом при будь-яких значеннях аргументів, зеленим – оператори, які виконуються при більших нуля значеннях, а синім – при значенні одного або обох аргументів, які менше або дорівнюють нулю:

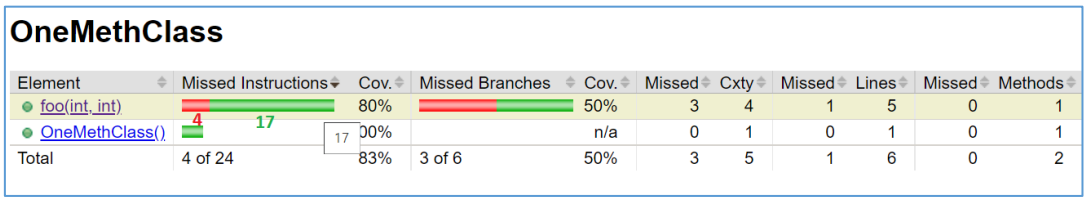
```
public int foo(int, int);
    Code:
1       0: iconst_0             //stack: 0
2       1: istore_3             //stack:      (int z = 0)
3       2: iload_1              //stack: 1    (1st arg load)
4       3: ifle          10     //stack:      (if x <= 0)
5       6: iconst_1             //stack: 1    (if x > 0)
6       7: goto          11     //stack: 1
7      10: iconst_0
8      11: iload_2              //stack: 1, 2 (2nd arg load)
9      12: ifle          19     //stack: 1    (if y <= 0)
10     15: iconst_1             //stack: 1, 1 (if y > 0)
11     16: goto          20     //stack: 1, 1
12     19: iconst_0
13     20: iand                 //stack: 1
14     21: ifeq          29     //stack: 1
15     24: iload_1              //stack: 1, 1 (1st arg load)
16     25: istore_3             //stack: 1    (z = 1st arg)
17     26: goto          31     //stack: 1
18     29: iload_2
19     30: istore_3
20     31: iload_3              //stack: 1, 1
21     32: ireturn              //stack:       (return z)
}
```

Байт-код демонструє цікаві та неочевидні особливості, тому виконаємо його детальний аналіз. Інструкція **`iconst_0`** завантажує до вершини стеку цілочисельну константу **`0`**, яка інструкцією **`istore_3`** переміщується зі стеку до масиву локальних змінних методу за індексом **`3`**, зауважимо, що за індексами **`1`** та **`2`** у цьому масиві знаходяться аргументи, передані методу при його виклику – **`int x`** та **`int y`**, відповідно. Команда **`iload_1`** завантажує аргумент **`x`** до вершини стеку, а наступна команда **`ifle 10`** перевіряє, чи менше або дорівнює **`0`** значення на вершині стеку, при цьому воно виймається зі стеку. Якщо це так, виконується перехід на команду з міткою **`10`**, якщо ні – виконується наступна команда (як у наведеному прикладі). Зверніть увагу, що компілятор змінив умову оператора **`if`** вихідного коду (**`x > 0`**) на протилежну – це перша особливість байт-коду. Далі команда **`iconst_1`** завантажує до вершини стеку цілочисельне значення **`1`** та виконується безумовний перехід командою **`goto 11`** до команди з відповідною міткою. Наступні команди **`iload_2`** та **`ifle 19`** виконують аналогічні операції з аргументом **`y`** метода. Використане у тесті значення також не ініціює переходу, тому виконуються наступні команди: **`iconst_1`**, що завантажує цілочисельне значення **`1`** до вершини стеку, та **`goto 11`**, що виконує безумовний перехід до команди з відповідною міткою.

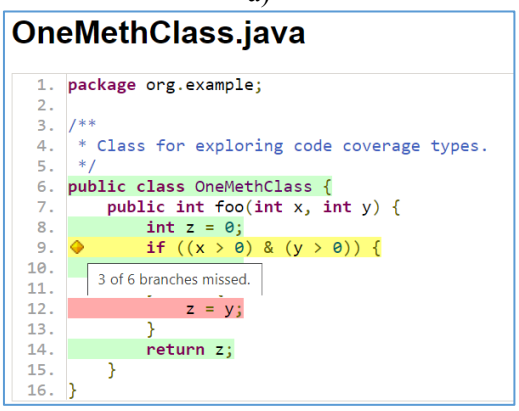
Команда **`iand`** виймає значення з вершини стеку та значення під ним та виконує побітову операцію AND над ними, результат якої завантажується до вершини стеку. У разі, якщо результат дорівнює **`0`**, виконується перехід на команду з міткою **`29`**. На перший погляд, до стеку завантажуються значення аргументів **`1`** та **`2`**, операція AND над якими дасть **`0`** і забезпечить перехід до команд, які повертають з метода аргумент **`y`** – це не відповідає високорівневому коду. І тільки з урахуванням особливості команди **`ifle`**, яка виймає значення з вершини стеку, та двох команд **`iconst_1`** (**`6`** та **`15`**), які завантажують до вершини стеку значення **`1`**, на момент виконання команди **`iand`** при зазначених у тесті аргументах у стеку знаходяться значення **`1`** та **`1`** і, відповідно, результат операції дорівнює **`1`**, що забезпечує виконання команд **`iload_1`** та **`istore_3`**, які привласнюють змінній **`z`** значення аргументу **`x`** метода, яке і повертається.

Виконане дослідження дозволяє стверджувати, що правильний аналіз програми на байт-коді вимагає вивчення алгоритму роботи задіяних інструкцій [10] та відтворення змін значень у стеку операндів методу до та після виконання кожної команди.

Результати аналізу програми на байт-коді дозволяють підрахувати, що при тестуванні з наведеними вище вхідними даними виконуються 17 інструкцій з 21, тому, значення покриття на рівні операторів (фактично на рівні інструкцій байт-коду) складає 17/21 ≈ 80%. На Рис. 1 представлені результати вимірювання покриття коду методу `foo` наведеним вище тестом, отримані за допомогою бібліотеки JaCoCo Library. Результати вимірювання на Рис. 1а демонструють 100% покриття на рівні методів (1 метод виконаний, 0 невиконані), 5/6 покриття на рівні рядків коду (5 рядків виконані, 1 рядок невиконаний), 80% покриття на рівні інструкцій байт-коду (17 інструкцій виконані, 4 невиконані) та 50% покриття на рівні гілок (останнє буде розглянуте нижче).



а)



б)

Рисунок 1 – Результати вимірювання покриття коду метода: а) статистика, б) покриття коду

Зауважимо, що покриття за критерієм гілок також підраховується на рівні байт-коду. Гілки створюються інструкціями умовних переходів, при цьому кожна з таких інструкцій містить дві гілки. У наведеному байт-коді таких інструкцій три, вони мають послідовні номери 4, 9, та 14, і при зазначених у тесті вхідних значеннях виконуються переходи, що відповідають `false`-гілкам цих інструкцій, тобто виконуються 3 з 6 гілок, що складає 50% покриття за вказаним критерієм (Рис. 2). Це відповідає результатам вимірювання JaCoCo Library (Рис. 1б). Зазначимо, що інструкції безумовного переходу `goto` на значення покриття за критерієм гілок не впливають.

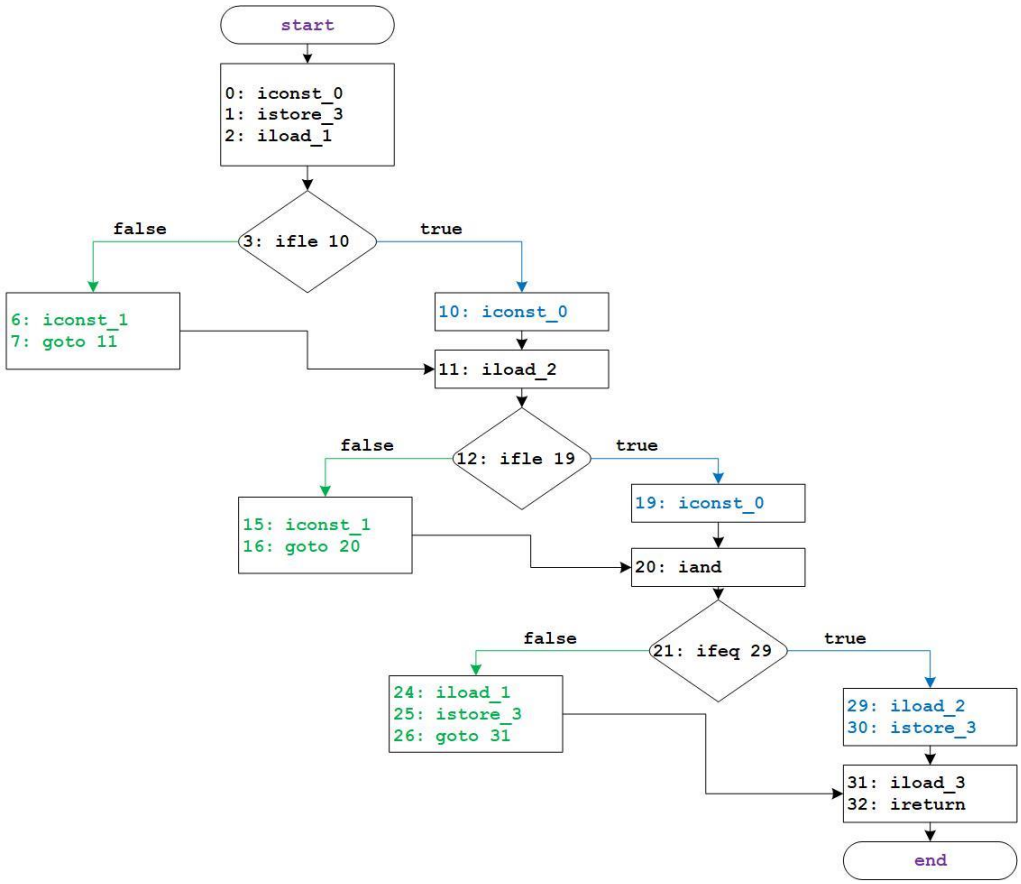


Рисунок 2 – Блок-схема програми на байт-коді метода

Нам здалося цікавим перевірити значення покриття за різними критеріями для наведеного вище методу, у якого комбінація умов створюється частіше використовуваним на практиці оператором швидкої оцінки AND (**&&**) замість звичайного оператора AND (**&**) у першому варіанті метода, тобто:

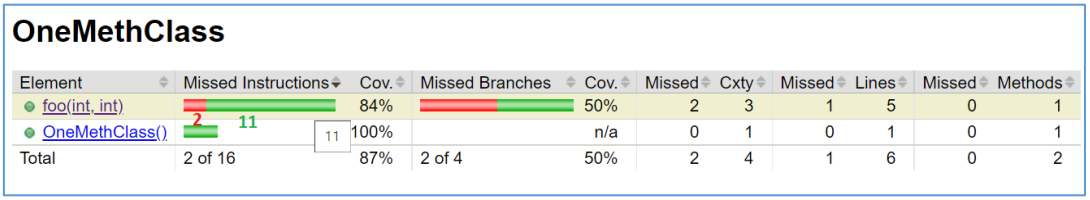
```
public class OneMethClass {
    int foo(int x, int y) {
        int z = 0;
        if ((x > 0) && (y > 0)) {
            z = x;
        } else {
            z = y;
        }
        return z;
    }
}
```

Байт-код такого методу має вигляд:

```
public int foo(int, int);
    Code:
1       0: iconst_0             //stack: 0
2       1: istore_3             //stack:      (int z = 0)
3       2: iload_1              //stack: 1    (1st arg load)
4       3: ifle          15     //stack:      (if x <= 0)
5       6: iload_2              //stack: 2    (2nd arg load)
6       7: ifle          15     //stack:      (if y <= 0)
7      10: iload_1              //stack: 1    (1st arg load)
8      11: istore_3             //stack:      (z = 1st arg)
9      12: goto         17     //stack:
10     15: iload_2
11     16: istore_3
```

```
12      17: iload_3                 //stack: 1
13      18: ireturn                 //stack:       (return z)
}
```

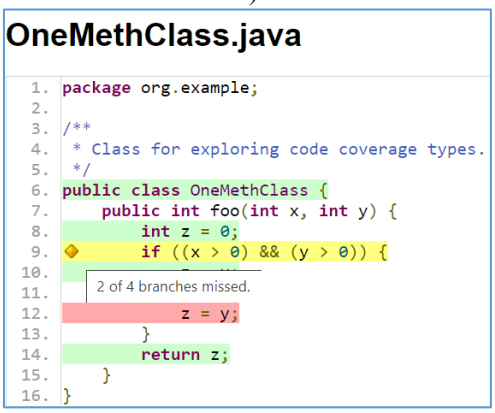
Виконавши аналогічний попередньому аналіз байт-коду, для вхідних даних **`x = 1`** та **`y = 2`** можна отримати значення покриття на рівні інструкцій 11/13 ≈ 84% та значення покриття за критерієм гілок 2/4 = 50%. Результати вимірювань JaCoCo Library (Рис. 3) та блок-схема програми (Рис. 4) підтверджують отримані значення.



OneMethClass

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
foo(int, int)	2 11	84%		50%	2	3	1	5	0	1
OneMethClass()	11	100%		n/a	0	1	0	1	0	1
Total	2 of 16	87%	2 of 4	50%	2	4	1	6	0	2

а)



OneMethClass.java

```
1.  package org.example;
2.
3.  /**
4.   * Class for exploring code coverage types.
5.   */
6.  public class OneMethClass {
7.      public int foo(int x, int y) {
8.          int z = 0;
9.          if ((x > 0) && (y > 0)) {
10.
11.             2 of 4 branches missed.
12.                 z = y;
13.         }
14.         return z;
15.     }
16. }
```

б)

Рисунок 3 – Результати вимірювання покриття коду метода з комбінованою умовою з оператором швидкої оцінки AND: а) статистика, б) покриття коду

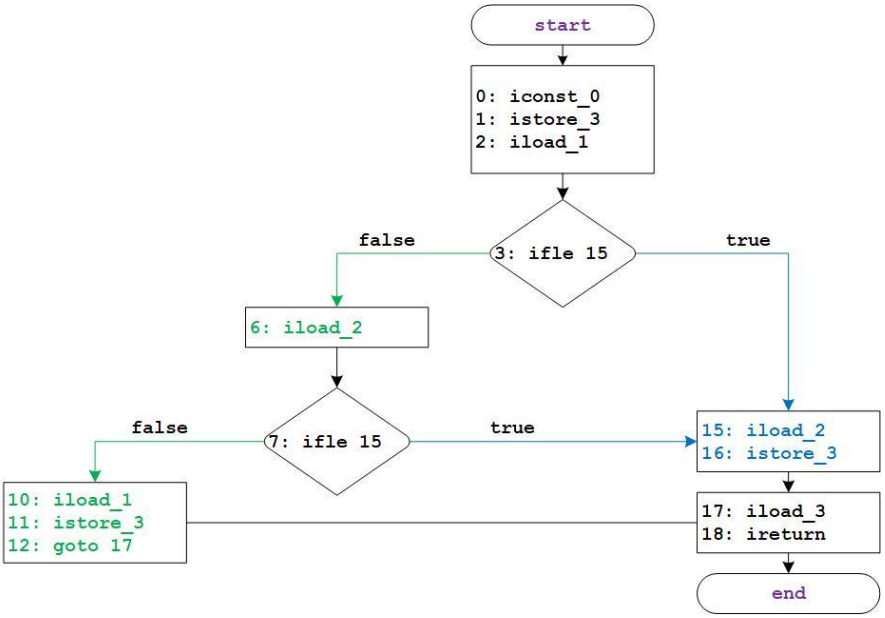


Рисунок 4 – Блок-схема програми на байт-коді метода з комбінованою умовою з оператором швидкої оцінки AND

Наведений аналіз байт-коду для двох методів був виконаний для різних наборів значень аргументів методів і були визначені ступені покриття коду тестами на рівні операторів та гілок (табл. 1).

Таблиця 1. Ступені покриття методів за критеріями операторів та гілок для різних комбінацій вхідних параметрів

Test Case #	**x > 0**	**y > 0**	**JaCoCo instructions coverage, (N/T) branch coverage (N/T)**	
			AND (&)	**short-circuit AND (&&)**
1	True	True	17/21 (3/6 branches)	11/13 (2/4 branches)
2	True	False	15/21 (3/6 branches)	10/13 (2/4 branches)
3	False	True	15/21 (3/6 branches)	8/13 (3/4 branches)
4	False	False	14/21 (3/6 branches)	8/13 (3/4 branches)
5=1+4	True False	True False	21/21 (6/6 branches)	13/13 (3/4 branches)
6=1+2	True True	True False	20/21 (5/6 branches)	13/13 (3/4 branches)
7=1+3	True False	True True	20/21 (5/6 branches)	13/13 (3/4 branches)
8=2+3	True False	False True	18/21 (5/6 branches)	10/13 (3/4 branches)
9=2+4	True False	False False	16/21 (4/6 branches)	10/13 (3/4 branches)
10=3+4	False False	True False	16/21 (4/6 branches)	8/13 (1/4 branches)
11=1+2+3	True True False	True False True	21/21 (6/6 branches)	13/13 (4/4 branches)
12=1+3+4	True False False	True True False	21/21 (6/6 branches)	13/13 (3/4 branches)
13=1+2+4	True True False	True False False	21/21 (6/6 branches)	13/13 (4/4 branches)
14=2+3+4	True False False	False True False	18/21 (3/6 branches)	10/13 (3/4 branches)
15=1+2+3+4	True True False False	True False True False	21/21 (6/6 branches)	13/13 (4/4 branches)

Зазначимо, що внаслідок визначення ступеню покриття коду на рівні байт-коду, досягнення 100% покриття для обох варіантів методів забезпечується різною мінімальною кількістю тестових сценаріїв, причому для метода зі звичайним оператором AND 100% покриття може бути досягнене двома тестовими прогонами, у той час, як для метода з комбінованою умовою з оператором швидкої оцінки AND необхідні, як мінімум, три прогони.

Висновки. Ступень тестового покриття є найпоширенішою метрикою вимірювання результатів тестування, на базі якої приймаються рішення про рівень якості та готовність продукту до випуску. Методики визначення покриття коду тестами є підмножиною методик тестового покриття. Існують декілька рівнів покриття коду тестами, частина з яких може визначатися за високорівневим кодом, але для покриття за критеріями операторів, гілок та умов гілок визначення необхідно виконувати аналізом низькорівневого коду, наприклад, байт-коду програм на Java. Компілятор Java виконує оптимізацію байт-коду, тому його інструкції можуть не відповідати високорівневим операторам, що дещо ускладнює аналіз. При аналізі байт-коду доцільно відтворювати стан стеку операндів методу до та після виконання інструкцій. При визначенні покриття на рівні гілок враховуються виключно інструкції умовних переходів з двома гілками на кожну інструкцію. Результати дослідження комбінованих умов з використанням звичайного оператора AND та оператора AND швидкої оцінки вказують на необхідність більшої кількості тестових сценаріїв для досягнення 100% покриття у останньому випадку.

Список бібліографічного опису

1. Маджид Ашурі, Пол Давідссон, Роміна Спалаццезе (2021) Атрибути якості в граничних обчисленнях для Інтернету речей: Системне дослідження. У матеріалах конференції Інженерні кіберфізичні людські системи. Т.13. С. 1-20.

2. Маурісіо Аніче Ефективне тестування програмного забезпечення. Посібник розробника. Manning Shelter Island, 2022. 329 с.
3. Шрея Босе. Покриття коду проти тестового покриття: детальний посібник. [Електронний ресурс] – Режим доступу: https://www.browserstack.com/guide/code-coverage-vs-test-coverage.
4. ISO/IEC/IEEE 29119-4:2021 (англ) Програмне забезпечення та системна інженерія – Тестування програмного забезпечення – Частина 4: Методики тестування. [Електронний ресурс] – Режим доступу: https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:29119:-4:ed-2:v1:en.
5. JaCoCo Бібліотека покриття коду Java. [Електронний ресурс] – Режим доступу: https://www.eclemma.org/jacoco.
6. Р. Сівагуру, Г. В. Канімозі, С. Алаудін Баша (2019) Оцінка різних інструментів покриття коду. Журнал глобальних досліджень та розробок у технічній галузі. Т. 4, Випуск 10. С. 21-24.
7. Самар Алі Абдалла, Ramadan Moawad (2015) Проблеми та запропоновані рішення для інструментів тестування на основі покриття. Європейський журнал наукових досліджень. Т. 131, N 1. С. 7-21.
8. Кхушбу, д-р Камна Соланкі, Сандіп Далал. (2018) Систематичне вивчення інструментів для аналізу покриття коду. Журнал нових технологій та інноваційних досліджень. Т. 5, Випуск 2. С. 216-222.
9. Команда javap. Специфікації інструменту Java® Development Kit версії 17. [Електронний ресурс] – Режим доступу: https://docs.oracle.com/en/java/javase/17/docs/specs/man/javap.html.
10. Тім Ліндхольм, Френк Єллін, Гілад Брача, Алекс Баклі, Деніел Сміт Специфікація віртуальної машини Java®. Java SE 17 Edition. Розділ 6. Набір інструкцій віртуальної машини Java. [Електронний ресурс] – Режим доступу: https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-6.html.

References

1. Majid Ashouri, Paul Davidsson , Romina Spalazzese (2021) Quality attributes in edge computing for the Internet of Things: A systematic mapping study. In Proceedings of the Engineering Cyber Physical Human Systems. V.13. P. 1-20.
2. Mauricio Aniche. Effective Software Testing. A Developer's Guide. Manning Shelter Island, 2022. 329 p.
3. Shreya Bose. Code Coverage vs Test Coverage : A Detailed Guide. [Electronic resource] - Access mode: https://www.browserstack.com/guide/code-coverage-vs-test-coverage.
4. ISO/IEC/IEEE 29119-4:2021(en) Software and systems engineering – Software testing – Part 4: Test techniques. [Electronic resource] - Access mode: https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:29119:-4:ed-2:v1:en.
5. JaCoCo Java Code Coverage Library. [Electronic resource] - Access mode: https://www.eclemma.org/jacoco.
6. R. Sivaguru, G. V. Kanimozhi, S. Alaudeen Basha (2019) Assessment on Various Code Coverage Tools. Global Research and Development Journal for Engineering. V. 4, Issue 10. P. 21-24.
7. Samar Ali Abdallah, Ramadan Moawad (2015) Challenges and Proposed Solutions of Coverage Based Testing Tools. European Journal of Scientific Research. V. 131, No 1. P. 7-21.
8. Khushbu, Dr. Kamna Solanki, Sandeep Dalal (2018) Systematic Study of Tools for Code Coverage Analysis. Journal of Emerging Technologies and Innovative Research. V. 5, Issue 2. P. 216-222.
9. The javap Command. Java® Development Kit Version 17 Tool Specifications. [Electronic resource] - Access mode: https://docs.oracle.com/en/java/javase/17/docs/specs/man/javap.html,
10. Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, Daniel Smith. The Java® Virtual Machine Specification. Java SE 17 Edition. Chapter 6. The Java Virtual Machine Instruction Set. [Electronic resource] - Access mode: https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-6.html.