Kódlefedettségi metrikák

Jeszenszky Péter

2023.03.26.

Felhasznált irodalom

- Vladimir Khorikov. *Unit Testing: Principles, Practices, and Patterns*. Manning Publications, 2020. https://github.com/vkhorikov/UnitTestingPPP
- Cătălin Tudose. *JUnit in Action*. 3rd ed. Manning Publications, 2021. https://github.com/ctudose/junit-in-action-third-edition
- Norman Fenton, James Bieman. *Software Metrics: A Rigorous and Practical Approach*. 3rd ed. CRC Press, 2014.

Kódlefedettségi metrikák

Egy kódlefedettségi metrika (röviden lefedettségi metrika) a végrehajtott forráskód mennyiségét méri százalékosan kifejezve egy tesztkészlet futtatásakor.

Kódlefedettség vs tesztlefedettség

A kódlefedettség és tesztlefedettség kifejezéseket gyakran egymással felcserélhető módon használják, noha ezek különböző dolgok.

- Lásd:
  - *Code coverage* https://en.wikipedia.org/wiki/Code_coverage
  - *Talk:Code coverage* https://en.wikipedia.org/wiki/Talk%3ACode_coverage#Code_coverage_improperly_described_as_test_coverage

A kódlefedettség kifejezést néha az utasítás lefedettség (sor lefedettség) szinonimájaként is használják.

Közvélekedés a lefedettségi metrikákról

A közvélekedés azt tartja, hogy minél nagyobb egy lefedettségi metrika értéke, annál jobb.

Az alacsony lefedettség azt jelzi, hogy a kód nem eléggé tesztelt.

A fordítottja azonban nem igaz: a nagy lefedettség nem garancia arra, hogy a tesztek jó minőségűek.

Metódus lefedettség

A **metódus lefedettség** (***method coverage***) a legalább egyszer meghívott metódusok arányát méri egy tesztkészlet futtatásakor az összes metódus számához viszonyítva.

A metódus lefedettség kiszámítása:

- Metódus lefedettség = Meghívott metódusok / Összes metódus száma

A legegyszerűbb lefedettségi metrika.

Utasítás lefedettség/sor lefedettség (1)

A leggyakrabban használt lefedettségi metrikák az **utasítás lefedettség** (***statement coverage***) és a **sor lefedettség** (***line coverage***):

- Utasítás lefedettség = Végrehajtott utasítások / Összes utasítás száma
- Sor lefedettség = Végrehajtott kódsorok / Összes sor száma

Minden egyes végrehajtott utasítást/sort egyszer számolunk.

A sor lefedettség meghatározásakor csak a végrehajtható kódot tartalmazó sorok kerülnek számolásra.

Vegyük észre, hogy a sor lefedettség függ a forráskód formázástól.

Utasítás lefedettség/sor lefedettség (2)

Példa:

```
public static boolean isLongString(String s) {
  if (s.length() > 5) {
    return true;
  }
  return false;
}

@Test
void testIsLongString() {
  assertFalse(isLongString("abc"));
}
```

A kódlefedettség $2/4 = 0{,}5 = 50\%$.

Utasítás lefedettség/sor lefedettség (3)

Példa:

```
public static boolean isLongString(String s) {
  return s.length() > 5;
}

@Test
void testIsLongString() {
  assertFalse(isLongString("abc"));
}
```

A kódlefedettség $1/1 = 1 = 100\%$.

Utasítás lefedettség/sor lefedettség (4)

Minél tömörebb a kód, annál jobb az utasítás/sor lefedettség, mivel az utasítások/sorok nyers számán alapul.

Utasítás lefedettség/sor lefedettség (5)

Példa:

```
public static String middle(String s) {
  int i = -1;
  if ((s.length() & 1) == 1) {
    i = s.length() / 2;
  }
  return s.substring(i, i + 1);
}


@Test
void testMiddle() {
  assertEquals("e", middle("voldemort"));
}
```

Vegyük észre, hogy az utasítás/sor lefedettség 100%, noha hibás a `middle()` metódus implementációja.

Ág lefedettség (1)

Az **ág lefedettség** (***branch coverage***) egy lefedettségi mérték, mely az olyan vezérlési szerkezeteken alapul, mint az `if` és a `switch`.

A végrehajtott ágak arányát méri egy tesztkészlet futtatásakor az összes ág számához viszonyítva.

Az ág lefedettség kiszámítása:

- Ág lefedettség = Végrehajtott ágak / Összes ág száma

Ág lefedettség (2)

Példa:

```
public static boolean isLongString(String s) {
  if (s.length() > 5) {
    return true;
  }
  return false;
}

@Test
void testIsLongString() {
  assertFalse(isLongString("abc"));
}
```

Az ág lefedettség $1/2 = 0{,}5 = 50\%$.

Ág lefedettség (3)

Példa:

```
public static boolean isLongString(String s) {
  return s.length() > 5;
}

@Test
void testIsLongString() {
  assertFalse(isLongString("abc"));
}
```

Az ág lefedettség $1/2 = 0{,}5 = 50\%$.

Ág lefedettség (4)

Példa: az ág lefedettség becsapása (az ág lefedettség 100%!)

```
public static int someMethod(int a, int b) {
  int x = 0, y = 0;
  if (a != 0) {
    x = a + 10;
  }
  if (b > 0) {
    y = b / x;
  }
  return y;
}

@Test
void testSomeMethod() {
  assertEquals(2, someMethod(1, 22));
  assertEquals(0, someMethod(0, -15));
}
```

Ág lefedettség (5)

Példa: az ág lefedettség becsapása (az ág lefedettség 100%!)

```
public static int someMethod(int a, int b) {
  int x = 0, y = 0;
  if (a != 0) {
    x = a + 10;
  }
  if (b > 0) {
    y = b / x;
  }
  return y;
}
```

Vegyük észre, hogy a `someMethod(0, 10)` metódushívás egy `ArithmeticException` kivételt eredményez.

Mi az ésszerű lefedettségi szám?

Veszélyes egy bizonyos érték elérésének megcélzása egy lefedettségi metrikánál, mivel könnyen ez válhat a fő céllá.

- Lásd: Martin Fowler. *AssertionFreeTesting*. 3 August 2004. https://martinfowler.com/bliki/AssertionFreeTesting.html

Inkább a megfelelő egységtesztelésre kell koncentrálni.

Ökölszabályok:

- Jó, ha egy rendszer fő részeinél nagy a lefedettség.
- Nem jó ezt magas szintű követelménnyé tenni.

Lefedettségi metrikák: összegzés (1)

A lefedettségi metrikák jó negatív indikátorok, de rossz pozitív indikátorok.

Egy alacsony kódlefedettségi metrika érték azt jelzi, hogy problémák vannak a tesztekkel, azonban egy magas érték nem jelenti azt, hogy a tesztek jó minőségűek.

Lefedettségi metrikák: összegzés (2)

A nagy lefedettség semmit sem jelent.

Példa:

```
public static int abs(int a) {
  return -a;
}

@Test
void testAbs() {
  assertEquals(5, abs(-5));
}
```

Vegyük észre, hogy az utasítás/sor/ág lefedettség 100%.

Java kódlefedettségi eszközök

- JaCoCo (licenc: Eclipse Public License 2.0)
  https://www.jacoco.org/jacoco/ https://github.com/jacoco/jacoco
  - Documentation: https://www.jacoco.org/jacoco/trunk/doc/
- IntelliJ IDEA code coverage runner:
  - *Code coverage*
    https://www.jetbrains.com/help/idea/code-coverage.html

JaCoCo Java Code Coverage Library (1)

Használat: *Maven Plug-in*
https://www.jacoco.org/jacoco/trunk/doc/maven.html

JaCoCo Java Code Coverage Library (2)

Maven konfiguráció:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>${jacoco.version}</version>
      <executions>
        <execution>
          <goals>
            <goal>prepare-agent</goal>
          </goals>
          <phase>initialize</phase>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

JaCoCo Java Code Coverage Library (3)

Maven konfiguráció:

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>${jacoco.version}</version>
      <reportSets>
        <reportSet>
          <reports>
            <report>report</report>
          </reports>
        </reportSet>
      </reportSets>
    </plugin>
  </plugins>
</reporting>
```

JaCoCo Java Code Coverage Library (4)

A JaCoCo a bájtkód utasításokat számolja az utasítás lefedettség kiszámításakor.

- Lásd: *Coverage Counters*
  https://www.jacoco.org/jacoco/trunk/doc/counters.html

JaCoCo Java Code Coverage Library (5)

Bájtkód "hangszerelést" (*bytecode instrumentation*) használ, azaz futásidejű bájtkód módosítást. A "hangszerelési" folyamat az osztálybetöltés közben történik az ASM könyvtár segítségével.

- Lásd: *Implementation Design*
  https://www.jacoco.org/jacoco/trunk/doc/implementation.html

Hangszerelés:

- *Module* `java.instrument`
  https://docs.oracle.com/en/java/javase/17/docs/api/java.instrument/java/lang/instrument/package-summary.html
- ASM https://asm.ow2.io/ https://gitlab.ow2.org/asm/asm

Ipari példák

Apache Commons Lang (licenc: *Apache License 2.0*)
https://commons.apache.org/proper/commons-lang/

- JaCoCo lefedettségi jelentés: https://commons.apache.org/proper/commons-lang/jacoco/index.html