Notatki - Testowanie - Coverage

Spis treści

Pokrycie kodu testami

Wróćmy do definicji klasy klasy `Calculator` jak poniżej:

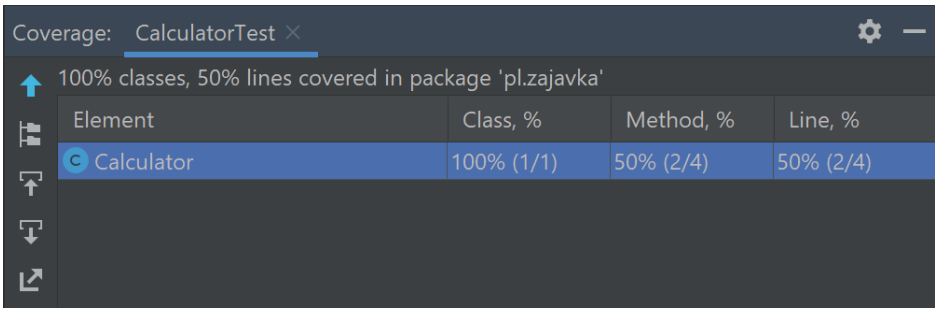
```
package pl.zajavka;

public class Calculator {

    public Integer add(int left, int right) {
        return left + right;
    }

    public Integer subtract(int left, int right) {
        return left - right;
    }

    public Integer multiply(int left, int right) {
        return left * right;
    }

    public Integer divide(int left, int right) {
        return left / right;
    }
}
```

Jeżeli teraz dodamy metody testowe, tylko do pierwszych 2 metod z klasy `Calculate` (jak poniżej):

```
package pl.zajavka;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    private Calculator calculator;

    @BeforeEach
    void beforeEach() {
        calculator = new Calculator();
    }
```

```
    @Test
    void testAdd() {
        // ciało testu
    }

    @Test
    void testSubtract() {
        // ciało testu
    }

}
```

Uruchom teraz te testy z poziomu klasy testowej, ale tym razem wykorzystaj ikonkę tarczy, która jest obok ikonki uruchamiania i ikonki debuggowania. Obok tej ikonki jest napisane **Run 'CalculatorTest' with coverage**. Na ekranie pojawi się wtedy okno, które będzie pokazywało stan pokrycia naszego kodu testami. Jeżeli teraz przeklikamy się przez paczki do `pl.zajavka` to zobaczymy klasę `Calculator` wraz z procentowymi wskazaniami pokrycia kodu testami.



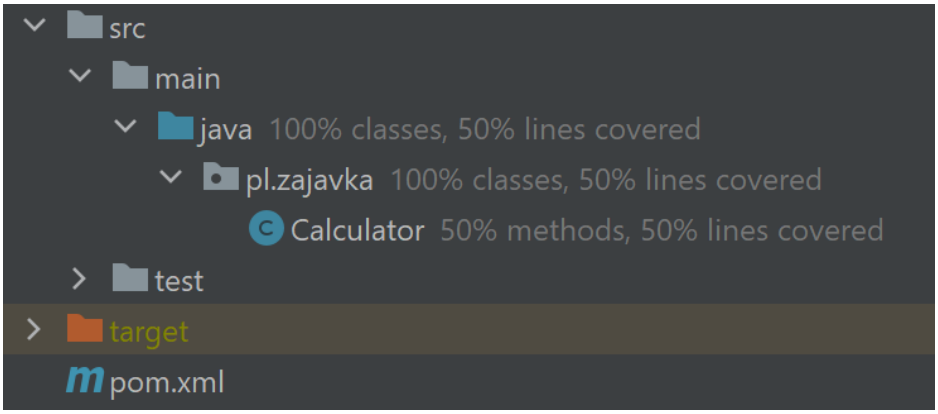
Obraz 1. Pokrycie kodu testami w Intellij

Mamy tutaj podział w tabelce na **class**, **method** i **line**. Wartości te oznaczają kolejno:

- **class** - ile klas zostało pokrytych testami - ze wszystkich możliwych klas z wykonywalnym kodem w danej paczce,
- **method** - ile metod zostało pokrytych testami z możliwych metod do wywołania,
- **line** - ile linijek kodu (z wykonywalnych) zostało pokrytych testami (nie są liczone linijki, których nie da się uruchomić). Zobacz, że wypisane są możliwe 4 linijki, które mogą być uruchomione i faktycznie w klasie `Calculator` kod "uruchamialny" jest tylko w 4 linijkach.

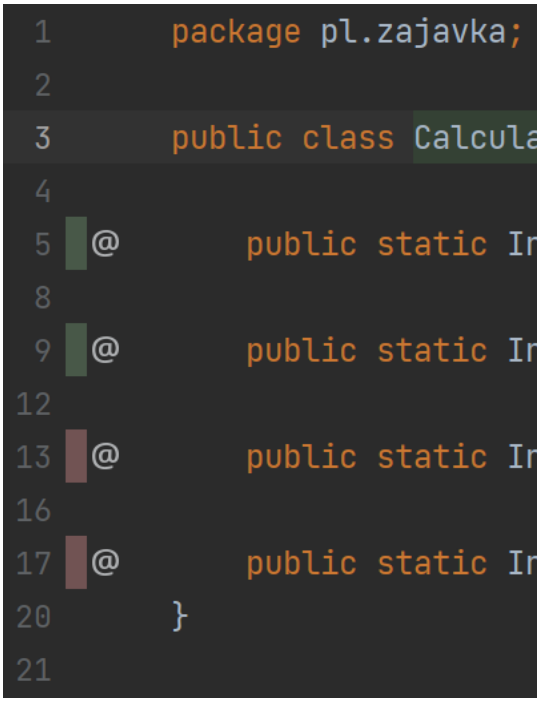
Jeżeli w tym momencie chcemy przejść do klasy, która nas interesuje, możemy kliknąć na niej w oknie z pokryciem testami i wcisnąć **F4**.

Ta sama informacja pojawia się teraz na drzewie projektu (po lewej stronie).



Obraz 2. Pokrycie kodu testami na drzewie projektu

Jednocześnie też w samej klasie `Calculator` pojawiają się kolorki mówiące, które linijki kodu zostały wykonane w trakcie działania testu (zielony), a które nie (czerwony).



Obraz 3. Pokrycie kodu testami w kodzie źródłowym

Intellj na podstawie tego, które linijki kodu zostały wykonane podczas uruchomionych testów, a które nie jest w stanie zliczyć ile faktycznie uruchamialnych linijek zostało uruchomionych podczas wykonywanych testów i zliczyć stosunek linijek wykonanych do wszytkich możliwych. W ten sposób wyliczane jest pokrycie kodu testami.

Zapewnienie minimalnego pokrycia testami

Maven

Narzędzia takie jak **Maven** albo **Gradle** pomagają nam wymusić minimalny poziom pokrycia kodu testami. Takie sprawdzenie może być uruchomione na etapie buildu, co uniemożliwi nam jego ukończenie, jeżeli nie są spełnione narzucone wymogi. Aby skonfigurować taki minimalny poziom pokrycia kodu testami, możemy to zrobić w poniższy sposób.

```xml
<project>

    <properties>
        <min.code.coverage>0.50</min.code.coverage>
    </properties>

    <!-- RESZTA PLIKU -->

    <plugin>
        <groupId>org.jacoco</groupId>
        <artifactId>jacoco-maven-plugin</artifactId>
        <version>0.8.6</version>
        <executions>
            <execution>
                <goals>
                    <goal>prepare-agent</goal>
                </goals>
            </execution>
            <execution>
                <id>jacoco-report</id>
                <phase>test</phase>
                <goals>
                    <goal>report</goal>
                </goals>
            </execution>
            <execution>
                <id>jacoco-check</id>
                <goals>
                    <goal>check</goal>
                </goals>
                <configuration>
                    <rules>
                        <rule implementation="org.jacoco.maven.RuleConfiguration">
                            <element>BUNDLE</element>
                            <limits>
                                <limit implementation="org.jacoco.report.check.Limit">
                                    <counter>LINE</counter>
                                    <value>COVEREDRATIO</value>
                                    <minimum>${min.code.coverage}</minimum>
                                </limit>
                            </limits>
                        </rule>
                    </rules>
                </configuration>
            </execution>
        </executions>
    </plugin>
</project>
```

Dodaliśmy plugin **jacoco**, który umożliwia nam określenie minimalnego progu pokrycia kodu testami. Na ten moment najważnieksze co musimy pamiętać to, że potrzebujemy jednocześnie **surefire** oraz **jacoco**, aby móc uruchamiać testy z **Maven** oraz określić minimalny próg pokrycia testami. Resztę informacji oraz przykłady konfiguracji da się wyszukać w internecie wiedząc jakie pluginy są nam potrzebne.

Po uruchomieniu polecenia `mvn verify`, zostaną uruchomione testy i obliczone pokrycie kodu testami. Tylko, że tym razem robi to plugin, a nie Intellij i zlicza to już inaczej, dlatego na ekranie pojawia się

poniższa informacja i build nie przechodzi:

```
[WARNING] Rule violated for bundle testing-maven-examples: instructions covered ratio is 0.43, but
expected minimum is 0.50
```

Jeżeli chcemy zobaczyć dlaczego **jacoco** obliczył pokrycie na poziome 0.43, a Intellij na poziomie 0.5 należy udać się do katalogu `target/site/jacoco` i otworzyć plik `index.html`. Możemy tutaj zobaczyć cały raport pokrycia kodu testami. Jeżeli wejdziemy do paczki `pl.zajavka` i do klasy `Calculator` zobaczymy, że uwzględniona została również linijka z konstruktorem, czego nie zrobił Intellij. Stąd też różnica w wynikach. Należy pamiętać o tym, że każde narzędzie jest skonfigurowane w inny sposób. Możemy w tym momencie albo dopisać więcej testów (usunęliśmy 2 z 4, więc ciągle mamy 2) aby przebić próg 50%, lub obniżyć próg. Ale już wiesz, co jest lepszym pomysłem ☺.

Oczywiście możliwych ustawień jest więcej, w tym celu możemy odnieść się do dokumentacji.

Gradle

Stosując **Gradle**, również możemy skorzystać z pluginu **jacoco**. Analogiczna konfiguracja dla **Gradle**:

```
plugins {
    id 'java'
    id 'jacoco'
}

group = 'pl.zajavka'
version = '1.0.0'
sourceCompatibility = '17'

repositories {
    mavenCentral()
}

dependencies {
    // zależności
}

test {
    useJUnitPlatform()
    testLogging {
        events "skipped", "failed"
    }
    finalizedBy jacocoTestReport
}

jacoco {
    toolVersion = "0.8.7"
}

jacocoTestReport {
    dependsOn test
    reports {
        xml.required = false
        csv.required = false
        html.outputLocation = layout.buildDirectory.dir('jacocoHtmlCustom')
    }
```

```
}

jacocoTestCoverageVerification {
    violationRules {
        rule {
            limit {
                counter = 'LINE'
                value = 'COVEREDRATIO'
                minimum = 0.5
            }
        }
    }
}

check.dependsOn jacocoTestCoverageVerification
```

Zwróć uwagę, że specjalnie w powyższej konfiguracji wpisałem swój własny katalog, gdzie będzie generowany raport **jacoco**. Katalog ten nosi nazwę **jacocoHtmlCustom**. Możemy oczywiście zmienić o wiele więcej ustawień, niż jest to pokazane w powyższym przykładzie. W tym celu najlepiej jest odnieść się do dokumentacji.