Testy jednostkowe

Według książki *The Art of Unit Testing*[14] testem jednostkowym nazywamy część kodu, w której wykonuje się pewna jednostka działania systemu i weryfikuje się czy ta jednostka zachowuje się zgodnie z pewnym określonym założeniem. Jednostką działania nazywamy pojedynczy logiczny, funkcjonalny przypadek użycia, który może być wywołany poprzez jakiś interfejs publiczny. Taką jednostką może być jedna metoda, klasa albo zbiór klas współdziałających razem, aby osiągnąć pewien jeden cel logiczny. Dlatego, że za pomocą testów jednostkowych sprawdzamy najbardziej podstawowe elementy projektu, takie jak klasy i metody – są pierwszą barierą przy walce z błędami i zwykle pisze się takich testów najwięcej.

Testy jednostkowe można podzielić na *solitary* (pl. samotny) i *sociable* (pl. towarzyski). Do pierwszej grupy zaliczają się testy, w których działanie klas używanych przez klasę testowaną jest symulowane przez programistę. Zaletę typu *solitary* stanowi fakt, że wynik takich testów zależy tylko od poprawności klasy testowanej i jest całkiem oderwany od poprawności klas używanych przez testowaną. Ponadto warto zauważyć, że działanie klas używanych przez klasę testowaną jest symulowane, przez co koszty wywoływania ich metod pozostają bardzo niskie, a same testy są znacznie szybsze. W przypadku grupy *sociable* nie symuluje się działania klas używanych przez klasę testowaną, a jedynie używa ich prawdziwych metod. Zaletą takich testów jest to, iż pozwalają zweryfikować także współdziałanie klas i mają bardziej realistyczny charakter.

Biorąc pod uwagę dane wejściowe, testy jednostkowe można podzielić na: testy klasy poprawności, testy klasy niepoprawności i testy przypadków brzegowych. Klasą poprawności nazywa się dane, w stosunku do których oczekuje się poprawnego wykonania. Klasą niepoprawności – dane, odnośnie do jakich przewiduje się, iż wykonanie skończy się błędem, a przypadkami brzegowymi – dane znajdujące się na granicy dwóch poprzednich klas. Przykładowo, przy testowaniu klasy do obliczania liczb Fibonacciego, według wzoru $F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n+1)$ *dla* $n >= 0$, do klasy poprawności należałyby m.in. {5, 8, 10}, do klasy niepoprawności m.in.. {-10, -5, -4}, natomiast do przypadków brzegowych – {-1, 0 , 1}. Sprawdzanie przypadków brzegowych ma bardzo istotne znaczenie. Często bywa bowiem tak, że błędy występują akurat dla przypadków brzegowych. Przyczynę tego faktu stanowi fakt, iż przypadki brzegowe nie są oczywiste i mogą zostać niezauważone przez programistę podczas pisania kodu lub testów do niego. Jako przykład można podać aplikację, gdzie wykorzystano typ kolumny VARCHAR(BYTE 10) w celu zapisywania dziesięciosymbolowych napisów do bazy danych. Przypadki brzegowe nie zostały sprawdzone w odpowiedni sposób. W efekcie nie istniała możliwość zapisania do kolumny dziesięciosymbolowych napisów tych z np. japońskimi znakami. Powodem opisanego stanu rzeczy był fakt, iż VARCHAR(BYTE 10) może zmieścić w sobie tylko dziesięciobajtowy napis. Na potrzeby testów używane były tylko symbole z UTF-8, z których każdy ma rozmiar 1 bajta. Wiele niełacińskich liter wykracza spoza UTF-8, a ich rozmiar może być większy od 1 bajta. W konsekwencji

dziesięcioznakowy napis w języku angielskim mieścił się w tej granicy, bo wszystkie jego znaki zaliczały się do UTF-8, ale w przypadku dziesięcioznakowego japońskiego sytuacja przedstawiała się inaczej. W testach należało użyć napisów w języku polskim, czeskim, japońskim czy zapisanych cyrylicą jako przypadków brzegowych. Tak się niestety nie stało, a fakt ten przyczynił się do utraty danych zawierających takie napisy.
Za zalety pisania testów jednostkowych można uznać to, że:

- Zapewniają poprawność. Dobrze napisane testy gwarantują prawidłowość tych fragmentów kodu, które są nimi „pokryte". W sytuacji, gdy w oprogramowaniu wystąpił błąd, to we wspomnianych fragmentach należy go szukać w ostatniej kolejności.

- Łatwo jest uruchomić test w trybie debug i sprawdzić bardziej precyzyjnie jak działa fragment kodu.

- Dokumentują kod. Są pomocne, gdy programista, który niedawno dołączył do projektu próbuje zrozumieć jak powinien działać dany fragment kodu, czyli jaki wynik jest oczekiwany w przypadku określonych danych wejściowych.

- Pomagają w szukaniu błędów. Znalezienie błędu często ułatwia napisanie testów jednostkowych do podejrzanych części kodu. Stanowi to wygodne rozwiązanie również dlatego, że taki test zadziała następnym razem, gdy jakiś nieostrożny programista powtórzy błąd.

- Zaoszczędzają czas. Często projekty są dosyć duże. Na uruchomienie, w celu sprawdzenia poprawności działania, można stracić dużo czasu. Napisanie testu nierzadko okazuje się o wiele mniej czasochłonne niż uruchomienie. Jest to także to dobra inwestycja w przyszłość, bo raz napisany test będzie zapobiegać błędom oraz umożliwiać szybkie zweryfikowanie działania fragmentu kodu. W praktyce można często spotkać się z kodem oprogramowania, którego kompilacja zajmuje ok. 3 minut, gdy uruchomienie tego trwa ok. 5 min. Zdarzają się także gorsze sytuacje, wynikające najczęściej ze złej infrastruktury, gdy uruchomienie np. aplikacji na serwerze testowym może trwać ok. 20 min.. Jest to dużą stratą czasu, szczególnie, jeśli okazuje się, że zaprowadzone zmiany nie są poprawne, więc programista jest zmuszony do naprawienia, kompilacji oraz uruchomienia aplikacji jeszcze raz.

- Szybko i często automatycznie powiadamiają o nowych błędach. Sprawdzają, czy po wprowadzeniu zmian zmieniony fragment nadal działa jak należy. Często narzędzia do automatyzacji są skonfigurowane tak, by uruchamiać testy jednostkowe po każdym wprowadzeniu zmian w repozytorium kodu. W przypadku dobrze napisanych testów jednostkowych, które wykonują się szybko, programista w bardzo krótkim czasie dowiaduje się o popełnionych błędach.

- Są szybkie - jest to najlepszy sposób na testowanie dużymi zestawami danych, w tym także przypadków brzegowych.

Jako wadę pisania testów można wymienić fakt, iż źle napisane testy powodują więcej problemów niż korzyści. Niekiedy mogą utrudniać szukanie błędów, sygnalizować o niepowodzeniu, gdy wszystko jest w porządku lub zbyt długo trwać. Naprawienie takich testów lub branie pod uwagę ich wyników może okazać się stratą czasu. Źle napisane testy nie dają programiście pewności, że jego nowo napisany kod jest poprawny, co znacznie utrudnia rozwój danego projektu. Warto zaznaczyć, że czasu wymaga nie tylko pisanie testów, ale też ich utrzymywanie, tj. dostosowywanie do zmian w wymaganiach oraz odpowiednio w kodzie, który testują. Z tego powodu, przy pisaniu testów, należy również zwracać uwagę na to, jak łatwe w modyfikacji one będą. Jeśli modyfikacja testów będzie zajmować zbyt dużo czasu, to zajęty programista będzie zmuszony odłożyć ten proces na później. W konsekwencji testy mogą zostać zaniedbane, a wówczas przywrócenie ich do dobrego stanu będzie trudnym zadaniem.

JUnit 5

JUnit 5 jest jednym z najbardziej rozbudowanych frameworków testowych w Java. Jego główną zaletę stanowi rozbudowany mechanizm testów sparametryzowanych, a mechanizm pisania własnych rozszerzeń czyni go bardzo elastycznym i funkcjonalnym dla programistów. Każda metoda testowa w klasie testowej powinna być oznaczoną adnotacją @Test.

```
@Test
void testTwoPlusTwo() {
    assertEquals(4, 2 + 2); //1
}
```

W linii //1 znajduje się asercja. Test bez asercji kończy się powodzeniem. JUnit 5 posiada wiele różnych asercji.

```
@Test
void assertSimpleExamples() {
    assertNotNull(new String("a")); //1
    assertNull(null); //2
    assertTrue(true); //3
    assertFalse(false); //4
    assertEquals(2, 2 + 2); //5
    assertEquals(2, 1 + 2, „Niepoprawny wynik działania arytmetycznego")//6
    assertArrayEquals(new String[]{"a", „b"}, new String[]{"a", „b"}); //7
}
```

W metodzie testowej powyżej są wymienione najczęściej używane asercje. Zwykle na podstawie nazwy metody można wywnioskować, jaka jest jej funkcjonalność. Przykładowo, *assertNotNull()* sprawdza, czy wartość podanego obiektu nie jest *null*, zaś *assertFalse()* weryfikuje, czy podana wartość jest *false* itd. W linii //5 znajduje się używana do porównania dwóch obiektów metoda *assertEquals()*. Jako pierwszy odpowiadający jej argument należy podać argument oczekiwany, a jako drugi – ten, który został otrzymany w wyniku. Warto również wspomnieć, że metoda jest generyczna i może porównywać obiekty dowolnego typu, jeśli metoda *equals()* została zaimplementowana w ich klasie. Metoda *assertArrayEquals()* w linii //7 działa podobnie do *assertEquals()* i także może porównywać tablice obiektów dowolnego typu, jeśli w klasach tych obiektów jest zaimplementowana metoda *equals()*. Jako jeszcze jeden argument do asercji może zostać podany napis, który ma być zwrócony przez metodę w razie nieudanej asercji jak np. w linii //6. Test może zawierać dowolną liczbę *asercji*, ale kończy się po pierwszym niepowodzeniu. Oznacza to, iż jeśli asercja w linii //3 skończy się niepowodzeniem, to żadna z następnych nie będzie już wykonywana.

Jeśli jednak zaistnieje potrzeba, by – mimo niepowodzenia – wykonać następne asercje, należy użyć metody *assertAll()*.

```
@Test
void assertAllExample() {
    assertAll(                              //1
         () -> assertEquals(2, 1 + 2), //2
         () -> assertFalse(true)        //3
    );
}
```

Metoda *assertAll()* zezwala na tworzenie bloków asercji. Wszystkie asercje – niezależnie od ich wyników w takim bloku – zostaną wykonane. Więc jeśli asercja w linii `//2` skończy się niepowodzeniem, to asercja w linii `//3` mimo to zostanie wykonana. W sytuacji, gdy przynajmniej jedna z tych metod zakończy się fiaskiem, to blok również. Gdy test składa się z kilku bloków, a jeden się skończy niepowodzeniem, to następny nie będzie wykonywany.

JUnit 5 także zezwala na sprawdzenie, czy metoda klasy testowanej wyrzuci wyjątek. Na przykład:

```
public class ExampleClass {

    public void throwException() throws Exception {
        throw new Exception("Exception text");
    }

}
```

Zweryfikować, czy metoda *throwException()* wyrzuci wyjątek oraz wychwycić go można za pomocą:

```
@Test
void assertThrowsExample() {
    ExampleClass exampleClass = new ExampleClass();
    Throwable throwable = assertThrows(
            Exception.class, exampleClass::throwException
    );
    assertEquals("Exception text", throwable.getMessage());
}
```

assertThrows() przechwyci wyjątek typu *Exception*, podany jako pierwszy argument, z wyrażenia, które jest podane jako drugi argument.

Bardzo często przy pisaniu testów zdarza się, iż niektóre obiekty są inicjalizowane poza metodą testową – jako pole klasy testowej. Dzieje się tak w przypadku, gdy taki obiekt jest wykorzystywany w wielu metodach testowych. Ponadto ma to na celu zwiększenie czytelności metody testowej. Na przykład:

```
class EntityCreatorTest {

    private EntityCreator entityCreator = new EntityCreator();

    @Test
    void numericEntityInitialization() {
        NumericEntity entity = (NumericEntity)
                entityCreator.getDbEntityByClass(NumericEntity.class);
        assertNotNull(entity);
    }

    @Test
    void textEntityInitialization() {
        TextEntity entity = (TextEntity)
                entityCreator.getDbEntityByClass(TextEntity.class);
        assertNotNull(entity);
    }

}
```

Klasa testowa *EntityCreatorTest* testuje klasę *EntityCreator,* tworzącą instancje obiektów z ich klas. W jednym teście sprawdza się tworzenie *NumericEntity*, a w drugim *TextEntity*. Bezsensownym działaniem byłoby inicjalizowanie pola `entityCreator` w obu metodach ze względu na czystość kodu oraz – jeśli tworzenie obiektu wiąże się z wysokimi kosztami – wydajność. To nadal jest podejście AAA (*arrange*, *act*, *assert*), z tym że *arrange* jest – częściowo albo całkiem – poza metodą testową i może być wspólny dla wielu testów.

Warto jednak pamiętać, że JUnit 5 tworzy domyślnie dla każdej metody testowej nową instancję klasy testowej. Jest to konieczne, by każdy test klasy testowej pozostawał niezależny od innych testów klasy testowej. Gdyby testy odbywały się na tej samej instancji klasy testowej z polami-obiektami utrzymującymi stan, to wyniki testów mogłyby ulegać zmianom w zależności od tego, co się działo w poprzednim teście. Wówczas podczas testowania *StringBuilder* może mieć miejsce poniższe zjawisko:

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class TestInstancePerClass {

    private StringBuilder stringBuilder = new StringBuilder();

    @Test
    void testUTF8() {
        String result = stringBuilder.append("Carpe")
                .append(" diem.")
                .append(" Memento")
                .append(" mori.").toString();

        assertEquals("Carpe diem. Memento mori.", result);
    }

    @Test
    void testNonUTF8() {
        String result = stringBuilder.append("Chwytaj")
                .append(" dzień.")
                .append(" Pamiętaj o")
                .append(" śmierci.").toString();

        assertEquals("Chwytaj dzień. Pamiętaj o śmierci.", result);
    }

}
```

W takim teście powstaje jedna instancja klasy testowej dla wszystkich metod testowych klasy. `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` nad klasą testową nakazuje JUnit 5 tworzyć jedną instancję klasy testowej dla wszystkich metod testowych. Metoda testowa *testNonUTF8()* kończy się niepowodzeniem. Wynika to z faktu, iż *stringBuilder* po wykonaniu *testUTF8()* utrzymuje w sobie napis „Carpe diem. Memento mori.”. Po wykonaniu kodu testu – *testNonUTF8()* – do napisu przechowywanego w *stringBuilder* dodane zostanie „Chwytaj dzień. Pamiętaj o śmierci.” i asercja skończy się niepowodzeniem. Tworzenie nowej instancji klasy testowej dla każdego testu znacznie upraszcza testowanie w takich przypadkach. Jeżeli `@TestInstance(TestInstance.Lifecycle.PER_CLASS)` zostanie usunięte, to w każdym teście pole `stringBuilder` będzie zainicjalizowane na nowo, a testy skończą się powodzeniem. Niekiedy jednak wykonywanie testów na jednej instancji klasy testowej może okazać się wygodniejsze. Będzie tak w sytuacji, gdy inicjalizacja pól klasy testowej jest zbyt kosztowna i są one obiektami bezstanowymi.

Niekiedy zainicjalizowanie pola instancji klasy testowej przed testem nie wystarcza lub nie da się tego zrobić w jednej linii. Do sytuacji takich dochodzi, gdy zaistnieje potrzeba skonfigurowania w określony sposób pól-obiektów, wypełnienia bazy danych przed testami czy wyczyszczenia jej po każdym z nich, by uniknąć konfliktu przy wykonywaniu następnego testu. JUnit 5 daje możliwość wykonywania tych działań poza metodami testowymi, za pomocą adnotacji: `@BeforeAll`, `@AfterAll`, `@BeforeEach`, `@AfterEach`.

```
class BeforeAfterExamplesTest {

    @BeforeAll
    static void beforeAll() {
        System.out.println("Początek wykonywania klasy testowej");
    }

    @AfterAll
    static void afterAll() {
        System.out.println("Koniec wykonywania klasy testowej");
    }

    @BeforeEach
    void beforeEach() {
        System.out.println("Początek wykonywania metody testowej");
    }

    @AfterEach
    void afterEach() {
        System.out.println("Koniec wykonywania metody testowej");
    }

    @Test
    void test1() {
        System.out.println("Metoda Testowa 1");
    }

    @Test
    void test2() {
        System.out.println("Metoda Testowa 2");
    }
}
```

Dla klasy testowej *BeforeAfterExamplesTest* w pierwszej kolejności wykonana zostanie metoda z adnotacją `@BeforeAll` – przed każdym z testów z `@BeforeEach`, po każdym z `@AfterEach`, i po wszystkich `@AfterAll`. Używanie tych adnotacji umożliwia pozostawienie w kodzie metod testowych tylko części *act* i *assert* z AAA (*arrange*, *act, assert*) oraz późniejsze zaimplementowanie, w prosty sposób, części *arrange.*

Jak już wspomniano, w klasach testowych mogą być zdefiniowane metody, które nie należą do metod testowych (bez adnotacji @Test). Wówczas taka metoda nie zostanie uruchomiona jako test. Mimo to może zostać użyta w innych metodach klasy testowej, włączając metody testowe. Co jednak, jeśli taka metoda duplikuje się w kilku klasach testowych lub, gdy istnieje kilka klas testowych, w których inicjalizacja danych wygląda podobnie? Czy można przeprowadzić ten proces w sposób bardziej generyczny, w celu zmniejszenia ilość kodu? Tak, umożliwiają to interfejsy testowe.

```
public interface TestInterface {

    Random randomizer = new Random();

    @BeforeEach
    default void startTest(TestInfo testInfo) {
        System.out.println(
                "Początek wykonywania testu: " + testInfo.getDisplayName()
        );
    }

    @AfterEach
    default void endTest(TestInfo testInfo) {
        System.out.println(
                "Koniec wykonywania testu: " + testInfo.getDisplayName()
        );
    }

    default long getRandomLong() {
        return randomizer.nextLong();
    }
}
```

Każda klasa testowa, implementująca interfejs *TestInterface,* ma dostęp do metod zdefiniowanych w tym interfejsie. Ponadto dla każdej klasy testowej implementującej ten interfejs zostaną uruchomione metody z adnotacją @BeforeEach i @AfterEach – przed oraz odpowiednio po każdym teście. Jeśli metody z adnotacją @BeforeEach i @AfterEach zostaną zaimplementowane także w klasie testowej, to @BeforeEach z klasy testowej zostanie wykonana po @BeforeEach z interfejsu testowego, a @AfterEach z klasy testowej przed @AfterEach z interfejsu testowego. Z użyciem takich interfejsów testowych można np. mierzyć czas wykonywania metod testowych, zapisując go w @BeforeEach i określając różnicę między nim a czasem wywołania metody @AfterEach. Dla klas implementujących interfejs powyżej będą wypisane dane wyjściowe, wskazujące, który test jest wykonywany. Także w metodach każdej klasy implementującej ten interfejs jest dostęp do metody *getRandomLong().* Jako argument dla metod @BeforeEach i @AfterEach jest podawany obiekt klasy *TestInfo* w którym mieści się informacja o teście przed albo po którym została wywołana metoda. Jeżeli metoda zostanie zdefiniowana z takim argumentem, to JUnit 5 automatycznie będzie wstrzykiwać obiekt tego typu do metody. Domyślnie JUnit 5 może wstrzykiwać tylko obiekty typu *TestInfo, TestReporter* i *RepeatedTest.* Z ich funkcjonalnościami można zapoznać się w dokumentacji JUnit 5.

JUnit 5 posiada potężny mechanizm tworzenia testów sparametryzowanych. Oznaczyć test jako sparametryzowany można za pomocą adnotacji @ParameterizedTest. Sparametryzowany test może być uruchomiony kilka razy, z wykorzystaniem różnych danych testowych. Istnieje kilka typów testów sparametryzowanych. Jednym z najprostszych jest @ValueSource:

```
@ParameterizedTest
@ValueSource(strings = {"a", "b", "c", "d"})
void valuesSourceExample(String s) {
    assertEquals(1, s.length());
}
```

@ValueSource pozwala zdefiniować zestaw parametrów jako argument adnotacji. Metoda testowa musi mieć zdefiniowany argument tego samego typu co parametry. Test powyżej zostanie wywołany 4 razy, czyli raz dla każdego ze zdefiniowanych parametrów. Do definicji parametrów mogą być użyte typy prymitywne – *String* i *Class<?>*. Parametry typu *String* są także automatycznie konwertowane do argumentów niektórych innych typów metody testowej.

```
@ParameterizedTest
@ValueSource(strings = {"2018-06-06", "2018-06-06"})
void implicitValueSourceConversionExample(LocalDate date) {
    assertEquals(2018, date.getYear());
}
```

Na przykład, w teście powyżej parametr podany jako *String* zostanie skonwertowany do typu *LocalDate*.

@MethodSource umożliwia definiowanie parametrów w metodzie. Główną zaletą tego rozwiązania jest fakt, że pozwala przekazywać parametry wieloargumentowe dowolnego typu.

```
@ParameterizedTest
@MethodSource("generateMultipleArgs")
void multipleArgsExample(String string, List<String> list) {
    assertTrue(list.contains(string));
}

private static Stream<Arguments> generateMultipleArgs() {
    return Stream.of(
            Arguments.of("M", Arrays.asList("a", "M", "b")),
            Arguments.of("D", Arrays.asList("a", "D", "b"))
    );
}
```

Metoda, w której tworzą się argumenty powinna być statyczna, jeśli instancja klasy testowej tworzy się na nowo dla każdej metody testowej. Jednak z metody statycznej nie można odwołać się do niestatycznych pól klasy testowej, co niekiedy utrudnia pracę. W przypadku, gdy instancja klasy testowej tworzy się raz dla wszystkich metod

testowych, to metoda tworząca argumenty może być niestatyczna. Żeby się do niej odwołać należy użyć adnotacji `@MethodSource`.

`@CsvSource` zezwala na przekazywanie wieloargumentowych parametrów, z których każdy jest zdefiniowany jako jeden napis formatu CSV, ale tylko niektórych typów wymienionych w dokumentacji JUnit 5.

```
@ParameterizedTest
@CsvSource({"2018-06-06, 2018", "2018-06-05, 2018"})
void implicitCSVConversionExample(LocalDate date, Year year) {
    assertEquals(date.getYear(), year.getValue());
}
```

Używając `@AggregateWith` można konwertować parametry przekazywane za pomocą `@CsvSource`.

```
@ParameterizedTest
@CsvSource({
        "The Lord of the Rings, John, Tolkien,",
        "Algorithms and Data Structures, Thomas, Cormen"
})
void testWithArgumentsAggregator(@AggregateWith(BookAggregator.class) Book
person) {

}

public class BookAggregator implements ArgumentsAggregator {
    @Override
    public Book aggregateArguments(ArgumentsAccessor arguments,
ParameterContext context) {
        Book book = new Book();
        book.setTitle(arguments.getString(0));
        Author author = new Author();
        author.setName(arguments.getString(1));
        author.setSurname(arguments.getString(2));
        book.setAuthor(author);
        return book;
    }
}
```

Jednak należy przy tym napisać własny konwerter. Konwerter ma być klasą implementującą interfejs *ArgumentAggregator* i jego metodę – *aggregateArguments()* jak to jest w przykładzie powyżej.

Za pomocą adnotacji `@CsvFileSource` także można definiować parametry w pliku w formacie CSV. Domyślnie każdy parametr ma być zdefiniowany w osobnej linii pliku, a argumenty parametru rozdzielone przecinkami. Jednak ustawiając argumenty adnotacji `lineSeparator` i `delimiter` można zmienić sposób parsowania pliku, w którym są zdefiniowane parametry.

W JUnit 5 został wprowadzony nowy sposób tworzenia testów, nazwany testami dynamicznymi. Za pomocą adnotacji `@TestFactory` można zaklasyfikować metodę jako fabrykę testów.

```
@TestFactory
Collection<DynamicTest> simpleDynamicTestExample() {
    return Arrays.asList(
            dynamicTest("Test 1", () -> assertTrue(true)),
            dynamicTest("Test 2", () -> assertEquals(4, 2 + 2))
    );
}
```

Przy uruchomieniu fabryki testów powyżej wygenerowane zostaną dwa osobno wykonywane testy. Zaletą takich testów jest fakt, że są wykonywane szybciej od testów sparametryzowanych i wykazują większą elastyczność. Dają np. możliwość czytania danych testowych z pewnego źródła i – w zależności od typu danych – tworzenia testów z różnymi asercjami.

Jednym z najbardziej złożonych mechanizmów w JUnit 5 jest mechanizm rozszerzeń. Rozszerzenie definiuje się jako klasę, która implementuje jeden lub kilka spośród następujących interfejsów: *BeforeAllCallback*, *BeforeEachCallback*, *BeforeTestExecutionCallback*, *TestExecutionExceptionHandler*, *AfterTestExecutionCallback*, *AfterEachCallback*, *AfterAllCallback*, *ParameterResolver*. Ten mechanizm umożliwia przechwytywanie wyjątków, transformację parametrów metod testowych, transformację wyników testów itd.. Na przykład, niekiedy istnieje konieczność przekazania do metody testowej obiektu ze wszystkimi rekurencyjnie zainicjalizowanymi polami. W takim przypadku można napisać rozszerzenie implementujące interfejs *ParameterResolver,* które pozwoli to robić w bardzo wygodny sposób.

```
@Test
@ExtendWith(InitTestObjectExtension.class)
void isInitializedTest(@InitTestObjectExtension.Initialized TestObj
testObj) {
    assertEquals("String", testObj.getString());
    assertNotNull(testObj.getNestedTestObj());
    assertEquals("String", testObj.getNestedTestObj().getString());
    assertEquals(42, (int) testObj.getNestedTestObj().getInteger());
}
```

InitTestObjectExtension implementuje interfejs *ParameterResolver* i jego metody: *supportParameter(),* która sprawdza, czy parametr ma być transformowany za pomocą tego rozszerzenia oraz *resolveParameter()* transformującą parametr. Rozszerzenie weryfikuje, czy parametr jest podany z adnotacją `@InitTestObjectExtension.Initialized`, pobiera jego typ, tworzy obiekt tego typu i rekurencyjnie inicjalizuje ten parametr. Aby uruchomić klasę testową lub metodę testową z pewnym rozszerzeniem, należy użyć adnotacji `@ExtendWith`.