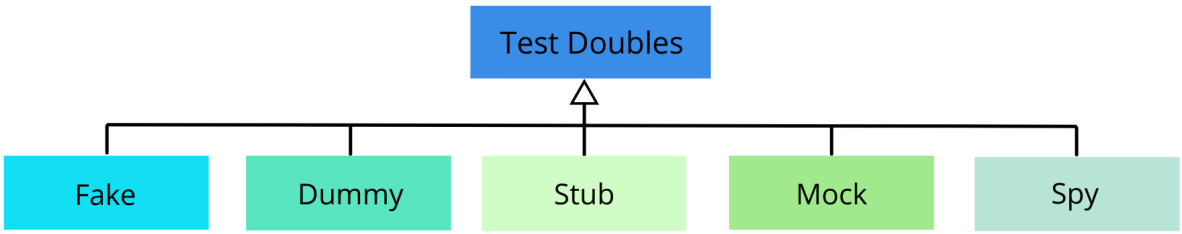
Mockowanie

Spis treści

Mockowanie

Mocki, zaślepki, atrapy - to wszystko określenia na to samo, czyli sztucznie utworzone obiekty symulujące zachowania prawdziwego obiektu w kodzie. W tej notatce wyjaśnimy po co jest i jak można tego używać.

Rodzaje mocków



Obraz 1. Rodzaje Test Doubles

- **Test Double** - pojęcie określające dowolny przypadek zastąpienia jakiegoś obiektu w celach testowych,
  - **Fake** - jego implementacja istnieje, ale jest uproszczona w sposób nienadający się do użycia w produkcji,
  - **Dummy** - rodzaj obiektu, który nie jest używany, a istnieje wyłącznie na potrzeby poprawnego działania pewnych metod np. do uzupełnienia listy parametrów,
  - **Stub** - dostarcza zaprogramowane odpowiedzi na wywołania metod wykonywanych podczas testu,
  - **Mock** - podobnie jak **Stub**, dostarcza zaprogramowane odpowiedzi na wywołania metod wykonywanych podczas testu. Do tego posiada wcześniej zdefiniowane oczekiwania użycia,

które w przypadku niespełnienia spowodują niepowodzenie testu,

- **Spy** - jest częściowym mockiem, do tego pozwala na wywołanie prawdziwych metod z obiektu.

Wiem, że w tym momencie to wszystko to jest masło maślane - wyjaśni się gdy przejdziemy do konkretów ☺.

Sporo ciekawej wiedzy na temat testów możesz znaleźć w książce *XUnit Test Patterns: Refactoring Test Code, Gerard Meszaros*. Adres do strony książki.

Testy jednostkowe najlepiej wykonywać w izolacji od innych zależności. Wymaga to zastąpienia czymś istniejących zależności testowanego **SUT**. Do tego służą wyżej wymienione rodzaje obiektów. W praktyce najczęściej spotkasz się ze **Stub** i **Mock**, rzadziej ze **Spy**. Spełniają one podobne funkcje, ale różni je implementacja i przypadki użycia.

Frameworki i biblioteki do mockowania

Frameworki i biblioteki do mockowania

- EasyMock - framework popularny w Springu do wersji 3.1 oficjalna strona
- jMock - mała biblioteka oficjalna strona
- PowerMock - framework do zadań specjalnych, takich jak prywatne metody statyczne, rozszerzający inne frameworki oficjalna strona
- **Mockito** - obecnie wiodący framework, następca EasyMock oficjalna strona — `na tym się skupimy`

Mockito

Mockito jest obecnie najpopularniejszym frameworkiem "do zaślepiania" używanym podczas pisania testów. Wyróżnia się tym, że oprócz zaślepek typu **Mock** wspiera także **Spy**. Jest dość intuicyjny, a napisany z jego pomocą test powinna łatwo zrozumieć nawet osoba niemająca na co dzień do czynienia z kodowaniem. Biblioteka Mockito dostarcza adnotacje, za pomocą których definiuje się poszczególne elementy testu, a także metody, potrzebne do weryfikacji wyników testów.

Obraz 2. Mockito logo. Źródło: https://github.com/mockito/

Przypomnienie. Aby móc skorzystać z mocy **JUnit**, należy odpowiednio skonfigurować zależności, przykładowo przy użyciu Gradle w pliku `build.gradle`:

```
dependencies {
    testImplementation "org.junit.jupiter:junit-jupiter-api:$junitVersion"
    testRuntimeOnly "org.junit.jupiter:junit-jupiter-engine:$junitVersion"
```

```
}
test {
    useJUnitPlatform()
    testLogging {
        events "passed", "skipped", "failed"
    }
}
```

To samo dla Maven:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>pl.zajavka</groupId>
    <artifactId>spring-example</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <properties>
        <java.version>17</java.version>
        <lombok.version>1.18.22</lombok.version>
        <junit.version>5.9.0</junit.version>
        <maven-compiler-plugin.version>3.8.0</maven-compiler-plugin.version>
        <maven-surefire-plugin.version>3.0.0-M5</maven-surefire-plugin.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>${lombok.version}</version>
            <scope>provided</scope>
        </dependency>

        <!-- test -->
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-api</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>${maven-compiler-plugin.version}</version>
                <configuration>
                    <release>${java.version}</release>
```

```
                    <annotationProcessorPaths>
                        <path>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                            <version>${lombok.version}</version>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>${maven-surefire-plugin.version}</version>
            </plugin>
        </plugins>
    </build>
</project>
```

Główną konkurencją JUnit jest framework TestNG.

Jak sprawić, żeby test JUnit został wykonany z użyciem Mockito?

Wystarczy taki test oznaczyć adnotacją `@ExtendWith` ze wskazaniem na klasę `MockitoExtension.class` pochodzącą ze specjalnej biblioteki *Mockito JUnit Jupiter*, która pozwala JUnit korzystać z Mockito.

Zależność biblioteki Mockito JUnit Jupiter do użycia w Gradle:

```
// https://mvnrepository.com/artifact/org.mockito/mockito-junit-jupiter
testImplementation "org.mockito:mockito-junit-jupiter:$mockitoJunitJupiterVersion"
```

To samo dla Maven:

```
<!-- ... -->
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
        <version>${mockito-junit-jupiter.version}</version>
        <scope>test</scope>
    </dependency>
<!-- ... -->
```

Adnotacja @ExtendWith,z którą spotkamy się w JUnit5:

```
import org.junit.jupiter.api.extension.ExtendWith; ①
import org.mockito.junit.jupiter.MockitoExtension; ②

@ExtendWith(MockitoExtension.class)
```

① Gradle: org.junit.jupiter:junit-jupiter-api

② Gradle: org.mockito:mockito-junit-jupiter

W JUnit4 koncepcja użycia Mockito była troszkę inna i wymagała użycia adnotacji

> *@RunWith.*
>
> *Adnotacja @RunWith:*
>
> ```
> import org.junit.runner.RunWith; ①
> import org.mockito.junit.MockitoJUnitRunner; ②
>
> @RunWith(MockitoJUnitRunner.class)
> ```
>
> ① Gradle: junit:junit
>
> ② Gradle: org.mockito:mockito-core

Główne adnotacje Mockito:

- `@InjectMock` - używana do zaznaczania testowanego serwisu, dla którego Mockito będzie podmieniać zależności na mockowane obiekty,
- `@Mock` - adnotacja stosowana do oznaczenia mockowanego serwisu,
- `@Spy` - adnotacja stosowana do oznaczenia spyowanego serwisu.

Adnotacje `@Mock` i `@Spy` mają swoje odpowiedniki w postaci metod `mock()` i `spy()`, które w odróżnieniu od adnotacji można używać wewnątrz metod testowych. Pokażemy przykład takiego zapisu.

Najczęściej używane metody Mockito:

- `when()` - do oznaczenia, którą metodę zaślepiamy,
- `thenReturn()` / `doReturn()` - do definiowania, jaką wartość ma zwracać mockowana metoda,
- `thenThrow()` / `doThrow()` - do wymuszenia rzucenia wyjątku przez mockowaną metodę,
- `verify()` - do weryfikowania wywołania metody przez test,
- `verifyNoMoreInteractions()` - do weryfikowania braku interakcji z mockowanym obiektem.

Przykłady stosowania mocków

Żeby zobrazować różne rodzaje mockowania, posłużmy się poniższym przykładem.

```
package pl.zajavka;

public interface InjectedBeanService {
    boolean anotherSampleMethod();
}
```

```
package pl.zajavka;

public class InjectedBeanServiceImpl implements InjectedBeanService {
    @Override
    public boolean anotherSampleMethod() {
        return false;
    }
```

```
}
```

```
package pl.zajavka;

public interface ExampleBeanService {
    void setInjectedBeanService(InjectedBeanService injectedBeanService);
    boolean sampleMethod();
}
```

```
package pl.zajavka;

public class ExampleBeanServiceImpl implements ExampleBeanService {
    private InjectedBeanService injectedBeanService;

    @Override
    public void setInjectedBeanService(InjectedBeanService injectedBeanService) {
        this.injectedBeanService = injectedBeanService;
    }

    @Override
    public boolean sampleMethod() {
        return injectedBeanService.anotherSampleMethod();
    }
}
```

Klasa `ExampleBeanServiceImpl` posiada jedną metodę `sampleMethod`, która korzysta z metody `anotherSampleMethod` pochodzącej ze wstrzykniętej klasy `InjectedBeanServiceImpl`. Wynikiem działania tej metody jest zawsze `boolean` wynoszący `false`. To, co będziemy chcieli osiągnąć w teście to zasymulowanie działania wstrzykniętej metody i zmuszenie jej do zwrócenia wartości zdefiniowanej przez nas, w tym przypadku będzie to wartość `boolean` wynosząca `true`.

Unit testy ze stubem

Jak to przetestować za pomocą Stuba? Całkiem prosto, wystarczy stworzyć specjalną klasę implementującą ten sam interfejs co stubbowany obiekt i uzupełnić ten obiekt o zachowanie, które jest potrzebne do przeprowadzenia testu. To o czym jeszcze trzeba pamiętać, to o wprowadzeniu tej specjalnej, stworzonej na potrzeby testu, implementacji zamiast produkcyjnej. W prostych słowach można to opisać to jako coś, co podmienia oryginalne działanie na inne, zdefiniowane przez nas na potrzeby przeprowadzenia testu. Zwróć jednocześnie uwagę, że w poniższych przykładach nie wykorzystujemy jeszcze Mockito.

Przykład stubbowania:

```
class ExampleBeanServiceImplTest {

    @Test
    void testSampleMethod() {
        // given
        ExampleBeanService exampleBeanService = new ExampleBeanServiceImpl();
        exampleBeanService.setInjectedBeanService(new StubInjectedBeanServiceImpl());

        // when
```

```
        boolean result = exampleBeanService.sampleMethod();

        // then
        Assertions.assertTrue(result);
    }

    static class StubInjectedBeanServiceImpl implements InjectedBeanService { ①
        @Override
        public boolean anotherSampleMethod() {
            return true; ②
        }
    }
}
```

① Specjalna klasa utworzona wyłącznie na potrzeby testu, implementująca interfejs `InjectedBeanService`.

② Zmiana zachowania oryginalnej metody `InjectedBeanServiceImpl#anotherSampleMethod()` w celu sprawdzenia, czy kod zachowuje się wtedy zgodnie z oczekiwaniami.

To samo można osiągnąć z zastosowaniem wewnętrznej klasy anonimowej:

```
class ExampleBeanServiceImplTest {

    @Test
    void testSampleMethodWithAnonymousClass() {
        ExampleBeanService exampleBeanService = new ExampleBeanServiceImpl();
        exampleBeanService.setInjectedBeanService(new InjectedBeanService() {
            @Override
            public boolean anotherSampleMethod() {
                return true;
            }
        });

        boolean result = exampleBeanService.sampleMethod();

        Assertions.assertTrue(result);
    }
}
```

A dodatkowo, w tym konkretnym przypadku, można jeszcze bardziej uprościć do formy korzystającej z wyrażenia lambda:

```
class ExampleBeanServiceImplTest {
    @Test
    void testSampleMethodWithLambda() {
        ExampleBeanService exampleBeanService = new ExampleBeanServiceImpl();
        exampleBeanService.setInjectedBeanService(() -> true); ①

        boolean result = exampleBeanService.sampleMethod();

        Assertions.assertTrue(result);
    }
}
```

① Użycie wyrażenia lambda jest możliwe, ponieważ interfejs InjectedBeanService jest funkcyjny.

W żadnym z przedstawionych przykładów nie użyliśmy jeszcze Mockito. Przykłady miały za zadanie pokazać jakiego rodzaju zachowania będziemy starali się osiągnąć przy wykorzystaniu Mockito.

Unit testy z mockami

Jak to przetestować za pomocą Mockito? Jeszcze prościej niż w poprzednich przykładach, bo nie ma potrzeby tworzenia dodatkowej klasy. Mockito ma dostęp do publicznych metod serwisu i na ich podstawie tworzy atrapę. Umożliwia zdefiniowanie, jakie wartości mają zwracać wywołane metody w konkretnych przypadkach. Jeżeli takiej definicji nie ma, wywołanie metody na mocku zwróci nulla. Może to wymagać dodatkowego stubbowania, ale daje pełną kontrolę deweloperowi, a także może oznaczać kolejny scenariusz testu do pokrycia. Jeżeli mamy do czynienia z metodami, które nic nie zwracają, to nadal mamy możliwość zdefiniowania różnych scenariuszy, np. rzucenia wyjątku.

Przykład mockowania:

```
@ExtendWith(MockitoExtension.class) ①
class ExampleBeanServiceImplTest {

    @InjectMocks ②
    private ExampleBeanService exampleBeanService;

    @Mock ③
    private InjectedBeanService injectedBeanService;

    @Test ④
    void testSampleMethod() {
        // given ⑤
        Mockito.when(injectedBeanService.anotherSampleMethod()).thenReturn(true);

        // when ⑥
        boolean result = exampleBeanService.sampleMethod();

        // then ⑦
        Assertions.assertTrue(result);
    }
}
```

① Oznaczamy klasę adnotacją `@ExtendWith`, żeby pokazać, że test używa Mockito,

② Definiujemy testowany serwis oraz oznaczamy go adnotacją `@InjectMocks`, żeby jak sama nazwa wskazuje, Mockito zadbało o wstrzyknięcie mockowanych zależności,

③ Definiujemy mockowane serwisy za pomocą adnotacji `@Mock`,

④ Tworzymy `@Test`, który:

⑤ *given:* definiuje, jaką wartość ma zwrócić wywołanie metody `anotherSampleMethod()` (czyli stubbing),

⑥ *when:* wywołuje testowaną metodę z serwisu `ExampleBeanService`,

⑦ *then:* weryfikuje czy otrzymany wynik jest równy oczekiwanej wartości, w tym przypadku `true`.

ArgumentMatchers.any()

Załóżmy, że metoda `anotherSampleMethod()` zaczyna teraz wymagać jednego parametru typu String.

```
package pl.zajavka;
```

```
public interface InjectedBeanService {
    boolean anotherSampleMethod(String input);
}
```

```
package pl.zajavka;

public class InjectedBeanServiceImpl implements InjectedBeanService {
    @Override
    public boolean anotherSampleMethod(String input) {
        return "zajavka".equals(input);
    }
}
```

Jeżeli będziemy chcieli w tej sytuacji zamockować metodę `anotherSampleMethod`, przyda nam się wykorzystanie `ArgumentMatchers.any()`:

Przykład mockowania:

```
@ExtendWith(MockitoExtension.class) ①
class ExampleBeanServiceImplTest {

    @InjectMocks ②
    private ExampleBeanService exampleBeanService;

    @Mock ③
    private InjectedBeanService injectedBeanService;

    @Test ④
    void testSampleMethod() {
        // given ⑤
        Mockito.when(injectedBeanService.anotherSampleMethod(ArgumentMatchers.anyString()))
            .thenReturn(false);

        // when ⑥
        boolean result = exampleBeanService.sampleMethod();

        // then ⑦
        Assertions.assertTrue(result);
    }
}
```

① Oznaczamy klasę adnotacją `@ExtendWith`, żeby pokazać, że test używa Mockito,

② Definiujemy testowany serwis oraz oznaczamy go adnotacją `@InjectMocks`, żeby jak sama nazwa wskazuje, Mockito zadbało o wstrzyknięcie mockowanych zależności,

③ Definiujemy mockowane serwisy za pomocą adnotacji `@Mock`,

④ Tworzymy `@Test`, który:

⑤ *given:* definiuje, jaką wartość ma zwrócić wywołanie metody `anotherSampleMethod()` (czyli stubbing). To co uległo tutaj zmianie to konieczność przekazania parametru typu `String` do metody `anotherSampleMethod()`. Mockito pozwala nam albo określić taki parametr na sztywno, albo możemy powiedzieć, że może tutaj w zasadzie wystąpić dowolna wartość `String` i Mockito zatroszczy się o dostarczenie mocka.

⑥ *when:* wywołuje testowaną metodę z serwisu `ExampleBeanService`,

⑦ *then:* weryfikuje czy otrzymany wynik jest równy oczekiwanej wartości, w tym przypadku `true`.

Metody `any()` służą do tego, żeby określić: *Hej Mockito, jeżeli tylko w tym miejscu znajdzie się jakikolwiek np. String, zwróć skonfigurowany mock. Jeżeli trafi się coś innego niż określono w konfiguracji, nie zwracaj skonfigurowanego mocka.* Oprócz metody `ArgumentMatchers.anyString()`, która została wykorzystana, żeby "zaślepić" przekazywanie parametru do metody mówiąc: *Tutaj może być dowolny String*, mamy dostępne też wiele innych kombinacji, przykładowo:

- `any()` - cytując dokumentację: "Matches anything, including nulls and varargs.",
- `any(Class<T> type)` - cytując dokumentację: "Matches any object of given type, excluding nulls.",
- `anyString()` - w tym miejscu może znaleźć się dowolny String, który nie jest nullem,
- `anyInt()` - w tym miejscu może znaleźć się dowolny Integer, który nie jest nullem,
- `anyList()` - w tym miejscu może znaleźć się dowolna Lista, która nie jest nullem,
- `anyMap()` - w tym miejscu może znaleźć się dowolna Mapa, która nie jest nullem,
- itp.

Wywołanie metody `any()` będzie musiało wystąpić oddzielnie dla każdego parametru jaki jest oczekiwany w metodzie. Czyli jeżeli metoda ma przykładowo poniższą definicję:

```
public String someMethod(final String someString, final Integer someInteger, final Dog someDog) {
    return "pizza";
}
```

To jej konfiguracja stubbingu Mockito mogłaby wyglądać następująco:

```
Mockito.when(someReference.someMethod(anyString(), anyInt(), any(Dog.class))).thenReturn("burger");
```

Oprócz tego możemy również zdefiniować mock w ten sposób:

```
Mockito.when(injectedBeanService.anotherSampleMethod("someStringValue")).thenReturn(false);
```

W takim przypadku mock zostanie zwrócony tylko gdy przekazany w kodzie parametr do tej metody będzie Stringiem `someStringValue`. W innym przypadku konfiguracja mocka nie zadziała i Mockito wyrzuci nam błąd informujący nas o błędnej konfiguracji mocków lub zwyczajnie zwróci w takim przypadku `null` - kwestia sytuacji.

Błędna konfiguracja Mockito

Mockito jest o tyle fajną biblioteką, że jeżeli popełnimy błąd przy konfiguracji, dostaniemy feedback przy próbie uruchomienia testu. Jeżeli przykładowo stworzymy taką konfigurację (ta sama metoda jest stubbowana dwa razy):

```
Mockito.when(injectedBeanService.anotherSampleMethod("zajavka")).thenReturn(true);
```

```
Mockito.when(injectedBeanService.anotherSampleMethod(ArgumentMatchers.anyString())).thenReturn(false);
```

To Mockito wyrzuci nam wyjątek:

```
Unnecessary stubbings detected.
Clean & maintainable test code requires zero unnecessary code.
Following stubbings are unnecessary (click to navigate to relevant line of code):
  1. -> at pl.zajavka.mocking.ExampleBeanServiceImplTest
    .testSampleMethod(ExampleBeanServiceImplTest.java:27)
Please remove unnecessary stubbings or use 'lenient' strictness.
    More info: javadoc for UnnecessaryStubbingException class.
```

Ostatnia linijka mówi nam, że mamy dwie możliwości:

- możemy pozbyć się nieużywanych stubbingów,
- drugą opcją jest dodać w konfiguracji informację, że Mockito ma przymykać na to oko - ta informacja jest nam przekazywana przez słówko *lenient*. Nie będziemy się natomiast skupiać na tej opcji, gdyż lepiej jest mieć czystą i bardziej zrozumiałą konfigurację.

Wyrzucanie wyjątków

Dotychczas konfigurowaliśmy Mockito w taki sposób, żeby metody zwracały wartości. Możliwe jest natomiast określenie, że metoda ma wyrzucić konkretny wyjątek. Przykładowo:

```
Mockito.when(injectedBeanService.anotherSampleMethod("someStringValue"))
    .thenThrow(new RuntimeException("some error"));
```

Powyższy przypadek jest przydatny gdy chcemy przetestować jak zachowa się nasz kod w momencie, gdy zależność, do której się odwołujemy wyrzuci wyjątek w trakcie działania programu.

Różne sposoby zapisu Mockito

Dotychczas korzystaliśmy z konwencji zapisu `when().thenReturn()`, gdyż jest ona najbardziej popularna w praktyce (mówimy z doświadczenia twórców zajavki, a doświadczenie każdy ma inne ☺). Oprócz tego Mockito pozwala nam na konfigurowanie stubbów na kilka różnych sposobów. Często spotykanym zapisem jest również:

```
Mockito
    .doReturn(new Dog()) ①
    .when(referenceList) ②
    .get(2); ③
```

① Ta metoda określa co ma zostać zwrócone,

② Tutaj określamy referencję, na której ma być wywołana stubbowana metoda z punktu 3,

③ Tutaj określamy faktyczną nazwę metody, która ma być stubbowana.

Ten sam zapis może zostać użyty gdy będziemy chcieli skonfigurować, że dana metoda ma wyrzucać wyjątek:

```
doThrow(new RuntimeException("some exception")).when(dog).bark();
```

Wspomniane sposoby konfigurowania stubbów, czyli `when().thenReturn()` oraz `doReturn().when()` nie są jedynymi możliwościami jakie oferuje Mockito. Nie chcemy natomiast poruszać innych sposobów na konfigurowanie stubbów. Różnych wariantów kombinacji nauczysz się już w praktyce rozwiązując zagadnienia praktyczne, na początek te przedstawione przez nas w zupełności Ci wystarczą ☺.

Jaka jest natomiast różnica pomiędzy zapisami `when().thenReturn()` a `doReturn().when()`?

Spójrzmy na poniższe przykłady:

Definicja metody `anotherSampleMethod()`

```
public interface InjectedBeanService {
    boolean anotherSampleMethod();
}
```

Różne konfiguracje mocków

```
Mockito.when(injectedBeanService.anotherSampleMethod()).thenReturn(true); ①
Mockito.when(injectedBeanService.anotherSampleMethod()).thenReturn(""); ②

Mockito.doReturn(true).when(injectedBeanService).anotherSampleMethod(); ③
Mockito.doReturn("").when(injectedBeanService).anotherSampleMethod(); ④
```

① Metoda `anotherSampleMethod()` zwraca typ `boolean`, ten przykład kompiluje się poprawnie.

② Metoda `anotherSampleMethod()` zwraca typ `boolean`, ale mock określa, że metoda ma zwrócić `String`. Dostaniemy tutaj błąd kompilacji.

③ Metoda `anotherSampleMethod()` zwraca typ `boolean`, ten przykład kompiluje się poprawnie.

④ Metoda `anotherSampleMethod()` zwraca typ `boolean`, ale mock określa, że metoda ma zwrócić `String`. Tym razem jednak nie dostajemy błędu kompilacji.

I taka jest właśnie różnica pomiędzy zapisami `when().thenReturn()` a `doReturn().when()`. Ten pierwszy zapewnia nam dodatkowe sprawdzenie poprawności typów.

verify

Przypomnij sobie, że gdy wcześniej było opisane czym jest **mock**, użyto takiego stwierdzenia: "Do tego posiada wcześniej zdefiniowane oczekiwania użycia, które w przypadku niespełnienia spowodują niepowodzenie testu". Przejdźmy zatem do tego, czym są te zdefiniowane wcześniej oczekiwania użycia.

W uproszczeniu Mockito pozwala nam sprawdzić, czy w trakcie działania jakiegoś testu, jakaś konkretna metoda została wywołana i ile razy. Możemy też upewnić się, że na pewno nie została wywołana - do tego właśnie służy `verify`. Spójrzmy na przykład poniżej:

```
Mockito.verify(injectedBeanService, Mockito.times(1)).anotherSampleMethod(Mockito.anyString()); ①
Mockito.verify(injectedBeanService, Mockito.never()).anotherSampleMethod("someValue"); ②
Mockito.verifyNoInteractions(injectedBeanService); ③
```

① W tym przypadku upewniamy się, że metoda `anotherSampleMethod()` przyjmująca dowolny String, wywoływana na `injectedBeanService`, w trakcie działania testu została wywołana tylko raz. Jeżeli w trakcie działania testu warunek ten nie zostanie spełniony - test zakończy się niepowodzeniem.

② W tym przypadku upewniamy się, że metoda `anotherSampleMethod()` przyjmująca String `someValue`, wywoływana na `injectedBeanService`, w trakcie działania testu nie została wywołana ani razu. Jeżeli w trakcie działania testu warunek ten nie zostanie spełniony - test zakończy się niepowodzeniem.

③ W tym przypadku upewniamy się, że w trakcie działania testu nie nastąpiła żadna interakcja (nie wywołano żadnych metod) na mocku `injectedBeanService`. Jeżeli w trakcie działania testu warunek ten nie zostanie spełniony - test zakończy się niepowodzeniem.

Wspomniane zapisy najczęściej pojawiają się w sekcji `// then`.

Testy parametrized

Przypomnij sobie, że rozmawialiśmy o takich testach gdy po praz pierwszy rozmawialiśmy o pisaniu testów. Specjalnie przywołujemy tutaj tę tematykę, żeby zaznaczyć, że Mockito i konfiguracja mockowania może być swobodnie używana z testami parametryzowanymi. Trzeba wtedy jedynie uważać na to, żeby nie pojawiały się u nas w teście stubby, które nie będą używane w trakcie wykonania testu - wracamy tutaj do tematu: *Unnecessary stubbings detected.*

Mockito bez wykorzystania adnotacji

Konfiguracja mockowania może się odbyć bez wykorzystania adnotacji. Poniżej znajdziesz przykład:

```
class ExampleBeanServiceImplTest {

    private ExampleBeanService exampleBeanService;

    private InjectedBeanService injectedBeanService;

    @BeforeEach
    void init() {
        this.injectedBeanService = Mockito.mock(InjectedBeanService.class); ①
        this.exampleBeanService = new ExampleBeanServiceImpl(); ②
        this.exampleBeanService.setInjectedBeanService(injectedBeanService); ③
    }

    @Test
    void testSampleMethod() {
        Mockito.when(injectedBeanService.anotherSampleMethod()).thenReturn(true);

        boolean result = exampleBeanService.sampleMethod();

        Assertions.assertTrue(result);
    }

}
```

① Tworzymy mock przy wykorzystaniu metody `Mockito.mock()`. Tak samo możemy stworzyć spy.

② Tworzymy instancję serwisu `ExampleBeanServiceImpl`, który będziemy testować.

③ Tutaj łączymy zależności ze sobą, korzystając ze wstrzyknięcia przez właściwość. To samo moglibyśmy zrobić przez konstruktor.

Unit testy ze spy

A o co chodzi z tym spyem? To jest specyficzny przypadek mocka, który oprócz tych możliwości, które daje mock, dodatkowo pozwala na wywołanie prawdziwych metod z obiektu. Żeby pokazać jego możliwości, dodajmy kolejną metodę do serwisu `ExampleBeanService`. Zadaniem metody `nextSampleMethod()` jest dodanie do listy `sampleList` kolejnych obiektów, przekazanych jako argument metody `nextSampleMethod()`.

Nowa metoda dodana na potrzeby pokazania działania spya

```
package pl.zajavka;

import java.util.List;

public class ExampleBeanServiceImpl implements ExampleBeanService {

    private List<String> sampleList = new ArrayList<>();

    // ...

    @Override
    public void nextSampleMethod(String... valuesToAdd) {
        for (String valueToAdd : valuesToAdd) {
            sampleList.add(valueToAdd);
        }
    }
}
```

W serwisie mamy nowy obiekt, listę `sampleList`, która jest uzupełniania przez metodę `nextSampleMethod()`. Parametrem metody są wartości do dodania do listy w postaci *varArgs*. W przykładzie poniżej mamy napisane dwa testy, które różnią się ilością argumentów, z jakimi została wywołana testowana metoda.

Dobra, ale jak działa spy? I o co chodzi z tym szpiegowaniem? Spya, definiujemy tak jak mocka, z użyciem adnotacji. Dalej można na nim definiować stuby, czyli określać co dane metody mają zwracać. Metody, dla których tych definicji nie ma, zostaną wywołane zgodnie z ich prawdziwą implementacją, podczas gdy w przypadku mocka, takie metody zwróciłyby `null`.

W poniższym przykładzie naszym spyem jest lista Stringów, która jest wstrzyknięta do serwisu `ExampleBeanServiceImpl`. Kiedy metoda `nextSampleMethod()` dodaje obiekty to listy, jest to właśnie ta spyowana lista. Test pracuje na wstrzykniętej liście `sampleList` oznaczonej adnotacją `@Spy`. Skoro w teście do listy dodaliśmy jakieś obiekty, to powinny one się w niej znajdować i to jest weryfikowane na końcu testu poprzez sprawdzenie rozmiaru listy metodą `.size()`.

Przykład spyowania:

```
@ExtendWith(MockitoExtension.class) ①
class ExampleBeanServiceImplTest {

    @InjectMocks ②
    private ExampleBeanService exampleBeanService;

    @Spy ③
    private List<String> sampleList = new ArrayList<>();
```

```
    @Test ④
    void testNextSampleMethod_whenOneValueToAdd() {
        // given ⑤
        String testValue = "testValue";

        // when ⑥
        exampleBeanService.nextSampleMethod(testValue);

        // then ⑦
        Mockito.verify(sampleList).add(Mockito.anyString());
        Mockito.verify(sampleList).add(testValue);
        Assertions.assertEquals(1, sampleList.size());
    }

    @Test ④
    void testNextSampleMethod_whenTwoValuesToAdd() {
        // given ⑤
        String testValue1 = "testValue1";
        String testValue2 = "testValue2";

        // when ⑥
        exampleBeanService.nextSampleMethod(testValue1, testValue2);

        // then ⑦
        Mockito.verify(sampleList, Mockito.times(2)).add(Mockito.anyString());
        Mockito.verify(sampleList).add(testValue1);
        Mockito.verify(sampleList).add(testValue2);
        Assertions.assertEquals(2, sampleList.size());
    }

}
```

① Oznaczamy klasę adnotacją `@ExtendWith`, żeby pokazać, że test używa Mockito,

② Definiujemy testowany serwis oraz oznaczamy go adnotacją `@InjectMocks`, żeby jak sama nazwa wskazuje, Mockito zadbał o wstrzyknięcie mockowanych zależności,

③ Definiujemy spyowane obiekty za pomocą adnotacji `@Spy`. Zwróć uwagę, że skoro korzystając z adnotacji `@Spy` określamy, że chcemy mockować tylko część zachowań obiektu, a pozostałe mają być rzeczywiste, to musimy powiedzieć na jakim obiekcie będzie się to odbywało. Stąd w tej samej linijce wywołujemy konstruktor tego obiektu. Jeżeli tego nie zrobimy - Mockito postara się to zrobić za nas przy wykorzystaniu konstruktora bezargumentowego w spyowanej klasie.

④ Tworzymy `@Test`, który:

⑤ *given:* definiuje zmienne wykorzystywane w teście,

⑥ *when:* wywołuje testowaną metodę z wcześniej zdefiniowanymi parametrami,

⑦ *then:* weryfikuje czy konkretne metody zostały wywołane z wybranymi argumentami, odpowiednią ilość razy oraz czy spyowany obiekt został zmodyfikowany przez metodę testową.

Jeżeli w tym przypadku, nad listą, użylibyśmy adnotacji `@Mock`, wtedy test `testNextSampleMethod_whenOneValueToAdd` zwróciłby nam błąd:

Widok konsoli

```
org.opentest4j.AssertionFailedError: expected: <1> but was: <0>
```

```
    at pl.zajavka.ExampleBeanServiceImplTest.testNextSampleMethod_whenOneValueToAdd
```

Błąd byłby spowodowany linią `Assertions.assertEquals(1, sampleList.size());`. Wcześniejsze sprawdzenia metodą `verify(sampleList).add()` przeszłoby pozytywnie. Jest tak, ponieważ metoda `add()` rzeczywiście została wywołana, ale nie mogła zmodyfikować listy `sampleList`, ponieważ ta była mockiem, a nie spyem.

To, że spy potrafi to samo co mock, można pokazać dodając stubbing metody `size()` na spyowanym obiekcie:

```
@ExtendWith(MockitoExtension.class)
class ExampleBeanServiceImplTest {

    @InjectMocks
    private ExampleBeanService exampleBeanService;

    @Spy
    private List<String> sampleList = new ArrayList<>();

    @Test
    void testNextSampleMethod_whenTwoValuesToAdd() {
        // given
        String testValue1 = "testValue1";
        String testValue2 = "testValue2";
        Mockito.when(sampleList.size()).thenReturn(987654321); ①

        // when
        exampleBeanService.nextSampleMethod(testValue1, testValue2);

        // then
        Mockito.verify(sampleList, Mockito.times(2)).add(Mockito.anyString());
        Mockito.verify(sampleList).add(testValue1);
        Mockito.verify(sampleList).add(testValue2);
        Assertions.assertEquals(2, sampleList.size());
    }
}
```

① Dodana linijka.

W rezultacie każde wywołanie metody `sampleList.size()` będzie zwracać zdefiniowaną wartość `987654321`, co zakończy test takim błędem:

Widok konsoli

```
org.opentest4j.AssertionFailedError: expected: <2> but was: <987654321>
    at pl.zajavka.ExampleBeanServiceImplTest.testNextSampleMethod_whenOneValueToAdd
```

Mockito i metody statyczne

Metody statyczne są kwestią sporną w opinii developerów, bo są odzwierciedleniem programowania proceduralnego, a nie obiektowego. Z jednej strony rozmawiamy o programowaniu obiektowym, a z drugiej sami twórcy Javy korzystają z metod statycznych, np. w `LocalDate.now()`. Występują jednak sytuacje, gdzie przyjęło się stosować metody statyczne i są to m.in:

- Metoda statyczna w przeciwieństwie do konstruktora może nosić nazwę i dzięki temu można w ten sposób czytelniej przedstawić proces tworzenia obiektu,
- Metoda statyczna może być wykorzystana we wzorcu fabryki, gdzie metoda może definiować, że zwraca jakąś klasę bazową, podczas gdy faktycznie będzie zwracana jakaś konkretna implementacja. Znowu wracamy tutaj do przykładu tworzenia obiektu,
- Stosując metody statyczne możemy dokonywać pewnego rodzaju optymalizacji. Wzorzec Singleton jest tego przykładem, używamy ciągle tego samego obiektu.

Jakie to ma znaczenie w kontekście testowania? Testy jednostkowe mają za zadanie sprawdzenie działania kawałka funkcjonalności w izolacji, którą otrzymuje się za pomocą mockowania zależności. Czyli czegoś, czego nie da się osiągnąć w przypadku metod statycznych, które nie wymagają tworzenia nowego obiektu, żeby je wywołać. Najczęściej, jeżeli jakieś metody statyczne wymagają mockowania, może to wskazywać na problem w designie lub *code smell*, który wypadałoby rozwiązać. Z drugiej strony mamy taki przypadek jak `LocalDate.now()` i ciężko powiedzieć, żeby był to problem w designie, gdy potrzebujemy zamockować datę.

> **Code smell** is any characteristic in the source code of a program that possibly indicates a deeper problem.
>
> — Wiki

Zajmijmy się zatem statyczną metodą `java.time.LocalTime.now()`. Jeżeli w naszym kodzie będzie występować jej użycie, to w jaki sposób ją zamockować?

Przykład użycia `java.time.LocalTime.now()`:

```
public class StaticMethodExample {

    public int getNano() {
        LocalTime now = LocalTime.now();
        return now.getNano();
    }
}
```

Przykład testowania metody statycznej bez mockowania:

```
class StaticMethodExampleTest {

    private StaticMethodExample staticMethodExample = new StaticMethodExample();

    @Test
    void testGetNanoNow() {
        // given
        int nanoNow = LocalTime.now().getNano();
        // when
        int result = staticMethodExample.getNano();
        // then
        Assertions.assertEquals(result, nanoNow);
    }
}
```

Jak widać, taki test niewiele nam daje. Do wersji 3.4.0, Mockito nie wspierało testowania metod

statycznych. Wtedy, jednym ze sposobów było opakowanie metody statycznej w dodatkową metodę. Taki zabieg pozwala na zamockowanie tej nowej metody. Niestety niesie ze sobą konsekwencje w postaci modyfikacji testowanego serwisu.

Przykład testowania metody statycznej z metodą wrappującą:

```
public class StaticMethodExample {

    public int getNano() {
        LocalTime now = getNow();
        return now.getNano();
    }

    private LocalTime getNow() {
        return LocalTime.now();
    }
}
```

```
@ExtendWith(MockitoExtension.class)
class StaticMethodExampleTest {

    @Spy
    private StaticMethodExample staticMethodExample = new StaticMethodExample();

    @Test
    void testGetNanoNow() {
        // given
        LocalTime now = LocalTime.now();
        LocalTime nowEarlier = now.minusNanos(100);
        int nanoNow = now.getNano();
        int nanoEarlier = nowEarlier.getNano();
        Mockito.when(staticMethodExample.getNow()).thenReturn(nowEarlier);
        // when
        int result = staticMethodExample.getNano();
        // then
        Assertions.assertNotEquals(result, nanoNow);
        Assertions.assertEquals(result, nanoEarlier);
    }
}
```

Do testowanego serwisu dodaliśmy metodę `getNow()`. Jej jedynym zadaniem jest wywołanie metody statycznej `LocalTime.now()`. Dzięki temu będziemy w stanie odizolować od testu wywołanie metody statycznej. W teście zastosowana została adnotacja `@Spy`, ponieważ zarówno metoda testowana `getNano()` jak i metoda `getNow()`, którą chcemy zamockować, pochodzą z tego samego serwisu.

Od wersji 3.4.0 Mockito wprowadziło wsparcie testów metod statycznych. Aby móc z niego skorzystać trzeba uzupełnić zależności o bibliotekę *mockito-inline*.

```
// https://mvnrepository.com/artifact/org.mockito/mockito-inline
testImplementation "org.mockito:mockito-inline:$mockitoVersion"
```

Przykład testowania metody statycznej z użyciem `MockedStatic`*:*

```
public class StaticMethodExample {
```

```
    public int getNano() {
        LocalTime now = LocalTime.now();
        return now.getNano();
    }
}
```

```
package pl.zajavka;

import java.time.LocalTime;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.mockito.MockedStatic;
import org.mockito.Mockito;

class StaticMethodExampleTest {

    private StaticMethodExample staticMethodExampleSpy = new StaticMethodExample();

    @Test
    void testGetNanoNow() {
        // given
        LocalTime now = LocalTime.now();
        int nanoNow = now.getNano();
        LocalTime nowEarlier = now.minusNanos(100);
        int nanoEarlier = nowEarlier.getNano();

        int result;
        try(MockedStatic<LocalTime> timeMock = Mockito.mockStatic(LocalTime.class)) {
            timeMock.when(LocalTime::now).thenReturn(nowEarlier);
            // when
            result = staticMethodExampleSpy.getNano();
        }

        // then
        Assertions.assertNotEquals(result, nanoNow);
        Assertions.assertEquals(result, nanoEarlier);
    }

}
```

W takim podejściu nie ma potrzeby modyfikowania testowanego serwisu `StaticMethodExample` i nie jest potrzebna metoda `getNano()`. W tym teście został użyty nowy element `MockedStatic`. Został on opakowany konstrukcją `try-with-resources`, aby mieć pewność, że taki mock będzie tymczasowy. Nie jest tu potrzebne użycia adnotacji `@Spy`, nie ma więc też potrzeby stosowania `@ExtendWith(MockitoExtension.class)`.

BDD Mockito

Wcześniej pojawiło się stwierdzenie **TDD**, teraz przejdziemy do omówienia **BDD**. Stwierdzenie to ukazało się po raz pierwszy w tym artykule: Introducing BDD, a rozwinięcie tego skrótu to ***B**ehaviour-**D**riven **D**evelopment*. Podejście to zaleca nam pisanie testów w języku naturalnym, który jest czytelny dla człowieka, jednocześnie koncentrując się na zachowaniu aplikacji. Podejście to definiuje ustrukturyzowany sposób pisania testów i określa trzy sekcje: *Arrange*, *Act* oraz *Assert*:

- Arrange - zakładając pewne warunki początkowe (*given*),
- Act - gdy wystąpi jakaś akcja (*when*),
- Assert - zweryfikuj dane wyjściowe (*then*).

O konwencji *given, when then* rozmawialiśmy już wcześniej. Teraz przyszła pora, żeby rozwinąć ten temat. Inne określenie na tę konwencję to **GWT** - Wikipedia. Nazwa ta wywodzi się od trzech wykorzystanych klauzul: *given, when then*. *Given* opisuje warunki początkowe testu przed jego rozpoczęciem i pozwala na określenie konfiguracji przed wykonaniem akcji. *When* opisuje działania, które są podejmowane podczas testu. *Then* opisuje wynik wynikający z działań podjętych w trakcie wykonywania testu.

Okej, ale co ma jedno do drugiego? Pan Dan North, który jest autorem **BDD** zaproponował konwencję **GWT** właśnie jako element podejścia do pisania testów jakim jest **BDD**. Podczas pisania testów, konwencja ta ma pomóc podzielić każdy z testów na sekcje, w których możemy spodziewać się konkretnych rodzajów zapisów w kodzie.

A jak *BDD* ma się do *TDD*?

BDD jest ewolucją **TDD**. **BDD** to metodologia tworzenia oprogramowania, w której aplikacja jest dokumentowana i zaprojektowana z myślą o zachowaniu, którego użytkownik oczekuje podczas interakcji z nią. Podejście to zaleca nam pisanie testów w języku naturalnym, który jest czytelny dla człowieka jednocześnie koncentrując się na zachowaniu aplikacji. BDD łączy, rozszerza i udoskonala praktyki, które zostały wprowadzone w TDD. BDD ma być z założenia bardziej opisowe, testy mają być pisane bardziej jak zdania niż procedury, dzięki czemu mamy być w stanie lepiej zrozumieć naturę testów i przypadki testowe. TDD skupiało się na konkretnych funkcjach w kodzie i na samych wynikach testów. BDD ma być bardziej życiowe z punktu widzenia użytkownika aplikacji.

Podejście BDD zakłada, że w tworzeniu danego przypadku, który będzie testowany powinni uczestniczyć developer, test engineer oraz menadżer produktu. Grupa taka może się wtedy spotkać żeby zebrać konkretne przypadki użycia na ścieżce użytkownika. Następnie mając taką specyfikację, developer jest w stanie napisać przypadek testowy.

BDD oraz TDD są do siebie bardzo podobne, oba odnoszą się do strategii testowania aplikacji. W obu przypadkach programista pisze testy przed napisaniem kodu i dąży do tego, żeby test finalnie przeszedł. W obu przypadkach testy służą do automatyzacji wykrywania błędów. Różnica jest natomiast taka, że BDD jest skupia sie na testowaniu zachowania aplikacji z punktu widzenia użytkownika końcowego. TDD koncentruje się na testowaniu konkretnych funkcjonalności w izolacji. Można to zrozumieć w ten sposób, że TDD oczekuje konkretnych wyników z wywoływanych metod, podczas gdy BDD działa na scenariuszach, które są na wyższym poziomie abstrakcji niż metody i oczekuje, że dane scenariusze zachowań zostaną spełnione. Należy zaznaczyć, że BDD i TDD nie wykluczają się wzajemnie. Można pisać testy i w jednym i w drugim podejściu obok siebie, w jednym programie. BDD ma zapewnić, że przypadki użycia zdefiniowane w aplikacji będą spełnione. Można to rozumieć jak testy na wyższym poziomie ogólności.

Podejście to ma zacieśnić współpracę między programistami, testerami i biznesem, żeby zmniejszyć ilość nieporozumień wokół kryteriów akceptacji.

Jak to z każdą kwestią filozoficzną bywa, każdy ma swoje zdanie. Chcieliśmy natomiast, żeby informacja o **BDD** pojawiła się na ścieżce Zajavkowej, bo zalecamy pisanie testów w konwencji **GWT** i dopiero przy omawianiu Mockito chcieliśmy wyjaśnić skąd ona pochodzi. A dlaczego dopiero teraz? Przez Mockito

BDD.

Mockito dostarcza API, które pozwala pisać testy bardziej pasujące do konwencji BDD. Możemy odwoływać się do tego API mówiąc **BDDMockito**. Spójrzmy zatem na przykład:

```
package pl.zajavka.bdd;

import lombok.Builder;
import lombok.Value;

import java.util.Objects;

@Value
@Builder
public class User {

    String name;
    String surname;
    String phone;

    public boolean isValid() {
        return Objects.nonNull(name)
            && Objects.nonNull(surname)
            && Objects.nonNull(phone);
    }
}
```

```
package pl.zajavka.bdd;

import lombok.RequiredArgsConstructor;

import java.util.Optional;

@RequiredArgsConstructor
public class UserRegistryService {

    private final UsersRepository usersRepository;

    public void register(User user) {
        if (!user.isValid() || usersRepository.contains(user.getPhone())) {
            throw new RuntimeException("Invalid user data");
        }
        usersRepository.put(user);
    }

    public Optional<User> find(String phone) {
        if (phone.isEmpty()) {
            throw new RuntimeException("Invalid phone number");
        }
        if (usersRepository.contains(phone)) {
            return Optional.ofNullable(usersRepository.find(phone));
        }
        return Optional.empty();
    }
}
```

```java
package pl.zajavka.bdd;

import java.util.HashMap;
import java.util.Map;

public class UsersRepository {

    private final Map<String, User> USERS_MAP = new HashMap<>();

    public void put(final User user) {
        if (user.getPhone().length() > 7) {
            throw new RuntimeException("Phone number is too long");
        }
        if (USERS_MAP.containsKey(user.getPhone())) {
            throw new RuntimeException(
                String.format("Phone: %s already exists in registry", user.getPhone()));
        }
        USERS_MAP.put(user.getPhone(), user);
    }

    public boolean contains(final String phone) {
        return USERS_MAP.containsKey(phone);
    }

    public User find(final String phone) {
        return USERS_MAP.get(phone);
    }
}
```

Spróbujmy teraz napisać test przy wykorzystaniu **BDDMockito**:

```java
package pl.zajavka.bdd;

import org.junit.jupiter.api.*;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.*;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class UserRegistryServiceTest {

    @InjectMocks
    private UserRegistryService userRegistryService;

    @Mock
    private UsersRepository usersRepository;

    @Test
    @DisplayName("Adding users to the registry works successfully")
    void test1() {
        final var user = User.builder()
            .name("Roman")
            .surname("Adamski")
            .phone("12345")
            .build();

        BDDMockito.given(usersRepository.contains(user.getPhone())).willReturn(false);
```

```
        userRegistryService.register(user);

        BDDMockito.then(usersRepository)
            .should()
            .put(user);
    }
```

cd.

```
    @Test
    @DisplayName("Adding users with the same phone numbers twice fails")
    void test2() {
        final var user = User.builder()
            .name("Stefan")
            .surname("Zajavkowy")
            .phone("12345")
            .build();

        BDDMockito.given(usersRepository.contains(user.getPhone())).willReturn(true);

        try {
            userRegistryService.register(user);
            Assertions.fail("Should throw exception");
        } catch (RuntimeException ignore) {}

        BDDMockito.then(usersRepository)
            .should(Mockito.never())
            .put(user);
    }
```

cd.

```
    @Test
    @DisplayName("Adding users with too long phone number fails")
    void test3() {
        final var user = User.builder()
            .name("Stefan")
            .surname("Zajavkowy")
            .phone("12345")
            .build();

        BDDMockito.given(usersRepository.contains(user.getPhone())).willReturn(false);
        BDDMockito.willThrow(new RuntimeException("Phone number is too long"))
            .given(usersRepository)
            .put(user);

        try {
            userRegistryService.register(user);
            Assertions.fail("Should throw exception");
        } catch (RuntimeException ignore) {}

        BDDMockito.then(usersRepository)
            .should()
            .put(user);
    }
}
```

Mockito jest tylko jednym z przykładów jak można pisać testy stosując podejście **BDD**. Na rynku istnieje więcej możliwości/technik/bibliotek, które będą służyły do pisania testów w podejściu BDD.

Mamy nadzieję, że odniesienie do BDD zobrazowało Ci, że można mieć wiele różnych podejść do pisania testów. To jak finalnie będzie skonstruowana aplikacja, jest kwestią podjętych decyzji. Natomiast, żeby podjąć dobre decyzje, trzeba mieć świadomość jakie mamy możliwości i jakie są różne sposoby podejścia do określonego zagadnienia - dlatego właśnie poruszyliśmy tematykę BDD.

Mockito FAQ

Mockito jest jednym z wielu rozwiązań, które będą pokazywane na ścieżce Zajavkowej. Jak możesz się domyślić, nie omawiamy każdego dostępnego zakamarka każdego rozwiązania, skupiamy się na najważniejszych aspektach. Oznacza to, że w praktyce przy pracy z jakimkolwiek narzędziem lub biblioteką, prędzej czy później czeka Cię praca z dokumentacją w celu poznania jakiejś funkcjonalności, której nie znasz i nie wiesz jak się zachowa. Z tego powodu umieszczamy link do Wiki Mockito. Pamiętaj również, że jeżeli chcesz się dowiedzieć jak dokładnie działa dana funkcjonalność, wykorzystaj `CTRL + Q`.