Paweł Kiełbasa, Wojciech Kosztyła

Sprawozdanie PO – Testy

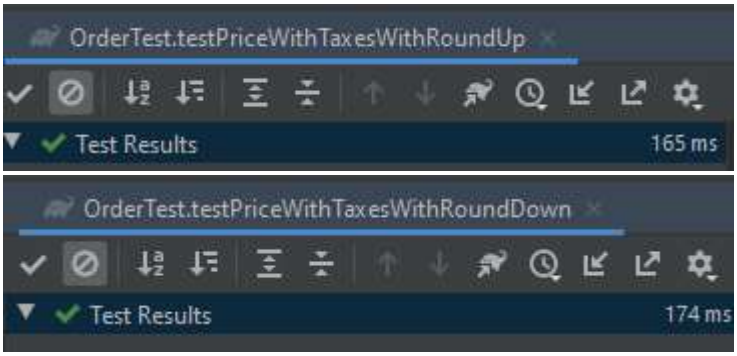
1. Zmiana wartości podatku na 23% - Zmiana wartości w klasie Order

```
public class Order {
    private static final BigDecimal TAX_VALUE = BigDecimal.valueOf(1.23);
```

Na potrzebę sprawdzenia zaokrąglania kwoty produktu w testach zdodano przypadki w sytacjach brzegowych

```
@Test
public void testPriceWithTaxesWithRoundDown() {
    // given

    // when
    Order order = getOrderWithCertainProductPrice( productPriceValue: 0.01); // 0.01 PLN
    Order order1 =getOrderWithCertainProductPrice( productPriceValue: 1.15); //1.15 PLN

    // then
    assertBigDecimalCompareValue(order.getPriceWithTaxes(), BigDecimal.valueOf(0.01)); // 0.01 PLN
    assertBigDecimalCompareValue(order1.getPriceWithTaxes(), BigDecimal.valueOf(1.41)); // 1.15 *1,23 = 1,4145
}

@Test
public void testPriceWithTaxesWithRoundUp() {
    // given

    // when
    Order order = getOrderWithCertainProductPrice( productPriceValue: 0.03); // 0.03 PLN
    Order order2 = getOrderWithCertainProductPrice( productPriceValue: 1.11);
    // then
    assertBigDecimalCompareValue(order.getPriceWithTaxes(), BigDecimal.valueOf(0.04)); // 0.04 PLN
    assertBigDecimalCompareValue(order2.getPriceWithTaxes(),BigDecimal.valueOf(1.37)); // 1.11*1.23 = 1.3653
}
```

Rezultat




2. Dodanie możliwości posiadania wielu produktów w jednym zamówieniu. W celu zaimplementowania tej funkcjonalności został zmieniony typ pola products w klasie Order

```
private final List<Product> products;
```

Zmiana w kostruktorze

```java
public Order(List<Product> products) {
    this.products = products;
    id = UUID.randomUUID();
    paid = false;
}
```

W związku ze zmianą typu pola product musiały zostać zmienione metody getProducts i getPrice

```java
public List<Product> getProducts() { return products; }
```

```java
public BigDecimal getPrice() {
    return products.stream().map(Product::getPrice).reduce(BigDecimal.ZERO,BigDecimal::add);
}
```

W związku z powyższymi zmianami wszystkie testy zawierające konstrunktor klasy Order przyjmujący jako argument Product musiały zostać zmienione. Jeden z przykładów poniżej

```java
private Order getOrderWithMockedProduct() {
    Product product = mock(Product.class);
    return new Order(Collections.singletonList(product));
}
```

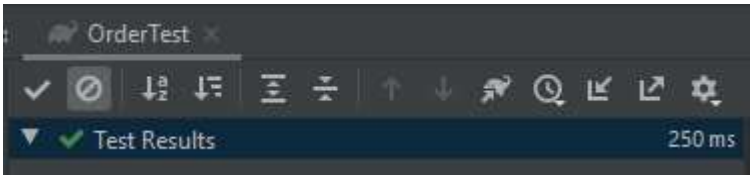
Oprócz tego dodano test sprawdzający cenę zamówienia z większą ilością produktów

```
private Order getOrderWithCertainProductPrices(List<Double> productPriceValues) {

    List<Product> products = productPriceValues.stream() Stream<Double>
            .map(BigDecimal::valueOf) Stream<BigDecimal>
            .map(val -> {
                Product product = mock(Product.class);
                given(product.getPrice()).willReturn(val);
                return product;
            }) Stream<Product>
            .collect(Collectors.toList());

    return new Order(products);
}
@Test
public void testGetPriceWhenMoreItems() throws Exception {
    // given
    List<Double> prices = new ArrayList<>(Arrays.asList(0.7, 0.5, 0.25));
    Order order = getOrderWithCertainProductPrices(prices);

    // when
    BigDecimal actualProductPrice = order.getPrice();
    BigDecimal expectedProductPrice = prices.stream()
            .map(BigDecimal::valueOf)
            .reduce(BigDecimal.ZERO, BigDecimal::add);

    // then
    assertBigDecimalCompareValue(expectedProductPrice, actualProductPrice);
}
```

Aby zapobiec próbie wstawienia pustej listy produktów do zamówienia należy zmianić kostruktor klasy Order

```
public Order(List<Product> products) {
    this.products = Objects.requireNonNull(products, message: "Invalid product list");
    this.products.forEach((p)-> Objects.requireNonNull(p, message: "Invalid product"));
    id = UUID.randomUUID();
    paid = false;
}
```

Wyniki testów



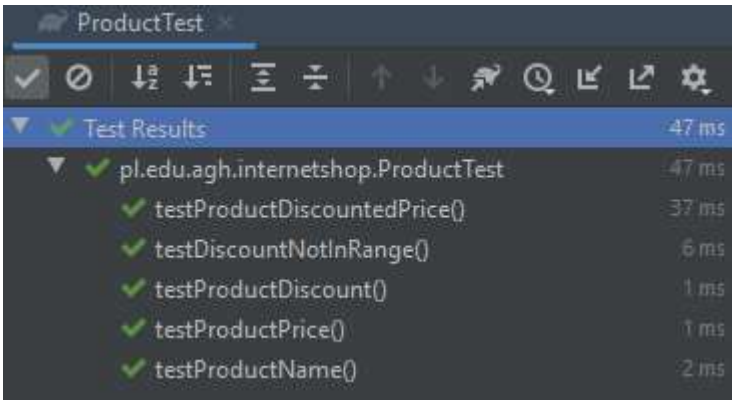

3. Dodanie rabatu na całe zamówienie i pojedyncze produkty. Wyamga to dodania nowego pola w klasie Product oraz getterów i seterów

```java
private final String name;
private final BigDecimal price;
private  BigDecimal discount; // 0 - no discount, 1 - full discount

public Product(String name, BigDecimal price, BigDecimal discount) {
    if(discount.doubleValue()>1 || discount.doubleValue()<0) throw new IllegalArgumentException("Illegal discount value");
    this.name = name;
    this.price = price;
    this.discount=discount;
    this.price.setScale(PRICE_PRECISION, ROUND_STRATEGY);
}
public void setDiscount(BigDecimal DISCOUNT){
    if(discount.doubleValue()>1 || discount.doubleValue()<0) throw new IllegalArgumentException("Illegal discount value");
    this.discount = DISCOUNT;
}

public BigDecimal getDiscount() {
    return discount;
}

public BigDecimal getPrice(){
    return  price.multiply(BigDecimal.valueOf(1).subtract(discount)).setScale(PRICE_PRECISION,ROUND_STRATEGY);
}
```

W związku z dodaniem nowej funkcjonalności stworzyłem dodatkowe testy na sprawdzenie czy dodatkowe metody działają

```java
@Test
public void testDiscountNotInRange() throws Exception{
    //given

    //when

    //then
    assertThrows(IllegalArgumentException.class, () -> new Product(NAME,PRICE,BigDecimal.valueOf(1.5)));
    assertThrows(IllegalArgumentException.class, () -> new Product(NAME,PRICE,BigDecimal.valueOf(-0.1)));
}

@Test
public void testProductDiscount() throws Exception{
    //given

    //when
    Product product = new Product(NAME,PRICE,DISCOUNT);

    //then
    assertBigDecimalCompareValue(product.getDiscount(),DISCOUNT);
}

@Test
public void testProductDiscountedPrice() throws Exception{
    //given

    //when
    Product product = new Product(NAME,PRICE,DISCOUNT);

    //then
    assertBigDecimalCompareValue(product.getPrice(), PRICE.multiply(BigDecimal.valueOf(1).subtract(DISCOUNT)));
}
```

Otrzymano następujące wyniki testów




Dodanie możliwości posiadania zniżki w klasie Order (wymaga to zmiany w konstruktorze oraz dodania gettera i setera dla danego pola).

```
private BigDecimal discount;

public Order(List<Product> products) {
    this.products = Objects.requireNonNull(products,  message: "Invalid product list");
    this.products.forEach((p)-> Objects.requireNonNull(p, message: "Invalid product"));
    id = UUID.randomUUID();
    paid = false;
    discount = BigDecimal.valueOf(0);
}

public BigDecimal getDiscount() {
    return discount;
}

public void setDiscount(BigDecimal discount) {
    if(discount.doubleValue() <0 || discount.doubleValue()>1) throw new IllegalArgumentException("Illegal discount value");
    this.discount = discount;
}
```

W związku z powyższymi modyfikacjami metody getPrice i getPriceWithTaxes musiały zostać zmienione

```
public  BigDecimal getPrice(){
    BigDecimal productPrice = products.stream().map(Product::getPrice).reduce(BigDecimal.ZERO,BigDecimal::add);
    return  productPrice.multiply(BigDecimal.valueOf(1).subtract(getDiscount()));
}

public BigDecimal getPriceWithTaxes() {
    return getPrice().multiply(TAX_VALUE).setScale(Product.PRICE_PRECISION, Product.ROUND_STRATEGY);
}
```

Do klasy OrderTest dodałem następujące testy aby sprawdzić działanie nowych funkcji. Wymagało to dodania metody dodawania zamówienia z określonymi kwotami produktów oraz zniżką ( na potrzeby testów założyłem że zniżka dla każdego produktu jest taka sama

Test gettera i setera dla zniżki

```java
@Test
public void testInvalidDiscount(){
    // given
    Order order = getOrderWithMockedProduct();

    // when

    // then
    assertThrows(IllegalArgumentException.class ,()-> order.setDiscount(BigDecimal.valueOf(1.5)));
    assertThrows(IllegalArgumentException.class, () -> order.setDiscount(BigDecimal.valueOf(-0.2)));
}

@Test
public void testGetDiscount(){
    //given
    Order order = getOrderWithMockedProduct();
    BigDecimal DISCOUNT = BigDecimal.valueOf(0.75);

    //when
    order.setDiscount(DISCOUNT);

    //then
    assertBigDecimalCompareValue(order.getDiscount(),DISCOUNT);
}
```

Metody tworzące nowe zamówienie na podstawie danych cen i wartości zniżki

```java
private Order getOrderWithCertainProductPrices(List<Double> productPriceValues) {

    return getOrderWithCertainProductPricesAndDiscount(productPriceValues, BigDecimal.valueOf(0));
}
private Order getOrderWithCertainProductPricesAndDiscount(List<Double> productPriceValues, BigDecimal discount) {

    List<Product> products = productPriceValues.stream() Stream<Double>
            .map(BigDecimal::valueOf) Stream<BigDecimal>
            .map(val -> {
                Product product = mock(Product.class);
                given(product.getPrice()).willReturn(val.multiply(BigDecimal.valueOf(1).subtract(discount)));
                return product;
            }) Stream<Product>
            .collect(Collectors.toList());

    return new Order(products);
}
```

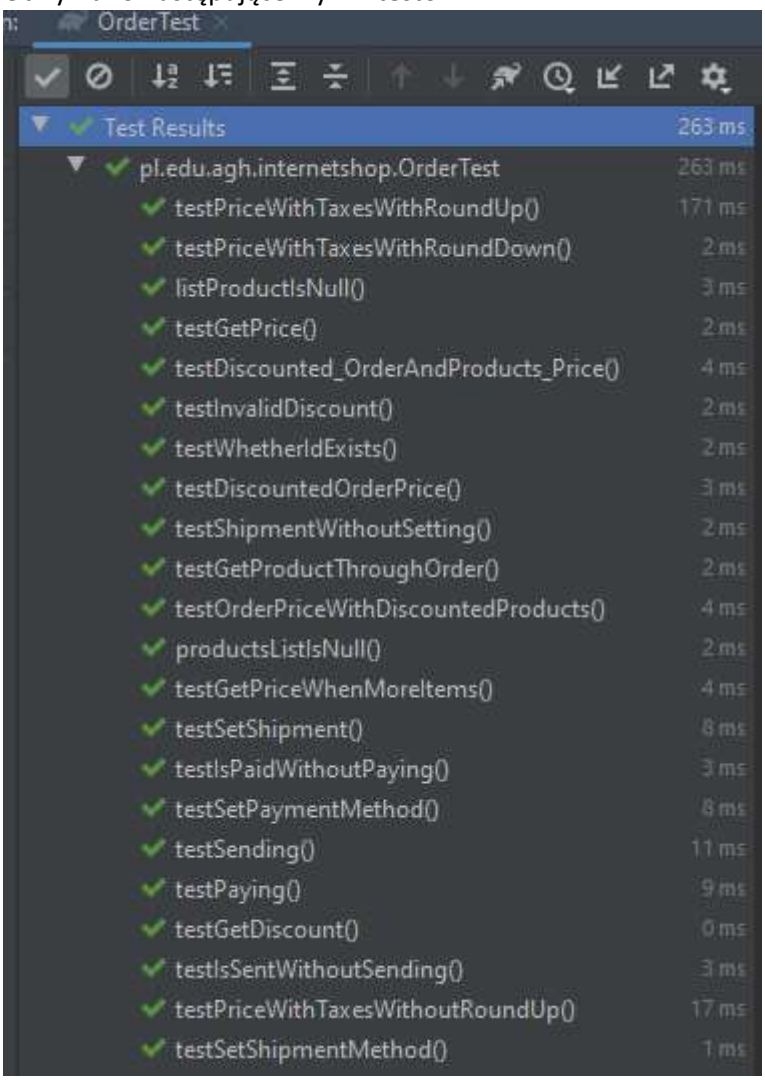
Testy sprawdzające poprawność naliczania zniżek

```java
@Test
public  void testOrderPriceWithDiscountedProducts() throws Exception{
    //given
    List<Double> prices = new ArrayList<>(Arrays.asList(1.0, 2.5, 3.5));
    BigDecimal DISCOUNT = BigDecimal.valueOf(0.3);
    Order order = getOrderWithCertainProductPricesAndDiscount(prices, DISCOUNT);

    //when
    BigDecimal actualProductPrice = order.getPrice();
    BigDecimal expectedProductPrice = prices.stream() Stream<Double>
            .map(BigDecimal::valueOf) Stream<BigDecimal>
            .map(val -> val.multiply(BigDecimal.valueOf(1).subtract(DISCOUNT))) Stream<BigDecimal>
            .reduce(BigDecimal.ZERO, BigDecimal::add);

    //then
    assertBigDecimalCompareValue(actualProductPrice, expectedProductPrice);

}
```

```java
@Test
public  void testDiscountedOrderPrice() throws Exception{
    //given
    List<Double> prices = new ArrayList<>(Arrays.asList(1.0, 2.5, 3.5));
    Order order = getOrderWithCertainProductPrices(prices);
    BigDecimal orderDiscount = BigDecimal.valueOf(0.2);

    //when
    order.setDiscount(orderDiscount);
    BigDecimal actualProductPrice = order.getPrice();
    BigDecimal expectedProductPrice = prices.stream() Stream<Double>
            .map(BigDecimal::valueOf) Stream<BigDecimal>
            .reduce(BigDecimal.ZERO, BigDecimal::add) BigDecimal
            .multiply(BigDecimal.valueOf(1).subtract(orderDiscount));

    //then
    assertBigDecimalCompareValue(actualProductPrice, expectedProductPrice);

}
```

```java
@Test
public void testDiscounted_OrderAndProducts_Price() throws Exception{
    //given
    List<Double> prices = new ArrayList<>(Arrays.asList(1.0, 2.5, 3.5));
    BigDecimal productDiscount = BigDecimal.valueOf(0.3);
    Order order = getOrderWithCertainProductPricesAndDiscount(prices,productDiscount);
    BigDecimal orderDiscount = BigDecimal.valueOf(0.2);

    //when
    order.setDiscount(orderDiscount);
    BigDecimal actualProductPrice = order.getPrice();
    BigDecimal expectedProductPrice = prices.stream() Stream<Double>
            .map(BigDecimal::valueOf) Stream<BigDecimal>
            .map(val -> val.multiply(BigDecimal.valueOf(1).subtract(productDiscount))) Stream<BigDecimal>
            .reduce(BigDecimal.ZERO, BigDecimal::add) BigDecimal
            .multiply(BigDecimal.valueOf(1).subtract(orderDiscount));

    //then
    assertBigDecimalCompareValue(actualProductPrice, expectedProductPrice);
}
```

Otrzymano następujące wyniki testów



4. Umożliwienie przechowywania historii zamówień z wyszukiwaniem po: nazwie produktu, kwocie zamówienia, nazwisku zamawiającego. Wyszukać można przy użyciu jednego lub wielu kryteriów.
   (już tworzyliśmy filtry i na tamtych filtrach się bazowaliśmy)
   a. Stworzyłem interfejs filtrów

```java
package pl.edu.agh.internetshop;

public interface SearchStrategy {
    boolean filter(Order order);
}
```

   b. Dodałem klasy obsługujące filtrowanie

```java
package pl.edu.agh.internetshop;

public class SearchStrategySurname implements SearchStrategy {
    private String surname;

    public SearchStrategySurname(String surname) {
        this.surname = surname;
    }

    @Override
    public boolean filter(Order order) {
        if (order.getShipment().getRecipientAddress().getName().split( regex: " ")[1] != null) {
            return order.getShipment().getRecipientAddress().getName().split( regex: " ")[1].equals(this.surname);
        }
        return false;
    }
}
```

```java
public class SearchStrategyProductName implements SearchStrategy {
    private String name;

    public SearchStrategyProductName(String name) {
        this.name = name;
    }

    @Override
    public boolean filter(Order order) {
        List<Product> products = order.getProducts();
        for (Product product: products) {
            if (product.getName().equals(this.name)) {
                return true;
            }
        }
        return false;
    }
}
```

```java
public class SearchStrategyPrice implements SearchStrategy {
    BigDecimal price;

    public SearchStrategyPrice(BigDecimal price) {
        this.price = price;
    }

    @Override
    public boolean filter(Order order) {
        return order.getPriceWithTaxes().compareTo(this.price) == 0;
    }
}
```

c. Aby wykorzystać dowolną kombinację filtrów stworzyłem kompozyt SearchStrategyComposite.

```java
public class SearchStrategyComposite implements SearchStrategy {
    private final List<SearchStrategy> filters;

    public SearchStrategyComposite(List<SearchStrategy> filters) {
        this.filters = filters;
    }

    @Override
    public boolean filter(Order order) {
        return filters.stream().allMatch(f -> f.filter(order));
    }
}
```

d. Aby przechowywać wszystkie zamówienia dodałem klasę OrderHistory

```
public class OrderHistory {
    private List<Order> zamowienia = new ArrayList<Order>();

    public OrderHistory(){
    }

    public void addOrder(Order order){
        this.zamowienia.add(order);
    }

    public List<Order> getOrders(){
        return this.zamowienia;
    }

    public List<Order> getSearchResult(SearchStrategy searchStrategy){
        List<Order> temp_zamowienia = new ArrayList<>();
        for(Order order : this.zamowienia){
            if(searchStrategy.filter(order)){
                temp_zamowienia.add(order);
            }
        }
        return temp_zamowienia;
    }

}
```

e. Dodałem klasę OrderHistoryTest, aby testować wszystkie dodane klasy

```
private Order getMockOrder1() {
    Order order = mock(Order.class);
    List<Product> productList = Arrays.asList(
            new Product( name: "Chipsy", BigDecimal.valueOf(4.49), BigDecimal.valueOf(0.5)),
            new Product( name: "Pepsi", BigDecimal.valueOf(5.39), BigDecimal.valueOf(0.39))
    );
    given(order.getProducts()).willReturn(productList);
    return order;
}

@Test
public void productIsInTheOrder() {
    // given
    Order order = getMockOrder1();
    // when
    SearchStrategyProductName searchStrategy = new SearchStrategyProductName("Chipsy");
    // then
    assertTrue(searchStrategy.filter(order));
}

@Test
public void productIsNotInTheOrder() {
    // given
    Order order = getMockOrder1();
    // when
    SearchStrategyProductName searchStrategy = new SearchStrategyProductName("mieso");
    // then
    assertFalse(searchStrategy.filter(order));
}
```

i. productIsInTheOrder – sprawdza, czy filtr z ProductName działa
ii. productIsNotInTheOrder – analogicznie, czy filtr działa i „mieso” nie należy do Orderu

f.

```java
@Test
public void sameSurname() {
    // given
    Order order = mock(Order.class);
    given(order.getPayerSurname()).willReturn("Surname");
    // when
    SearchStrategySurname searchStrategy = new SearchStrategySurname("Surname");
    // then
    assertTrue(searchStrategy.filter(order));
}

@Test
public void notTheSameSurname() {
    // given
    Order order = mock(Order.class);
    given(order.getPayerSurname()).willReturn("notSurname");
    // when
    SearchStrategySurname searchStrategy = new SearchStrategySurname("Surname");
    // then
    assertFalse(searchStrategy.filter(order));
}

@Test
public void surnameIsNull() {
    // given
    Order order = mock(Order.class);
    given(order.getPayerSurname()).willReturn(null);
    // when
    SearchStrategySurname searchStrategy = new SearchStrategySurname("Surname");
    // then
    assertFalse(searchStrategy.filter(order));
}
```

i. sameSurname – sprawdza działanie filtru „surname”, czy funkcje zwrócą znaleziony wynik
ii. notTheSameSurname – analogicznie do tego wyżej
iii. surnameIsNull – sprawdza działanie programu na wypadek podania do filtru wartość null

g.

```
private Order getMockOrder2() {
    Order order = mock(Order.class);
    given(order.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(10));
    return order;
}

@Test
public void samePrice() {
    // given
    Order order = getMockOrder2();
    // when
    SearchStrategyPrice searchStrategy = new SearchStrategyPrice(BigDecimal.valueOf(10));
    // then
    assertTrue(searchStrategy.filter(order));
}

@Test
public void notTheSamePrice() {
    // given
    Order order = getMockOrder2();
    // when
    SearchStrategyPrice searchStrategy = new SearchStrategyPrice(BigDecimal.valueOf(9));
    // then
    assertFalse(searchStrategy.filter(order));
}
```

   i. samePrice – sprawdza, czy filtr z „price” zadziała i pokaże właśnie dodany Order
   ii. notTheSamePrice – na odwrót – sprawdza czy filtr działa i czy przy podanym błędnym polu coś zwróci

h.

```
private Order getMockOrder3() {
    Order order = mock(Order.class);
    List<Product> productList = Arrays.asList(
            new Product( name: "Chipsy", BigDecimal.valueOf(4.49), BigDecimal.valueOf(0.5)),
            new Product( name: "Pepsi", BigDecimal.valueOf(5.39), BigDecimal.valueOf(0.39))
    );
    given(order.getProducts()).willReturn(productList);
    given(order.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(10));
    given(order.getPayerSurname()).willReturn("Surname");
    return order;
}

@Test
public void sameParameters() {
    // given
    Order order = getMockOrder3();
    SearchStrategy productNameSearchStrategy = new SearchStrategyProductName("Chipsy");
    SearchStrategy payersSurnameSearchStrategy = new SearchStrategySurname("Surname");
    SearchStrategy totalPriceSearchStrategy = new SearchStrategyPrice(BigDecimal.valueOf(10));

    // when
    SearchStrategyComposite searchStrategy = new SearchStrategyComposite(
            Arrays.asList(productNameSearchStrategy, payersSurnameSearchStrategy, totalPriceSearchStrategy));

    // then
    assertTrue(searchStrategy.filter(order));
}
```

```java
@Test
public void sameParametersButProductName() {
    // given
    Order order = getMockOrder3();
    SearchStrategy productNameSearchStrategy = new SearchStrategyProductName("mieso");
    SearchStrategy payersSurnameSearchStrategy = new SearchStrategySurname("Surname");
    SearchStrategy totalPriceSearchStrategy = new SearchStrategyPrice(BigDecimal.valueOf(10));

    // when
    SearchStrategyComposite searchStrategy = new SearchStrategyComposite(
            Arrays.asList(productNameSearchStrategy, payersSurnameSearchStrategy, totalPriceSearchStrategy));

    // then
    assertFalse(searchStrategy.filter(order));
}

@Test
public void sameParametersButSurname() {
    // given
    Order order = getMockOrder3();
    SearchStrategy productNameSearchStrategy = new SearchStrategyProductName("Pepsi");
    SearchStrategy payersSurnameSearchStrategy = new SearchStrategySurname("notSurname");
    SearchStrategy totalPriceSearchStrategy = new SearchStrategyPrice(BigDecimal.valueOf(10));

    // when
    SearchStrategyComposite searchStrategy = new SearchStrategyComposite(
            Arrays.asList(productNameSearchStrategy, payersSurnameSearchStrategy, totalPriceSearchStrategy));

    // then
    assertFalse(searchStrategy.filter(order));
}

@Test
public void sameParametersButPrice() {
    // given
    Order order = getMockOrder3();
    SearchStrategy productNameSearchStrategy = new SearchStrategyProductName("Pepsi");
    SearchStrategy payersSurnameSearchStrategy = new SearchStrategySurname("Surname");
    SearchStrategy totalPriceSearchStrategy = new SearchStrategyPrice(BigDecimal.valueOf(9));

    // when
    SearchStrategyComposite searchStrategy = new SearchStrategyComposite(
            Arrays.asList(productNameSearchStrategy, payersSurnameSearchStrategy, totalPriceSearchStrategy));

    // then
    assertFalse(searchStrategy.filter(order));
}
```

i. sameParameters – test wspólnego działania wszystkich filtrów
ii. sameParametersBut * - testy każdej kombinacji dwóch filtrów

i.

```java
@Test
void getFewOrdersFromOrderHistory() {
    // given
    List<Order> orders = Arrays.asList(mock(Order.class), mock(Order.class));

    // when
    OrderHistory OrderHistory = new OrderHistory(orders);

    // then
    assertEquals( expected: 2, OrderHistory.getOrders().size());
    assertSame(orders.get(0), OrderHistory.getOrders().get(0));
    assertSame(orders.get(1), OrderHistory.getOrders().get(1));
}
```

```java
@Test
void getSearchResultsWithProductName() {
    // given
    Product product_0 = mock(Product.class);
    Product product_1 = mock(Product.class);
    Product product_2 = mock(Product.class);
    Product product_3 = mock(Product.class);

    given(product_0.getName()).willReturn("mieso");
    given(product_1.getName()).willReturn("befsztyk");
    given(product_2.getName()).willReturn("kaszanka");
    given(product_3.getName()).willReturn("pasztetowa");

    Order order = mock(Order.class);
    Order order1 = mock(Order.class);
    Order order2 = mock(Order.class);

    given(order.getProducts()).willReturn(Arrays.asList(product_0, product_1));
    given(order1.getProducts()).willReturn(Arrays.asList(product_1, product_3));
    given(order2.getProducts()).willReturn(Arrays.asList(product_0, product_1, product_2, product_3));

    SearchStrategy searchStrategy = new SearchStrategyProductName("mieso");

    // when
    OrderHistory OrderHistory = new OrderHistory(Arrays.asList(order, order1, order2));

    // then
    assertEquals( expected: 2, OrderHistory.getSearchResult(searchStrategy).size());
    assertSame(order, OrderHistory.getSearchResult(searchStrategy).get(0));
    assertSame(order2, OrderHistory.getSearchResult(searchStrategy).get(1));
}
```

```java
@Test
void getSearchResultsWithSurname() {
    // given
    Order order = mock(Order.class);
    Order order1 = mock(Order.class);
    Order order2 = mock(Order.class);

    given(order.getPayerSurname()).willReturn("Surname1");
    given(order1.getPayerSurname()).willReturn("Surname2");
    given(order2.getPayerSurname()).willReturn("Surname1");

    SearchStrategy searchStrategy = new SearchStrategySurname("Surname1");

    // when
    OrderHistory OrderHistory = new OrderHistory(Arrays.asList(order, order1, order2));

    // then
    assertEquals( expected: 2, OrderHistory.getSearchResult(searchStrategy).size());
    assertSame(order, OrderHistory.getSearchResult(searchStrategy).get(0));
    assertSame(order2, OrderHistory.getSearchResult(searchStrategy).get(1));
}
@Test
void getSearchResultsWithPrice() {
    // given
    Order order = mock(Order.class);
    Order order1 = mock(Order.class);
    Order order2 = mock(Order.class);

    given(order.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(10));
    given(order1.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(20));
    given(order2.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(10));

    SearchStrategy searchStrategy = new SearchStrategyPrice(BigDecimal.valueOf(10));

    // when
    OrderHistory OrderHistory = new OrderHistory(Arrays.asList(order, order1, order2));

    // then
    assertEquals( expected: 2, OrderHistory.getSearchResult(searchStrategy).size());
    assertSame(order, OrderHistory.getSearchResult(searchStrategy).get(0));
    assertSame(order2, OrderHistory.getSearchResult(searchStrategy).get(1));
}
```

```java
@Test
void getCompositeSearchResults() {
    // given
    Product product = mock(Product.class);
    Product product_1 = mock(Product.class);
    Product product_2 = mock(Product.class);
    Product product_3 = mock(Product.class);

    given(product.getName()).willReturn("mieso");
    given(product_1.getName()).willReturn("befsztyk");
    given(product_2.getName()).willReturn("kaszanka");
    given(product_3.getName()).willReturn("pasztetowa");

    Order order = mock(Order.class);
    Order order1 = mock(Order.class);
    Order order2 = mock(Order.class);

    given(order.getProducts()).willReturn(Arrays.asList(product, product_1, product_3));
    given(order1.getProducts()).willReturn(Arrays.asList(product_1, product_3));
    given(order2.getProducts()).willReturn(Arrays.asList(product, product_1, product_2, product_3));

    given(order.getPayerSurname()).willReturn("Surname1");
    given(order1.getPayerSurname()).willReturn("Surname2");
    given(order2.getPayerSurname()).willReturn("Surname1");

    given(order.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(20));
    given(order1.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(10));
    given(order2.getPriceWithTaxes()).willReturn(BigDecimal.valueOf(10));

    SearchStrategy searchStrategy = new SearchStrategyComposite(Arrays.asList(
            new SearchStrategyProductName("pasztetowa"),
            new SearchStrategySurname("Surname1"),
            new SearchStrategyPrice(BigDecimal.valueOf(10))
    ));

    // when
    OrderHistory OrderHistory = new OrderHistory(Arrays.asList(order, order1, order2));

    // then
    assertEquals( expected: 1, OrderHistory.getSearchResult(searchStrategy).size());
    assertSame(order2, OrderHistory.getSearchResult(searchStrategy).get(0));
}

@Test
public void searchStrategyIsNull() {
    // given

    // when
    OrderHistory OrderHistory = new OrderHistory(Arrays.asList(mock(Order.class), mock(Order.class)));

    // then
    assertThrows(NullPointerException.class, () -> OrderHistory.getSearchResult( searchStrategy: null));
}
```

i. getFewOrdersFromOrdersHistory – sprawdza poprawne działanie historii Order'ów
ii. getSearchResultsWith* - sprawdzają zachowanie programu, gdy muszą obsłużyć więcej niż jeden filtr każdego typu

iii. getCompositeSearchResults – sprawdza działanie getSearchResults przy filtrach na wszystkich polach
iv. searchStrategyIsNull – sprawdza zachowanie programu w przypadku podania nulowego / nie podania sposobu filtracji

5. Wyniki testów: