Testy jednostkowe

W tym laboratorium jeśli chodzi o podejście do testów jednostkowych wykorzystano technikę TDD (Test-Driven Development), czyli programista najpierw pisze test a potem przystępuje do implementacji i ewentualnej refaktoryzacji powstałego kodu.

Przed rozpoczęciem dokonywania zmian w kodzie uruchomiłem testy, aby sprawdzić, czy będę pracować z działającym kodem. Wszystkie przeszły pomyślnie.

Zadania

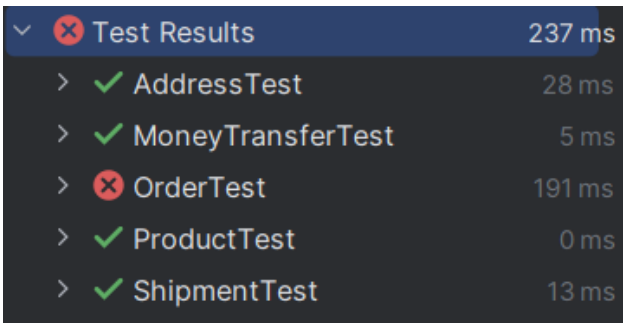
1. **Zmienić wartość procentową naliczonego podatku z 22% na 23%. Należy zweryfikować przypadki brzegowe przy zaokrągleniach.**

- zmieniono kod testu (z 2.44 na 2.46):

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

      // when
      Order order = getOrderWithCertainProductPrice(2); // 2 PLN

      // then
      assertBigDecimalCompareValue(order.getPriceWithTaxes(),
BigDecimal.valueOf(2.46)); // 2.46 PLN
}
```

Nie było potrzeby modyfikować innych metod testujących pod implementowaną funkcjonalność.

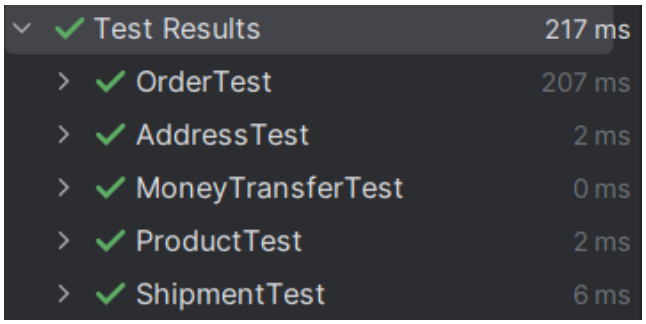
- teraz test kończy się niepowodzeniem:



- zmieniono kod klasy Order pod zmodyfikowane testy:

```
private static final BigDecimal TAX_VALUE = BigDecimal.valueOf(1.23);
```

- test kończy się powodzeniem:



2. **Rozszerzyć funkcjonalność systemu, tak aby zamówienie mogło obejmować więcej niż jeden produkt na raz.**

- zmieniono kod prywatnej metody, która tworzy mocki klasy Product tak, aby zwracała obiekt klasy Order, który przyjmuje w konstruktorze listę produktów:

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

- zmieniono kod i nazwę interesującej nas metody testującej:

```
@Test
public void testGetProductsThroughOrder() {
      // given
      Product expectedProduct1 = mock(Product.class);
      Product expectedProduct2 = mock(Product.class);
      Product expectedProduct3 = mock(Product.class);
      List<Product> expectedProducts = Arrays.asList(expectedProduct1,
expectedProduct2, expectedProduct3)
      Order order = new Order(expectedProducts);

      // when
      List<Product> actualProducts = order.getProduct();
```

```
        // then
        assertSame(expectedProducts, actualProducts);
}
```

- dodano także nowy test sprawdzający, czy lista produktów nie jest nullem:

```
@Test
public void testProductListNullValue() {
        // given, when, then
        assertThrows(NullPointerException.class, () -> new Order(null));
}
```

Testy jednostkowe kończyły się niepowodzeniem, więc zaczęto modyfikować klasę Order:

- zmieniono atrybut klasy:

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

- konstruktor:

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

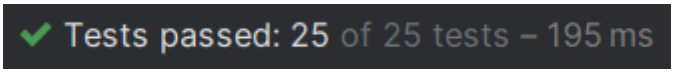
- oraz metodę obliczającą cenę zamówienia i zwracającą listę produktów:

```
public BigDecimal getPrice() {
    BigDecimal sum = new BigDecimal(0);
    for (Product product : this.products) {
        sum = sum.add(product.getPrice());
    }
    return sum;
}
```

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

```
}
```

- testy kończą się powodzeniem:



3. **Dodać możliwość naliczania rabatu do pojedynczego produktu i do całego zamówienia.**

Na początku dodano możliwość naliczania rabatu do pojedynczego produktu.

- stworzono klasę Discount zarządzającą zniżkami:

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

import java.math.BigDecimal;

public class Discount {

    private final BigDecimal discountValue;
    public Discount(BigDecimal discountValue) {
        this.discountValue = discountValue;
        if((this.discountValue.compareTo(new BigDecimal(0)) < 0) ||
                (new BigDecimal(1).compareTo(this.discountValue)) < 0) {
            throw new IllegalArgumentException("Discount value has to be in range
from 0 to 1, but was: " + this.discountValue);
        }
    }

    public BigDecimal getDiscountValue() {
        return discountValue;
    }

    public BigDecimal applyDiscount(BigDecimal price) {
        BigDecimal valueToMultiply = new
BigDecimal(1).subtract(this.getDiscountValue());
        return price.multiply(valueToMultiply);
    }
}
```

- dodano metodę testującą stosowanie zniżek dla pojedynczego produktu:

```
@Test
public void testProductDiscount() {
    // given
    Product product = new Product(NAME, PRICE, new Discount(new
BigDecimal(0.5)));

    // when
    product.applyDiscount();
    BigDecimal discountedPrice = product.getPrice();

    // then
    assertBigDecimalCompareValue(BigDecimal.valueOf(0.5), discountedPrice);
}
```

- dodano jeszcze kilka metod testujących sprawdzających różne przypadki brzegowe:

```
@Test
public void testProductWithFullDiscount() {
    // given
    Product product = new Product(NAME, PRICE, new Discount(new
BigDecimal(1)));

    // when
    product.applyDiscount();
    BigDecimal discountedPrice = product.getPrice();

    // then
    assertBigDecimalCompareValue(BigDecimal.valueOf(0),
discountedPrice);
}

@Test
public void testProductWithoutDiscount() {
    // given
    Product product = new Product(NAME, PRICE, new Discount(new
BigDecimal(0)));

    // when
    product.applyDiscount();
```

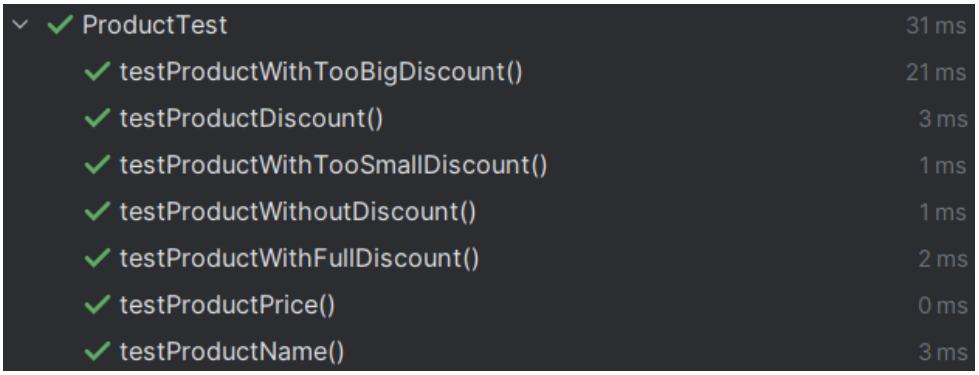
```
    BigDecimal discountedPrice = product.getPrice();

    // then
    assertBigDecimalCompareValue(PRICE, discountedPrice);
}

@Test
public void testPriceWithTooBigDiscount() {
    assertThrows(IllegalArgumentException.class, () -> new
Product(NAME, PRICE, new Discount(new BigDecimal(101))));
}

@Test
public void testPriceWithTooSmallDiscount() {
    assertThrows(IllegalArgumentException.class, () -> new
Product(NAME, PRICE, new Discount(new BigDecimal(-1))));
}
```

- do klasy Product dodano metodę aktualizującą cenę produktu po zastosowaniu zniżki:

```
public void applyDiscount() {
    this.price = this.discount.applyDiscount(this.price);
}
```

- obiekt klasy Discount dodano także jako atrybut w klasie Product oraz jako parametr kostruktora.

- testy klasy Product kończą się powodzeniem:



Teraz przyszła kolej na dodanie rabatu do całego zamówienia.

a. Zmiany w klasie OrderTest:

- dodano prywatną metodę, która szybko tworzy nowe zamówienie wraz z mockowym produktem oraz żądaną ceną produktu:

```
private Order getOrderWithCertainProductPrice(double
productPriceValue) {
	BigDecimal productPrice =
BigDecimal.valueOf(productPriceValue);
	Product product = mock(Product.class);
	given(product.getPrice()).willReturn(productPrice);
	return new Order(Collections.singletonList(product));
}
```

- dodano metodę testującą:

```
@Test
public void testOrderDiscount() {
	// given
	double productPriceValue = 100;
	Order order =
getOrderWithCertainProductPrice(productPriceValue);
	Discount discount = order.getDiscount();

	// when
	BigDecimal expectedOrderPrice = order.getPrice();

	// then
	assertBigDecimalCompareValue(expectedOrderPrice,
discount.applyDiscount(new BigDecimal(productPriceValue)));
}
```

- rozpatrzono także kilka warunków brzegowych oraz przetestowano metodę `setDiscount(Discount discount)` dla zamówień:

```
@Test
public void testOrderWithFullDiscount() {
	// given
	double productPriceValue = 100;
	Order order =
```

```
getOrderWithCertainProductPrice(productPriceValue);
      order.setDiscount(new Discount(new BigDecimal(1)));

      // when
      BigDecimal expectedOrderPrice = order.getPrice();

      // then
      assertBigDecimalCompareValue(BigDecimal.valueOf(0),
expectedOrderPrice);
}

@Test
public void testOrderWithoutDiscount() {
      // given
      double productPriceValue = 100;
      Order order =
getOrderWithCertainProductPrice(productPriceValue);
      order.setDiscount(new Discount(new BigDecimal(0)));

      // when
      BigDecimal expectedOrderPrice = order.getPrice();

      // then
      assertBigDecimalCompareValue(BigDecimal.valueOf(100),
expectedOrderPrice);
}

@Test
public void testSetDiscount() {
      // given
      Order order = getOrderWithMockedProducts();
      Discount expectedDiscount = new Discount(new
BigDecimal("0.2"));

      // when
      order.setDiscount(expectedDiscount);

      // then
      assertSame(order.getDiscount(), expectedDiscount);
}
```

b. zmiany w klasie Order:

- dodano nowy atrybut klasy:

```
private Discount discount;
```

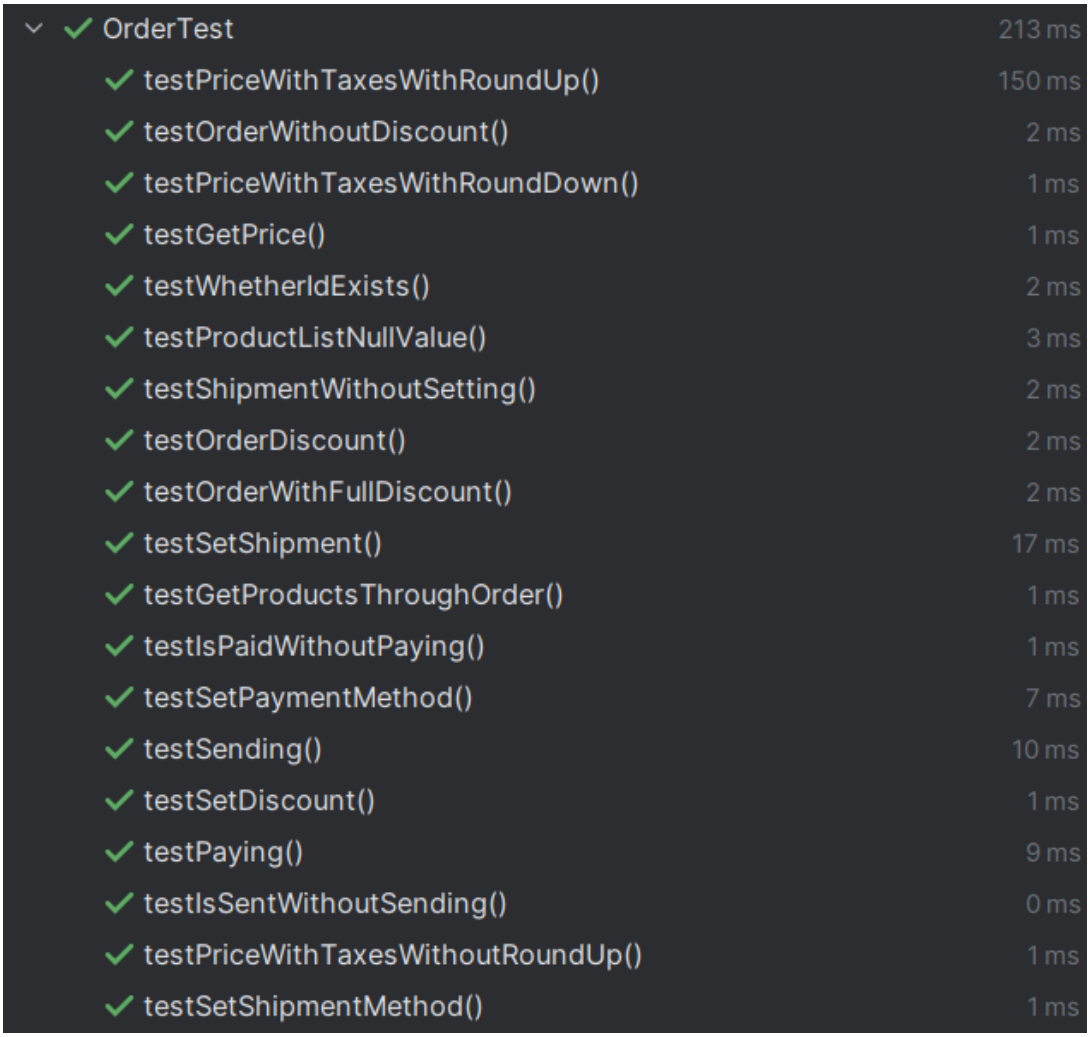
- zmieniono konstruktor:

```
public Order(List<Product> products) {
    this.products = Objects.requireNonNull(products);
    id = UUID.randomUUID();
    paid = false;
    this.discount = new Discount(new BigDecimal(0));
}
```

- dodano getter i setter dla nowego atrybutu:

```
public void setDiscount(Discount discount) {
    this.discount = discount;
}

public Discount getDiscount() {
    return this.discount;
}
```

c. Testy kończą się powodzeniem:



4. **Umożliwić przechowywanie historii zamówień z wyszukiwaniem po: nazwie produktu, kwocie zamówienia, nazwisku zamawiającego. Wyszukiwać można przy użyciu jednego lub wielu kryteriów.**

a. na początku zaimplementowałem podane rodzaje filtrowania:

- klasa ClientNameSearchStrategy:

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

public class ClientNameSearchStrategy implements SearchStrategy {

    private final String clientName;

    public ClientNameSearchStrategy(String clientName) {
        this.clientName = clientName;
```

```
    }

    @Override
    public boolean search(Order order) {
        Shipment shipment = order.getShipment();
        Address recipientAddress = shipment.getRecipientAddress();
        String name = recipientAddress.getName();
        return
order.getShipment().getRecipientAddress().getName().equals(clientN
ame);
    }
}
```

- klasa PriceSearchStrategy:

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

import java.math.BigDecimal;

public class PriceSearchStrategy implements SearchStrategy {

    private final BigDecimal price;

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

    @Override
    public boolean search(Order order) {
        return order.getPrice().equals(price);
    }
}
```

- klasa ProductNameSearchStrategy:

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

public class ProductNameSearchStrategy implements SearchStrategy {

    private final String productName;
```

```
    public ProductNameSearchStrategy(String productName) {
        this.productName = productName;
    }

    @Override
    public boolean search(Order order) {
        return order.getProducts().stream()
                .anyMatch(product ->
product.getName().equals(productName));
    }
}
```

- oraz klasa umożliwiające filtrowanie po wielu kryteriach:

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

import java.util.ArrayList;

public class CompositeSearchStrategy implements SearchStrategy {

    private final ArrayList<SearchStrategy> searchStrategies;

    public CompositeSearchStrategy() {
        this.searchStrategies = new ArrayList<>();
    }

    public void addStrategy(SearchStrategy searchStrategy) {
        searchStrategies.add(searchStrategy);
    }

    @Override
    public boolean search(Order order) {
        return this.searchStrategies
                .stream()
                .allMatch(searchStrategy ->
searchStrategy.search(order));
    }
}
```

- wszystkie powyższe klasy implementują interfejs:

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

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

b. Dodałem testy dla każdego rodzaju filtrowania:

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

import org.junit.jupiter.api.Test;

import java.math.BigDecimal;
import java.util.Collections;

import static org.junit.jupiter.api.Assertions.assertFalse;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.BDDMockito.given;
import static org.mockito.Mockito.mock;

public class SearchStrategyTest {

    @Test
    public void clientNameSearchStrategyTest() {

        // given
        String wantedClientName = "Kowalski";

        Address address1 = mock(Address.class);
        given(address1.getName()).willReturn(wantedClientName);
        Shipment shipment1 = mock(Shipment.class);

given(shipment1.getRecipientAddress()).willReturn(address1);
        Order goodOrder = mock(Order.class);
        given(goodOrder.getShipment()).willReturn(shipment1);

        Address address2 = mock(Address.class);
        given(address2.getName()).willReturn("wrong name");
        Shipment shipment2 = mock(Shipment.class);

given(shipment2.getRecipientAddress()).willReturn(address2);
```

```
        Order badOrder = mock(Order.class);
        given(badOrder.getShipment()).willReturn(shipment2);

        ClientNameSearchStrategy clientNameSearchStrategy = new
ClientNameSearchStrategy(wantedClientName);

        // when, then
        assertTrue(clientNameSearchStrategy.search(goodOrder));
        assertFalse(clientNameSearchStrategy.search(badOrder));
    }

    @Test
    public void priceSearchStrategyTest() {

        // given
        BigDecimal wantedPrice = new BigDecimal(2);

        Product product1 = mock(Product.class);
        given(product1.getPrice()).willReturn(wantedPrice);
        Order goodOrder = new
Order(Collections.singletonList(product1));

        Product product2 = mock(Product.class);

given(product2.getPrice()).willReturn(BigDecimal.valueOf(3.7));
        Order badOrder = new
Order(Collections.singletonList(product2));

        PriceSearchStrategy priceSearchStrategy = new
PriceSearchStrategy(wantedPrice);

        // when, then
        assertTrue(priceSearchStrategy.search(goodOrder));
        assertFalse(priceSearchStrategy.search(badOrder));
    }

    @Test
    public void productNameSearchStrategyTest() {

        // given
        String wantedProductName = "Piłka";
```

```
        Product product1 = mock(Product.class);
        given(product1.getName()).willReturn(wantedProductName);
        Order goodOrder = new
Order(Collections.singletonList(product1));

        Product product2 = mock(Product.class);
        given(product2.getName()).willReturn("Pompka");
        Order badOrder = new
Order(Collections.singletonList(product2));

        ProductNameSearchStrategy productNameSearchStrategy = new
ProductNameSearchStrategy(wantedProductName);

        // when, then
        assertTrue(productNameSearchStrategy.search(goodOrder));
        assertFalse(productNameSearchStrategy.search(badOrder));
    }

    @Test
    public void compositeSearchStrategyTest() {

        // given
        String wantedProductName = "Piłka";
        BigDecimal wantedPrice = new BigDecimal(2);

        ProductNameSearchStrategy productNameSearchStrategy = new
ProductNameSearchStrategy(wantedProductName);
        PriceSearchStrategy priceSearchStrategy = new
PriceSearchStrategy(wantedPrice);
        CompositeSearchStrategy compositeSearchStrategy = new
CompositeSearchStrategy();

        compositeSearchStrategy.addStrategy(priceSearchStrategy);

compositeSearchStrategy.addStrategy(productNameSearchStrategy);

        Product product1 = mock(Product.class);
        given(product1.getName()).willReturn(wantedProductName);
        given(product1.getPrice()).willReturn(wantedPrice);
        Order goodOrder = new
Order(Collections.singletonList(product1));
```

```
        Product product2 = mock(Product.class);
        given(product2.getName()).willReturn("wrong name");
        given(product2.getPrice()).willReturn(wantedPrice);
        Order badOrder1 = new
Order(Collections.singletonList(product2));

        Product product3 = mock(Product.class);
        given(product3.getName()).willReturn(wantedProductName);

given(product3.getPrice()).willReturn(BigDecimal.valueOf(3.7));
        Order badOrder2 = new
Order(Collections.singletonList(product3));

        // when, then
        assertTrue(compositeSearchStrategy.search(goodOrder));
        assertFalse(compositeSearchStrategy.search(badOrder1));
        assertFalse(compositeSearchStrategy.search(badOrder2));
    }
}
```

c. stworzono interfejs `OrderHistoryInterface`, który będzie implementować klasa odpowiedzialna za wyświetlanie historii zamówień:

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

import java.util.List;

public interface OrderHistoryInterface {
    void addOrder(Order order);
    List<Order> findOrderByStrategy(SearchStrategy strategy);
    List<Order> getOrders();
}
```

d. klasa OrderHistory:

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

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
```

```
public class OrderHistory implements OrderHistoryInterface {

    private final List<Order> orders;

    public OrderHistory() {
        this.orders = new ArrayList<>();
    }

    @Override
    public void addOrder(Order order) {
        orders.add(order);
    }

    @Override
    public List<Order> findOrderByStrategy(SearchStrategy
strategy) {
        return this.orders.stream()
                .filter(strategy::search)
                .collect(Collectors.toList());
    }

    @Override
    public List<Order> getOrders() {
        return new ArrayList<>(orders);
    }
}
```

e. Na koniec napisano testy dla klasy OrderHistory:

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

import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import static org.junit.jupiter.api.Assertions.assertEquals;
```

```
import static org.junit.jupiter.api.Assertions.assertSame;
import static org.mockito.BDDMockito.given;
import static org.mockito.Mockito.mock;

public class OrderHistoryTest {

    private static final String customerName1 = "Adam Smith";
    private static final String customerName2 = "Adam Nowak";

    private static final BigDecimal productPrice1 =
BigDecimal.valueOf(1.2);
    private static final BigDecimal productPrice2 =
BigDecimal.valueOf(1.9);

    private static final String productName1 = "Milk";
    private static final String productName2 = "Apple";

    private static Order order1;
    private static Order order2;

    @BeforeAll
    static void addMockOrders() {

        Product product1 = new Product(productName1,
productPrice1, new Discount(new BigDecimal(0)));
        Product product2 = new Product(productName2,
productPrice2, new Discount(new BigDecimal(0)));

        List<Product> list1 = new
ArrayList<>(Arrays.asList(product1, product1, product2));
        List<Product> list2 = new
ArrayList<>(Arrays.asList(product1, product1));


        order1 = new Order(list1);
        order2 = new Order(list2);

        Address address1 = mock(Address.class);
        given(address1.getName()).willReturn(customerName1);

        Address address2 = mock(Address.class);
        given(address2.getName()).willReturn(customerName2);
```

```
        Shipment shipment1 = mock(Shipment.class);

given(shipment1.getRecipientAddress()).willReturn(address1);

        Shipment shipment2 = mock(Shipment.class);

given(shipment2.getRecipientAddress()).willReturn(address2);

        order1.setShipment(shipment1);
        order2.setShipment(shipment2);
    }

    @Test
    public void addAndGetOrderTest() {
        // given
        OrderHistory orderHistory = new OrderHistory();
        Order order = mock(Order.class);

        // when
        orderHistory.addOrder(order);
        Order order1 = orderHistory.getOrders().get(0);

        // then
        assertSame(order, order1);
    }

    @Test
    public void testClientNameSearch() {
        // given
        ClientNameSearchStrategy searchStrategy = new
ClientNameSearchStrategy(customerName1);
        OrderHistory orderHistory = new OrderHistory();
        orderHistory.addOrder(order1);
        orderHistory.addOrder(order2);

        // when
        List<Order> actualOrders =
orderHistory.findOrderByStrategy(searchStrategy);

        // then
        assertEquals(1, actualOrders.size());
```

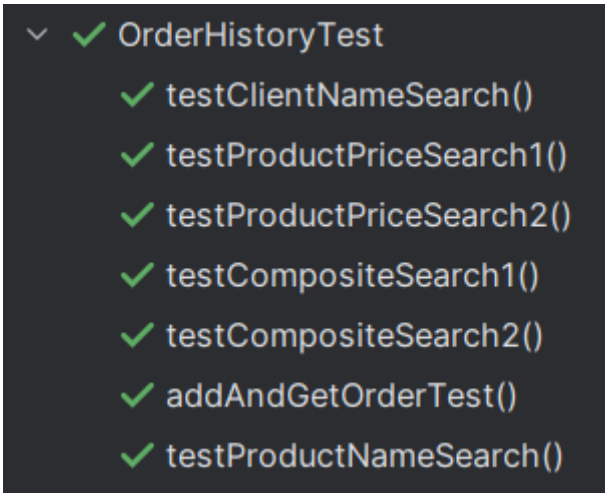
```
        assertEquals(order1, actualOrders.get(0));
    }

    @Test
    public void testProductPriceSearch1() {
        // given
        BigDecimal expectedPrice = BigDecimal.valueOf(1.4);
        PriceSearchStrategy searchStrategy = new
PriceSearchStrategy(expectedPrice);
        OrderHistory orderHistory = new OrderHistory();
        orderHistory.addOrder(order1);
        orderHistory.addOrder(order2);

        // when
        List<Order> actualOrders =
orderHistory.findOrderByStrategy(searchStrategy);

        // then
        assertEquals(0, actualOrders.size());
    }

    @Test
    public void testProductPriceSearch2() {
        // given
        BigDecimal expectedPrice = order1.getPrice();
        PriceSearchStrategy searchStrategy = new
PriceSearchStrategy(expectedPrice);
        OrderHistory orderHistory = new OrderHistory();
        orderHistory.addOrder(order1);
        orderHistory.addOrder(order2);

        // when
        List<Order> actualOrders =
orderHistory.findOrderByStrategy(searchStrategy);

        // then
        assertEquals(1, actualOrders.size());
        assertEquals(order1, actualOrders.get(0));
    }

    @Test
    public void testProductNameSearch() {
```

```
        // given
        ProductNameSearchStrategy searchStrategy = new
ProductNameSearchStrategy(productName2);
        OrderHistory orderHistory = new OrderHistory();
        orderHistory.addOrder(order1);
        orderHistory.addOrder(order2);

        // when
        List<Order> actualOrders =
orderHistory.findOrderByStrategy(searchStrategy);

        // then
        assertEquals(1, actualOrders.size());
        assertEquals(order1, actualOrders.get(0));
    }

    @Test
    public void testCompositeSearch1() {
        // given
        ProductNameSearchStrategy productSearchStrategy = new
ProductNameSearchStrategy(productName2);
        ClientNameSearchStrategy clientSearchStrategy = new
ClientNameSearchStrategy(customerName1);
        CompositeSearchStrategy searchStrategy = new
CompositeSearchStrategy();

        searchStrategy.addStrategy(productSearchStrategy);
        searchStrategy.addStrategy(clientSearchStrategy);

        OrderHistory orderHistory = new OrderHistory();
        orderHistory.addOrder(order1);
        orderHistory.addOrder(order2);

        // when
        List<Order> actualOrders =
orderHistory.findOrderByStrategy(searchStrategy);

        // then
        assertEquals(1, actualOrders.size());
        assertEquals(order1, actualOrders.get(0));
    }
```

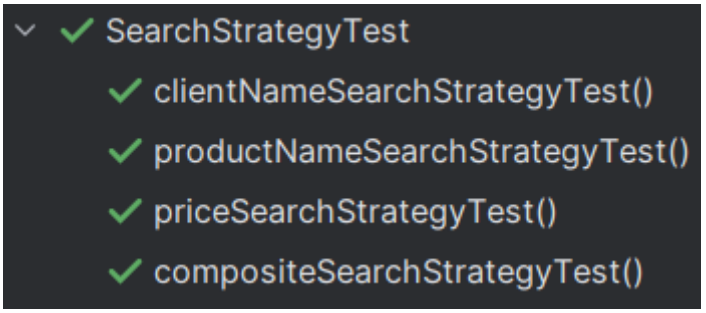
```
    @Test
    public void testCompositeSearch2() {
        // given
        ProductNameSearchStrategy productSearchStrategy = new
ProductNameSearchStrategy(productName2);
        ClientNameSearchStrategy clientSearchStrategy = new
ClientNameSearchStrategy(customerName2);
        CompositeSearchStrategy searchStrategy = new
CompositeSearchStrategy();

        searchStrategy.addStrategy(productSearchStrategy);
        searchStrategy.addStrategy(clientSearchStrategy);

        OrderHistory orderHistory = new OrderHistory();
        orderHistory.addOrder(order1);
        orderHistory.addOrder(order2);

        // when
        List<Order> actualOrders =
orderHistory.findOrderByStrategy(searchStrategy);

        // then
        assertEquals(0, actualOrders.size());
    }
}
```

f. Wszystkie testy przeszły pomyślnie:

- dla klasy OrderHistory:



- dla klasy różnych strategii filtrowania:



- całość testów wygląda następująco: