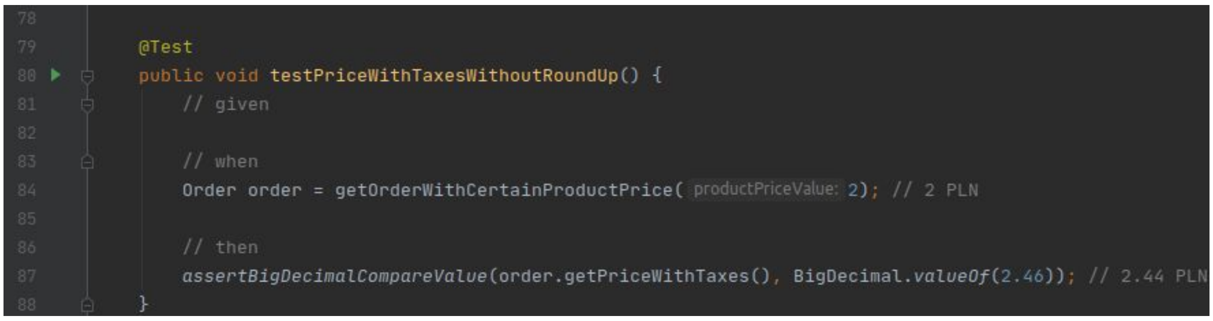
Laboratorium 5

Testy jednostkowe

Krzysztof Nalepa
Jakub Solecki

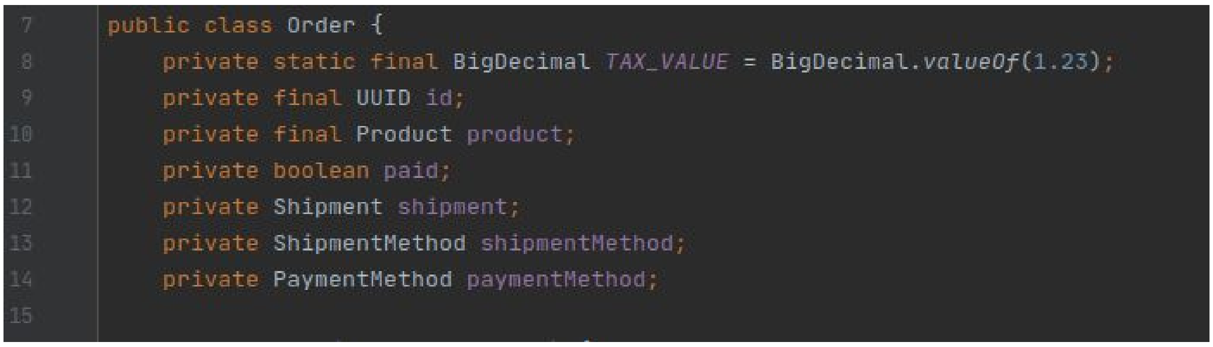
Zadanie 1

Po przeanalizowaniu testów oraz treści zadania doszliśmy do wniosku, że jedyny test który wymaga korekty to test dla wartości produktu 2. Zmieniliśmy oczekiwany wynik tak aby wartość była liczona dla podatku 23%



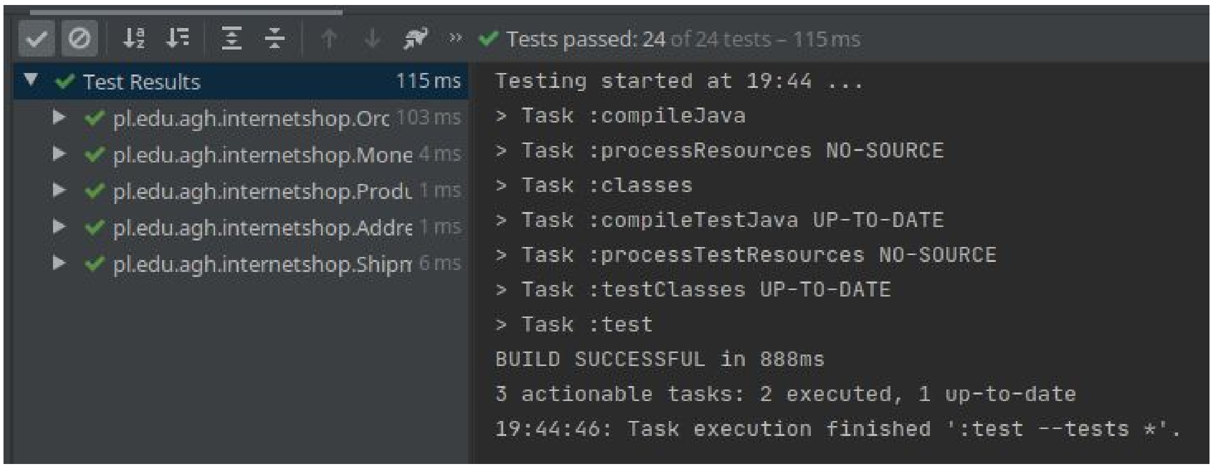
```java
    @Test
    public void testPriceWithTaxesWithoutRoundUp() {
        // given

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

        // then
        assertBigDecimalCompareValue(order.getPriceWithTaxes(), BigDecimal.valueOf(2.46)); // 2.44 PLN
    }
```

Resztę testów pozostawiliśmy bez zmian gdyż zaokrąglenia przy tak małych wartościach nie wpływają na wynik.
Następnie ustawiliśmy nową wartość podatku. Od teraz ma on wartość 23%



```java
public class Order {
    private static final BigDecimal TAX_VALUE = BigDecimal.valueOf(1.23);
    private final UUID id;
    private final Product product;
    private boolean paid;
    private Shipment shipment;
    private ShipmentMethod shipmentMethod;
    private PaymentMethod paymentMethod;
```

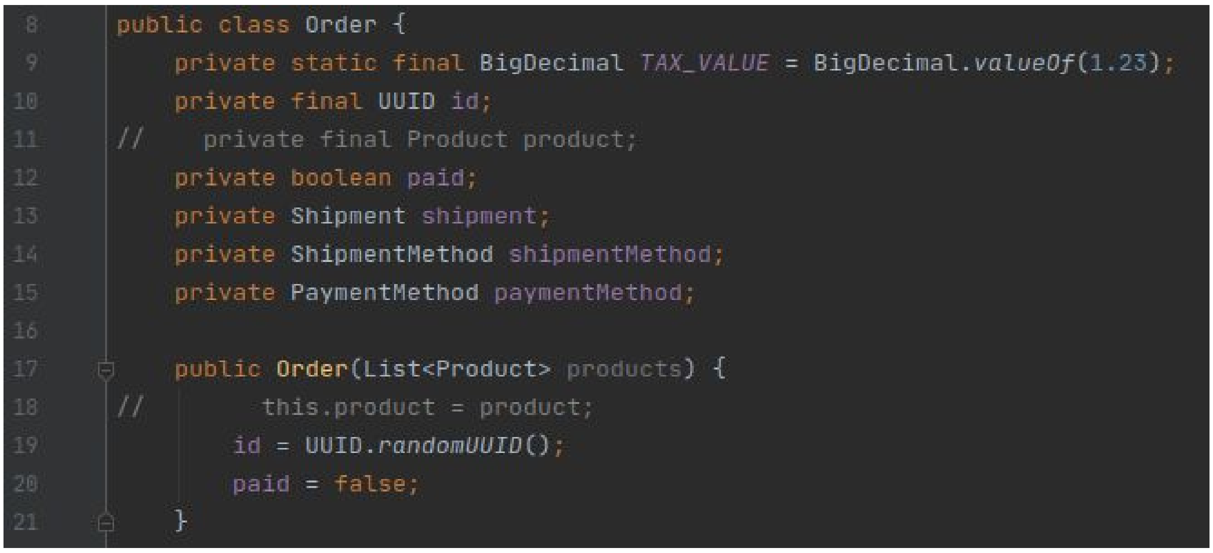
I uruchomiliśmy testy



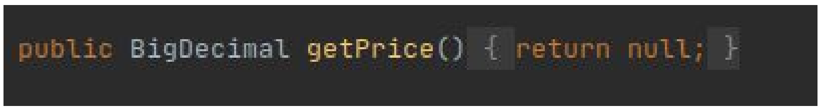
```
✓ Tests passed: 24 of 24 tests – 115 ms

▼ ✓ Test Results                     115 ms
  ▶ ✓ pl.edu.agh.internetshop.Orc    103 ms
  ▶ ✓ pl.edu.agh.internetshop.Mone     4 ms
  ▶ ✓ pl.edu.agh.internetshop.Produ    1 ms
  ▶ ✓ pl.edu.agh.internetshop.Addre    1 ms
  ▶ ✓ pl.edu.agh.internetshop.Shipn    6 ms

Testing started at 19:44 ...
> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :compileTestJava UP-TO-DATE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
> Task :test
BUILD SUCCESSFUL in 888ms
3 actionable tasks: 2 executed, 1 up-to-date
19:44:46: Task execution finished ':test --tests *'.
```

Zadanie 2

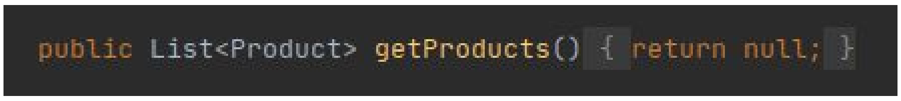
Zaczęliśmy od zmodyfikowania klasy Order w taki sposób aby możliwe było przechowywanie listy produktów. Wprowadzone modyfikacje nie obejmują jeszcze implementacji metod. Ich celem jest poprawna kompilacja z testami.



```java
public class Order {
    private static final BigDecimal TAX_VALUE = BigDecimal.valueOf(1.23);
    private final UUID id;
//    private final Product product;
    private boolean paid;
    private Shipment shipment;
    private ShipmentMethod shipmentMethod;
    private PaymentMethod paymentMethod;

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

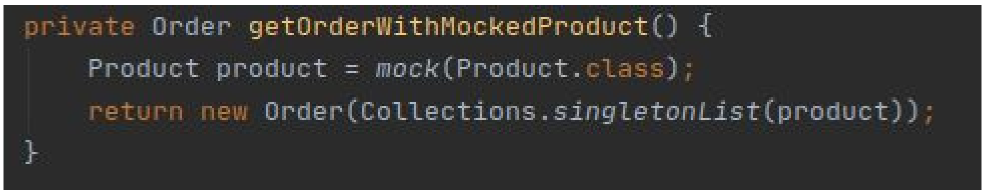


```java
public BigDecimal getPrice() { return null; }
```

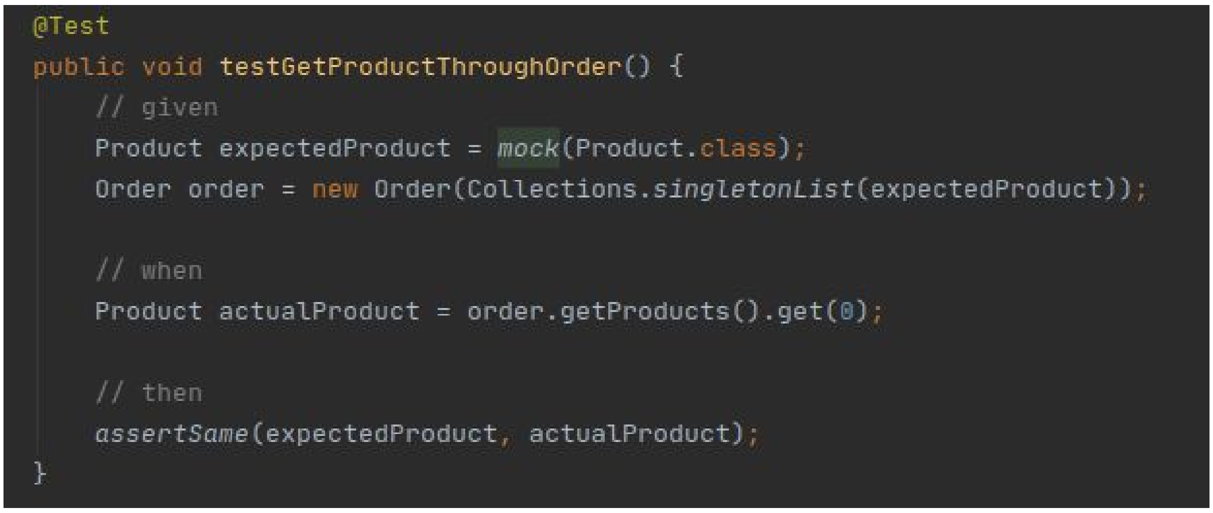


```java
public List<Product> getProducts() { return null; }
```

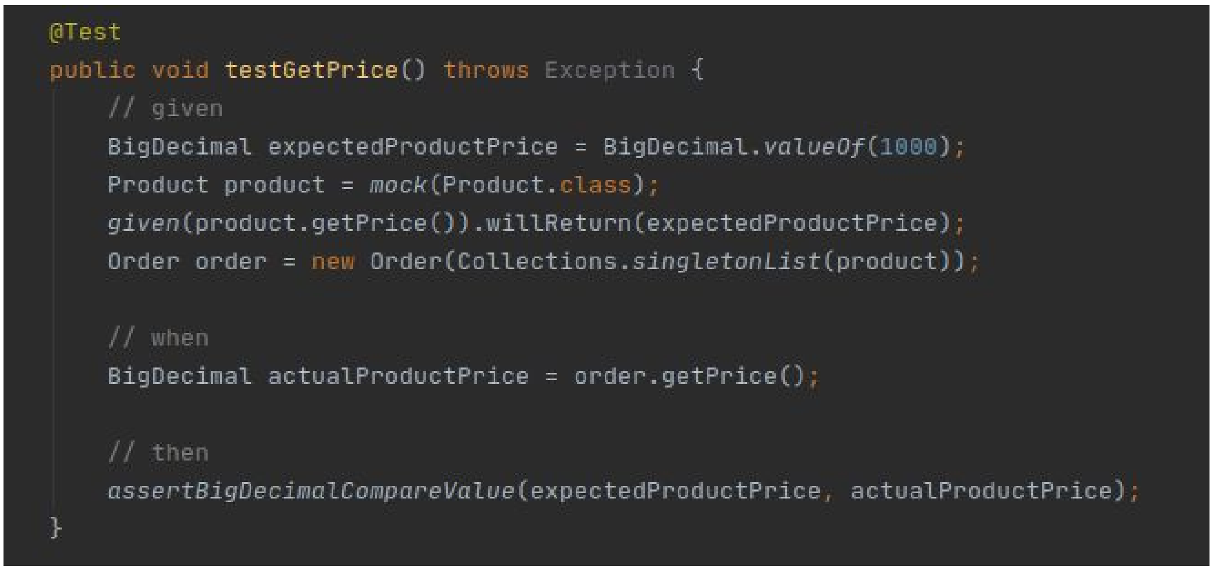
Następnie zmodyfikowaliśmy testy.



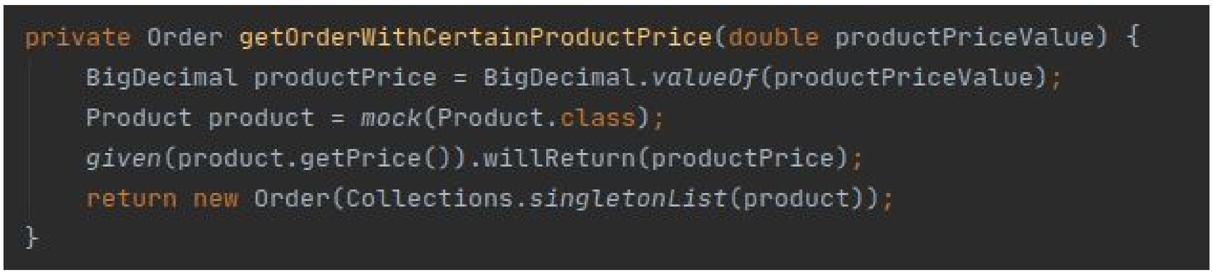
```java
private Order getOrderWithMockedProduct() {
    Product product = mock(Product.class);
    return new Order(Collections.singletonList(product));
}
```



```java
@Test
public void testGetProductThroughOrder() {
    // given
    Product expectedProduct = mock(Product.class);
    Order order = new Order(Collections.singletonList(expectedProduct));

    // when
    Product actualProduct = order.getProducts().get(0);

    // then
    assertSame(expectedProduct, actualProduct);
}
```

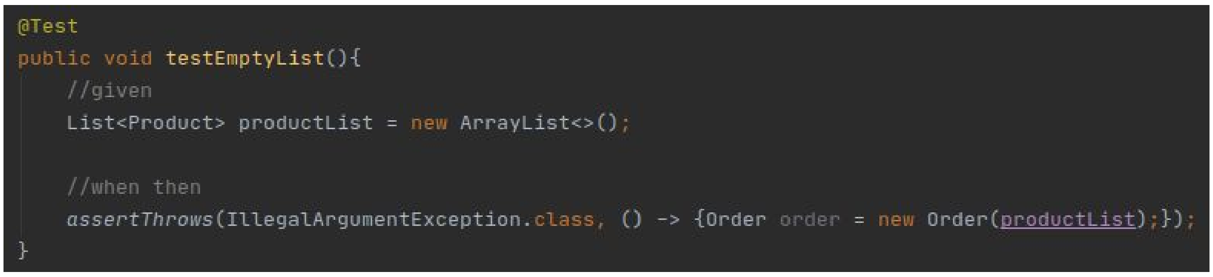


```java
@Test
public void testGetPrice() throws Exception {
    // given
    BigDecimal expectedProductPrice = BigDecimal.valueOf(1000);
    Product product = mock(Product.class);
    given(product.getPrice()).willReturn(expectedProductPrice);
    Order order = new Order(Collections.singletonList(product));

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

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



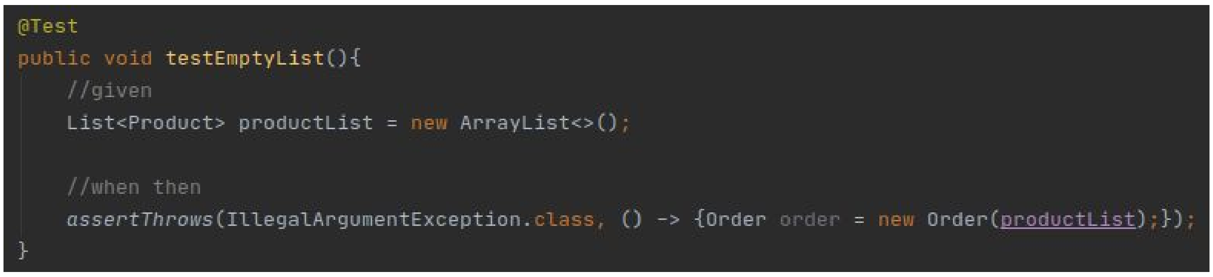
```java
private Order getOrderWithCertainProductPrice(double productPriceValue) {
    BigDecimal productPrice = BigDecimal.valueOf(productPriceValue);
    Product product = mock(Product.class);
    given(product.getPrice()).willReturn(productPrice);
    return new Order(Collections.singletonList(product));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

Dopisaliśmy również test sprawdzający działanie tworzenia zamówienia z pustą listą produktów



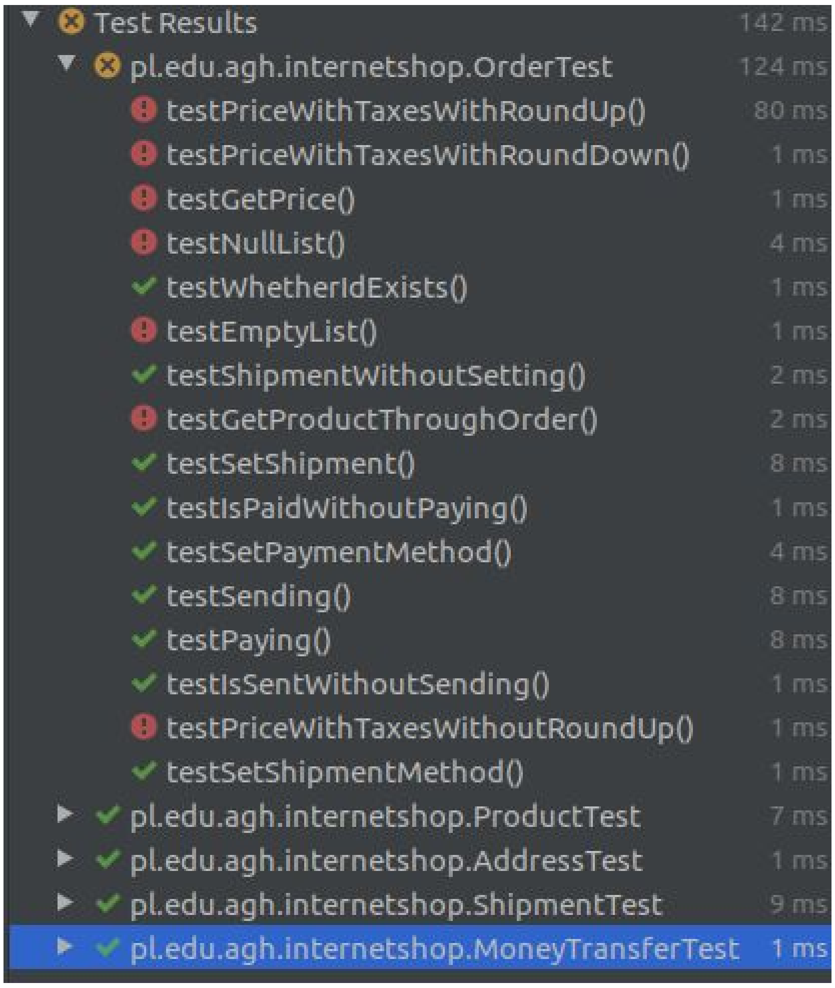
```java
@Test
public void testEmptyList(){
    //given
    List<Product> productList = new ArrayList<>();

    //when then
    assertThrows(IllegalArgumentException.class, () -> {Order order = new Order(productList);});
}
```

Oraz test który sprawdza działanie tworzenia zamówienia gdy lista nie istnieje (jest nullem)



```java
@Test
public void testEmptyList(){
    //given
    List<Product> productList = new ArrayList<>();

    //when then
    assertThrows(IllegalArgumentException.class, () -> {Order order = new Order(productList);});
}
```

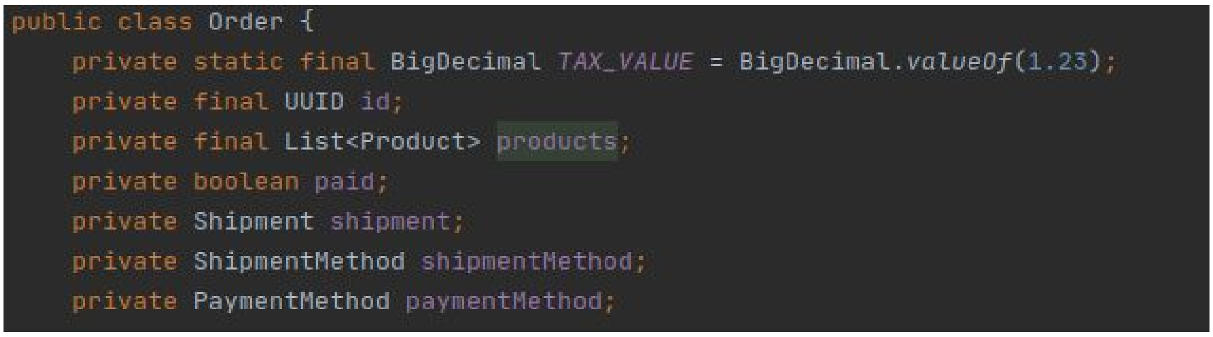
Następnie uruchomiliśmy testy



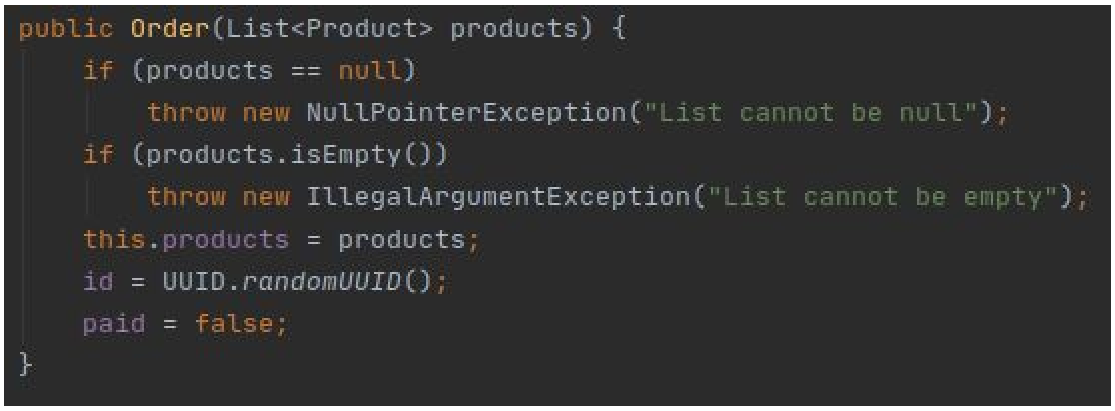
```
▼ ⊗ Test Results                                   142 ms
  ▼ ⊗ pl.edu.agh.internetshop.OrderTest            124 ms
      ! testPriceWithTaxesWithRoundUp()             80 ms
      ! testPriceWithTaxesWithRoundDown()            1 ms
      ! testGetPrice()                               1 ms
      ! testNullList()                               4 ms
      ✓ testWhetherIdExists()                        1 ms
      ! testEmptyList()                              1 ms
      ✓ testShipmentWithoutSetting()                 2 ms
      ! testGetProductThroughOrder()                 2 ms
      ✓ testSetShipment()                            8 ms
      ✓ testIsPaidWithoutPaying()                    1 ms
      ✓ testSetPaymentMethod()                       4 ms
      ✓ testSending()                                8 ms
      ✓ testPaying()                                 8 ms
      ✓ testIsSentWithoutSending()                   1 ms
      ! testPriceWithTaxesWithoutRoundUp()           1 ms
      ✓ testSetShipmentMethod()                      1 ms
  ▶ ✓ pl.edu.agh.internetshop.ProductTest            7 ms
  ▶ ✓ pl.edu.agh.internetshop.AddressTest            1 ms
  ▶ ✓ pl.edu.agh.internetshop.ShipmentTest           9 ms
  ▶ ✓ pl.edu.agh.internetshop.MoneyTransferTest      1 ms
```

Część z nich zgodnie z oczekiwaniami nie przeszła.

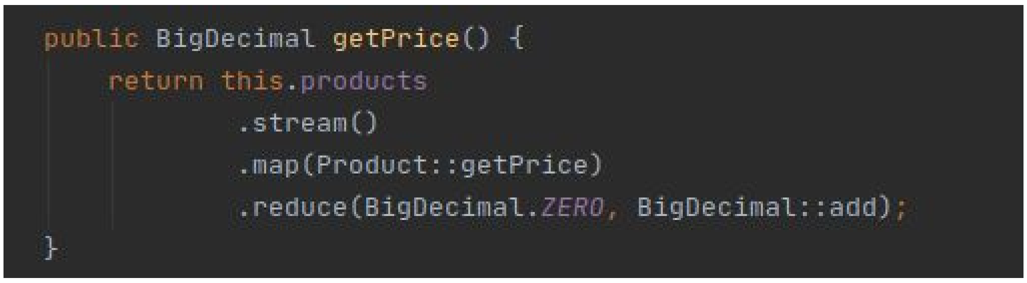
Następnie przeszliśmy do zmodyfikowania klasy Order tak aby przechodziła testy. Zmieniliśmy atrybut Product product na List<Product> products, zmodyfikowaliśmy konstruktor oraz metodę getPrice tak, aby obliczała cenę zamówienia jako sumę cen produktów.



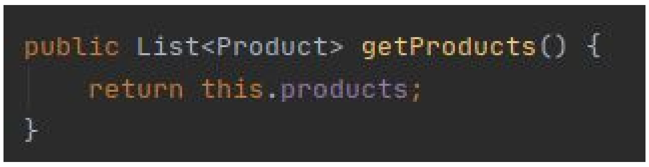
```java
public class Order {
    private static final BigDecimal TAX_VALUE = BigDecimal.valueOf(1.23);
    private final UUID id;
    private final List<Product> products;
    private boolean paid;
    private Shipment shipment;
    private ShipmentMethod shipmentMethod;
    private PaymentMethod paymentMethod;
```



```java
public Order(List<Product> products) {
    if (products == null)
        throw new NullPointerException("List cannot be null");
    if (products.isEmpty())
        throw new IllegalArgumentException("List cannot be empty");
    this.products = products;
    id = UUID.randomUUID();
    paid = false;
}
```

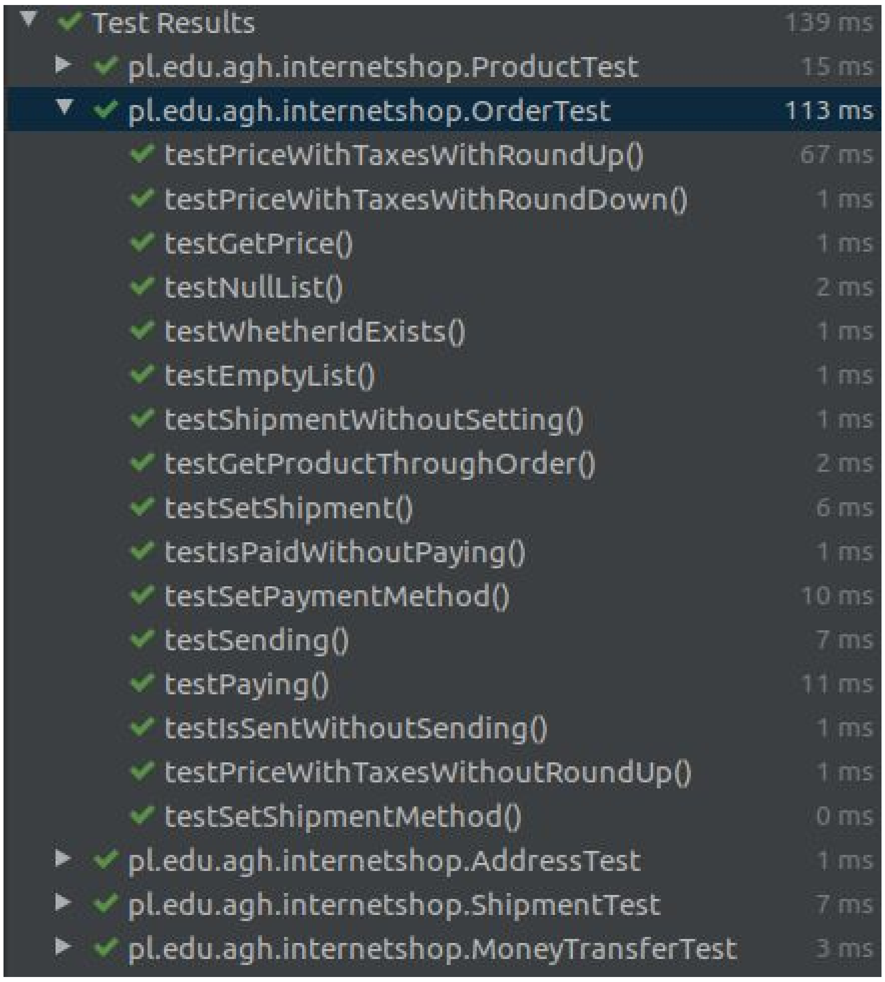


```java
public BigDecimal getPrice() {
    return this.products
            .stream()
            .map(Product::getPrice)
            .reduce(BigDecimal.ZERO, BigDecimal::add);
}
```



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

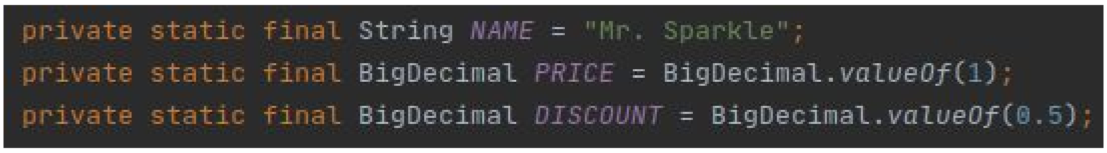
Następnie uruchomiliśmy testy. Wszystkie wykonały się pomyślnie.



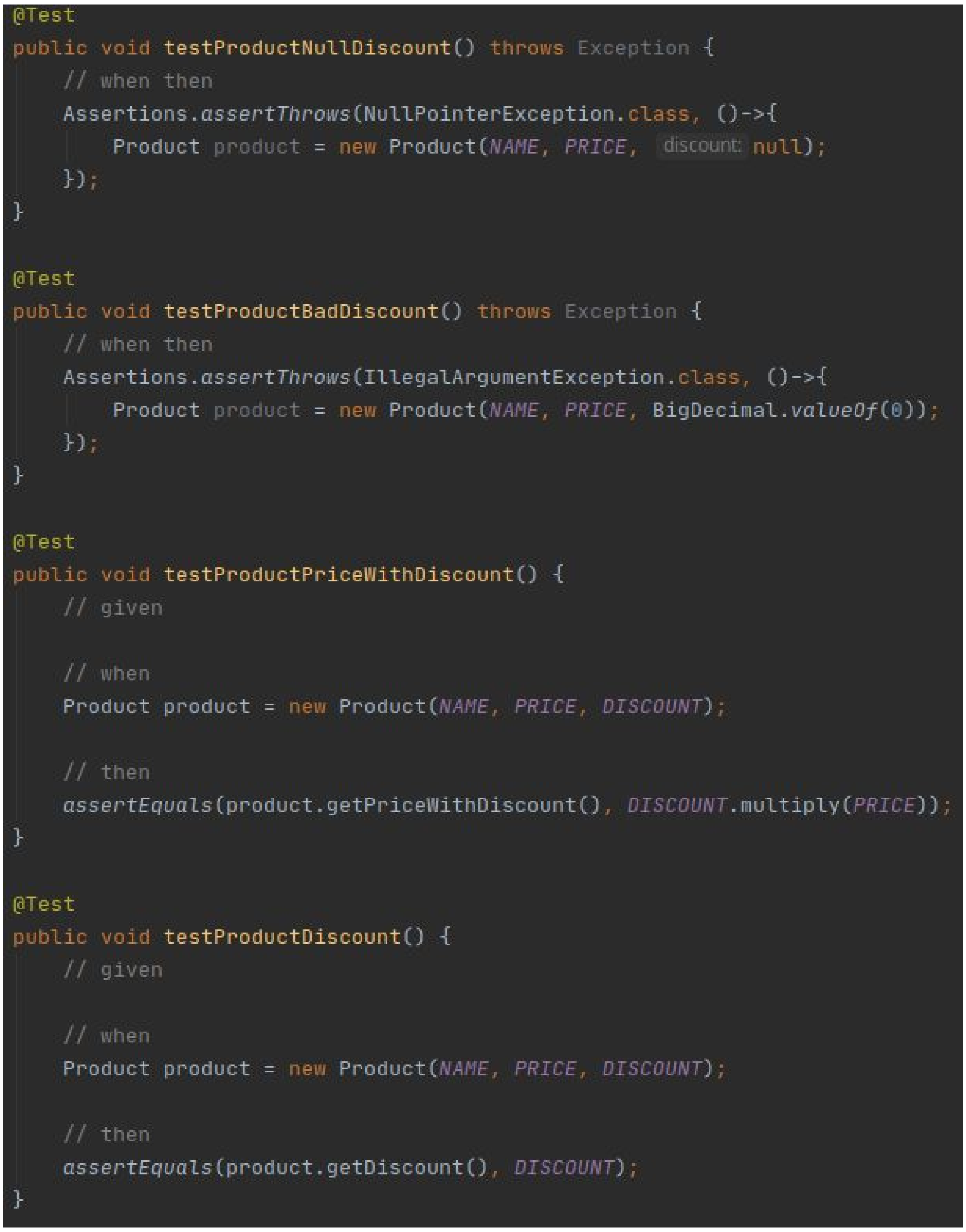
```
▼ ✓ Test Results                                        139 ms
  ▶ ✓ pl.edu.agh.internetshop.ProductTest                15 ms
  ▼ ✓ pl.edu.agh.internetshop.OrderTest                 113 ms
      ✓ testPriceWithTaxesWithRoundUp()                  67 ms
      ✓ testPriceWithTaxesWithRoundDown()                 1 ms
      ✓ testGetPrice()                                    1 ms
      ✓ testNullList()                                    2 ms
      ✓ testWhetherIdExists()                             1 ms
      ✓ testEmptyList()                                   1 ms
      ✓ testShipmentWithoutSetting()                      1 ms
      ✓ testGetProductThroughOrder()                      2 ms
      ✓ testSetShipment()                                 6 ms
      ✓ testIsPaidWithoutPaying()                         1 ms
      ✓ testSetPaymentMethod()                           10 ms
      ✓ testSending()                                     7 ms
      ✓ testPaying()                                     11 ms
      ✓ testIsSentWithoutSending()                        1 ms
      ✓ testPriceWithTaxesWithoutRoundUp()                1 ms
      ✓ testSetShipmentMethod()                           0 ms
  ▶ ✓ pl.edu.agh.internetshop.AddressTest                 1 ms
  ▶ ✓ pl.edu.agh.internetshop.ShipmentTest                7 ms
  ▶ ✓ pl.edu.agh.internetshop.MoneyTransferTest           3 ms
```

Zadanie 3

Rozpoczęliśmy od napisania wymaganych testów dla klasy Product.

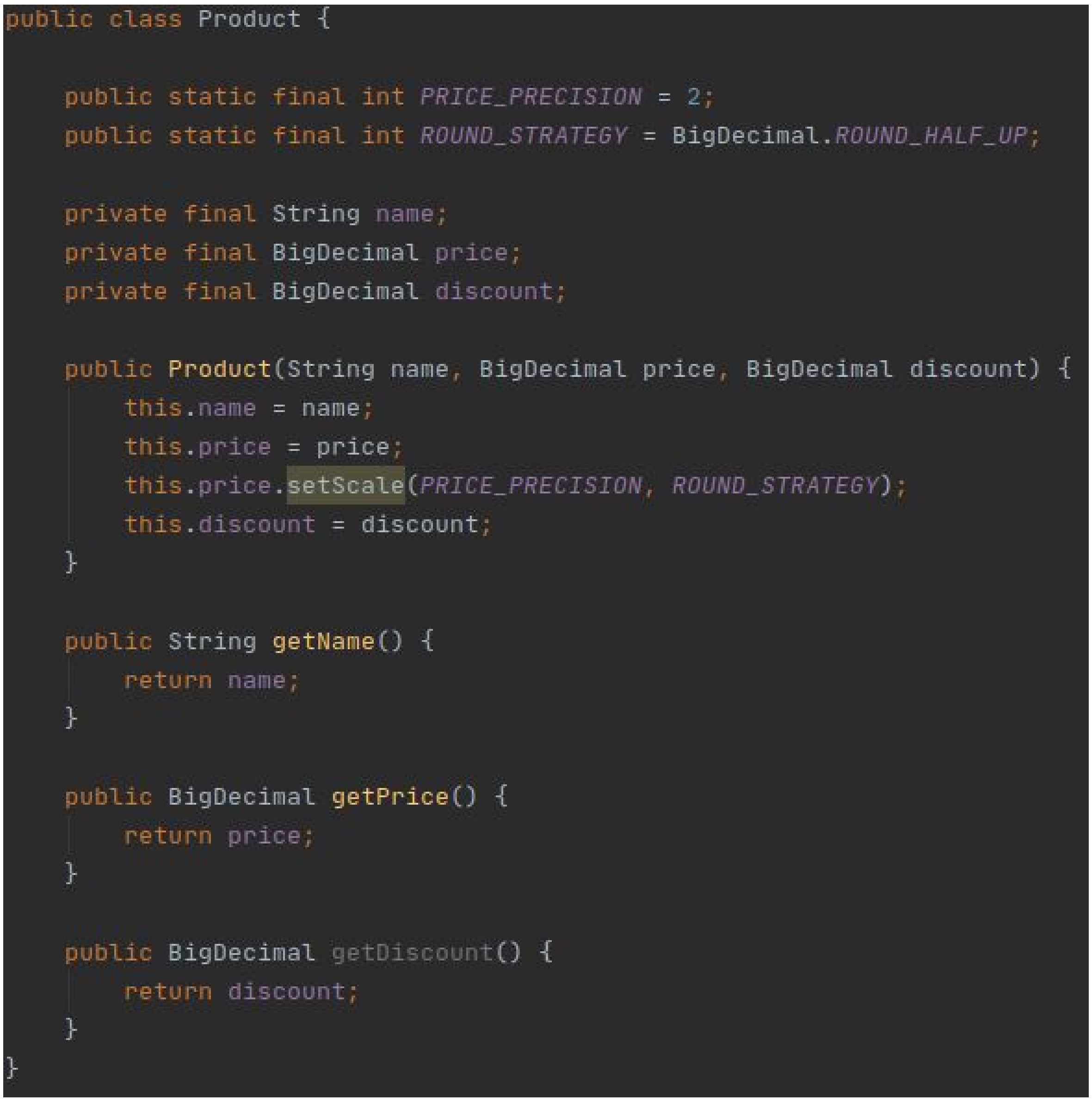


```java
private static final String NAME = "Mr. Sparkle";
private static final BigDecimal PRICE = BigDecimal.valueOf(1);
private static final BigDecimal DISCOUNT = BigDecimal.valueOf(0.5);
```



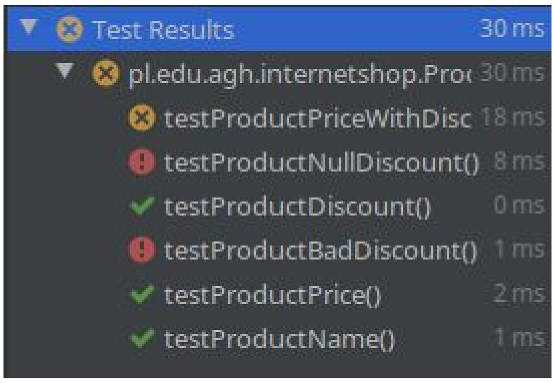
```java
@Test
public void testProductNullDiscount() throws Exception {
    // when then
    Assertions.assertThrows(NullPointerException.class, ()->{
        Product product = new Product(NAME, PRICE,  discount: null);
    });
}

@Test
public void testProductBadDiscount() throws Exception {
    // when then
    Assertions.assertThrows(IllegalArgumentException.class, ()->{
        Product product = new Product(NAME, PRICE, BigDecimal.valueOf(0));
    });
}

@Test
public void testProductPriceWithDiscount() {
    // given

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

    // then
    assertEquals(product.getPriceWithDiscount(), DISCOUNT.multiply(PRICE));
}

@Test
public void testProductDiscount() {
    // given

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

    // then
    assertEquals(product.getDiscount(), DISCOUNT);
}
```

Następnie wprowadziliśmy jedynie niezbędne zmiany w klasie Product w celu poprawnej kompilacji testów. Dodaliśmy atrybut discount, metodę getDiscount oraz rozszerzyliśmy konstruktor o parametr discount.



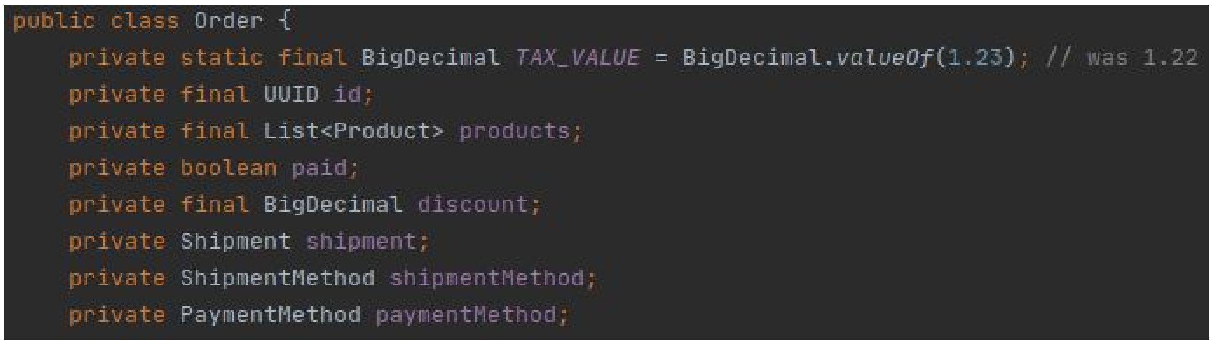
```java
public class Product {

    public static final int PRICE_PRECISION = 2;
    public static final int ROUND_STRATEGY = BigDecimal.ROUND_HALF_UP;

    private final String name;
    private final BigDecimal price;
    private final BigDecimal discount;

    public Product(String name, BigDecimal price, BigDecimal discount) {
        this.name = name;
        this.price = price;
        this.price.setScale(PRICE_PRECISION, ROUND_STRATEGY);
        this.discount = discount;
    }

    public String getName() {
        return name;
    }

    public BigDecimal getPrice() {
        return price;
    }

    public BigDecimal getDiscount() {
        return discount;
    }
}
```

Zgodnie z oczekiwaniami część testów nie przeszła.

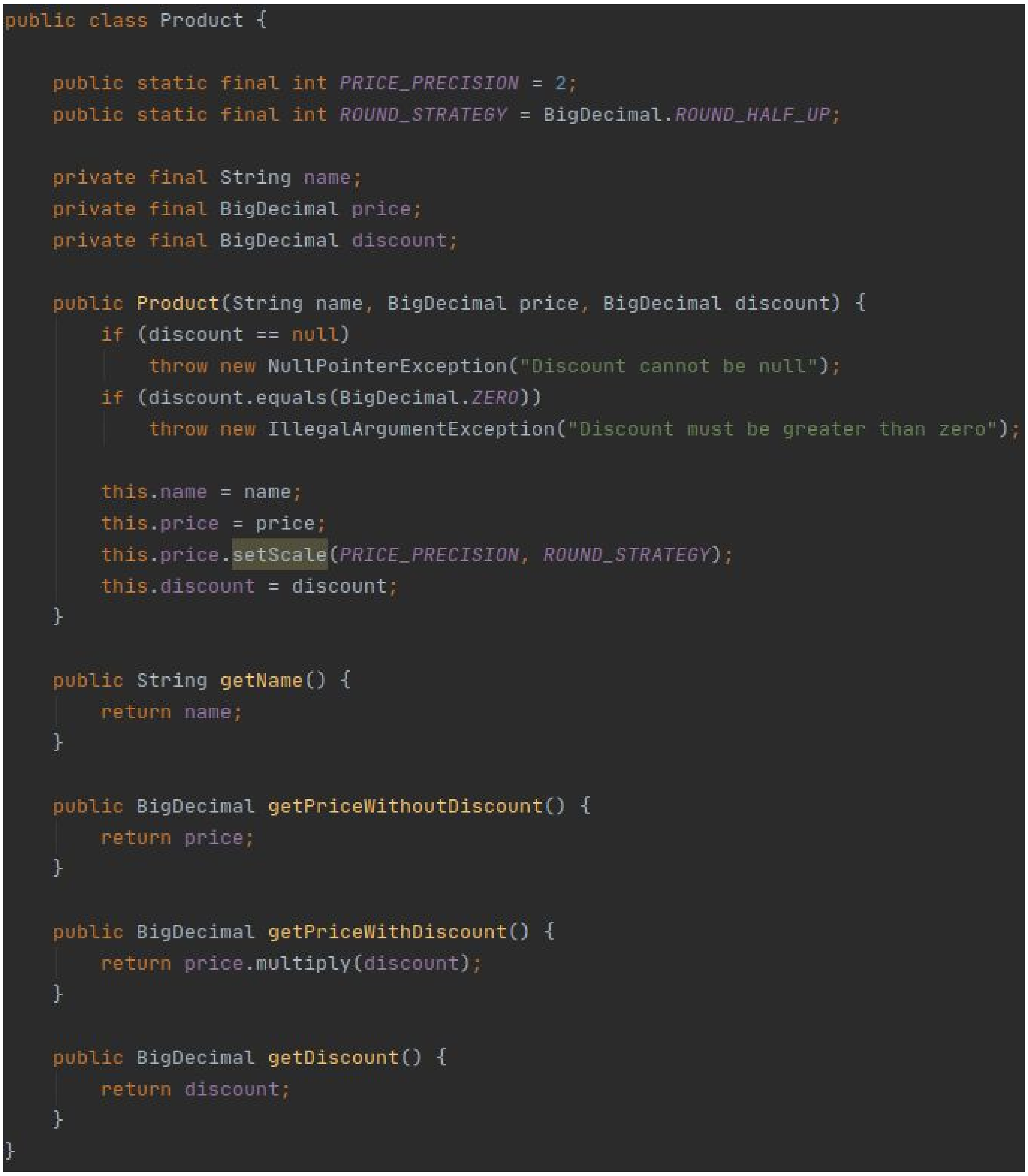


```
▼ ⊗ Test Results                              30 ms
  ▼ ⊗ pl.edu.agh.internetshop.Pro            30 ms
      ⊗ testProductPriceWithDisc             18 ms
      ! testProductNullDiscount()             8 ms
      ✓ testProductDiscount()                 0 ms
      ! testProductBadDiscount()              1 ms
      ✓ testProductPrice()                    2 ms
      ✓ testProductName()                     1 ms
```

Następnie dokonaliśmy koniecznych zmian w klasie Product aby testy przeszły.

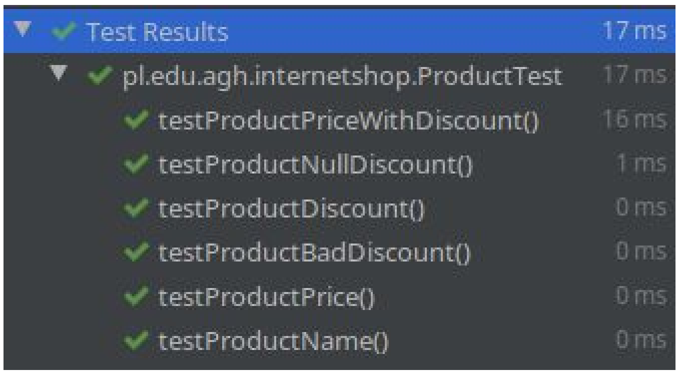


```java
public class Order {
    private static final BigDecimal TAX_VALUE = BigDecimal.valueOf(1.23); // was 1.22
    private final UUID id;
    private final List<Product> products;
    private boolean paid;
    private final BigDecimal discount;
    private Shipment shipment;
    private ShipmentMethod shipmentMethod;
    private PaymentMethod paymentMethod;
```



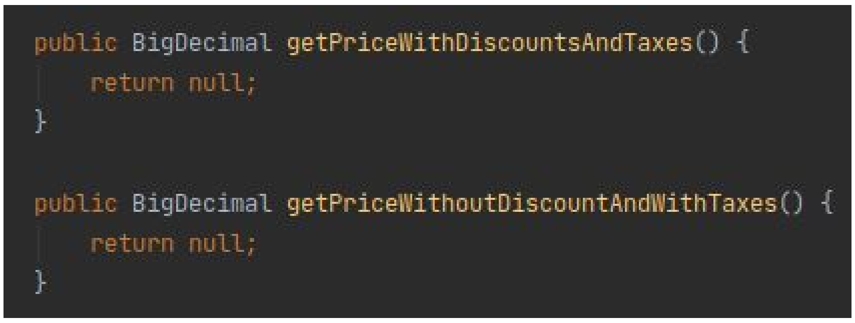
```java
public class Product {

    public static final int PRICE_PRECISION = 2;
    public static final int ROUND_STRATEGY = BigDecimal.ROUND_HALF_UP;

    private final String name;
    private final BigDecimal price;
    private final BigDecimal discount;

    public Product(String name, BigDecimal price, BigDecimal discount) {
        if (discount == null)
            throw new NullPointerException("Discount cannot be null");
        if (discount.equals(BigDecimal.ZERO))
            throw new IllegalArgumentException("Discount must be greater than zero");

        this.name = name;
        this.price = price;
        this.price.setScale(PRICE_PRECISION, ROUND_STRATEGY);
        this.discount = discount;
    }

    public String getName() {
        return name;
    }

    public BigDecimal getPriceWithoutDiscount() {
        return price;
    }

    public BigDecimal getPriceWithDiscount() {
        return price.multiply(discount);
    }

    public BigDecimal getDiscount() {
        return discount;
    }
}
```

Na końcu zweryfikowaliśmy zmiany testami.



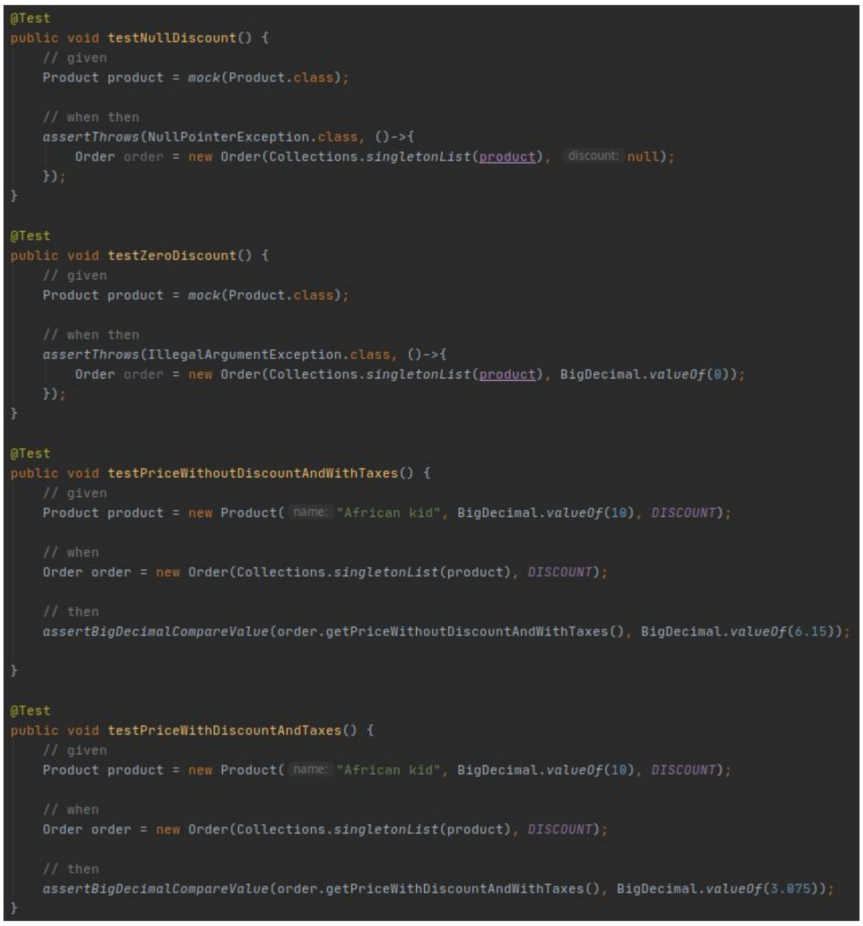
```
▼ ✓ Test Results                                   17 ms
  ▼ ✓ pl.edu.agh.internetshop.ProductTest          17 ms
      ✓ testProductPriceWithDiscount()             16 ms
      ✓ testProductNullDiscount()                   1 ms
      ✓ testProductDiscount()                       0 ms
      ✓ testProductBadDiscount()                    0 ms
      ✓ testProductPrice()                          0 ms
      ✓ testProductName()                           0 ms
```

Kolejnym krokiem było dodanie metod do klasy order



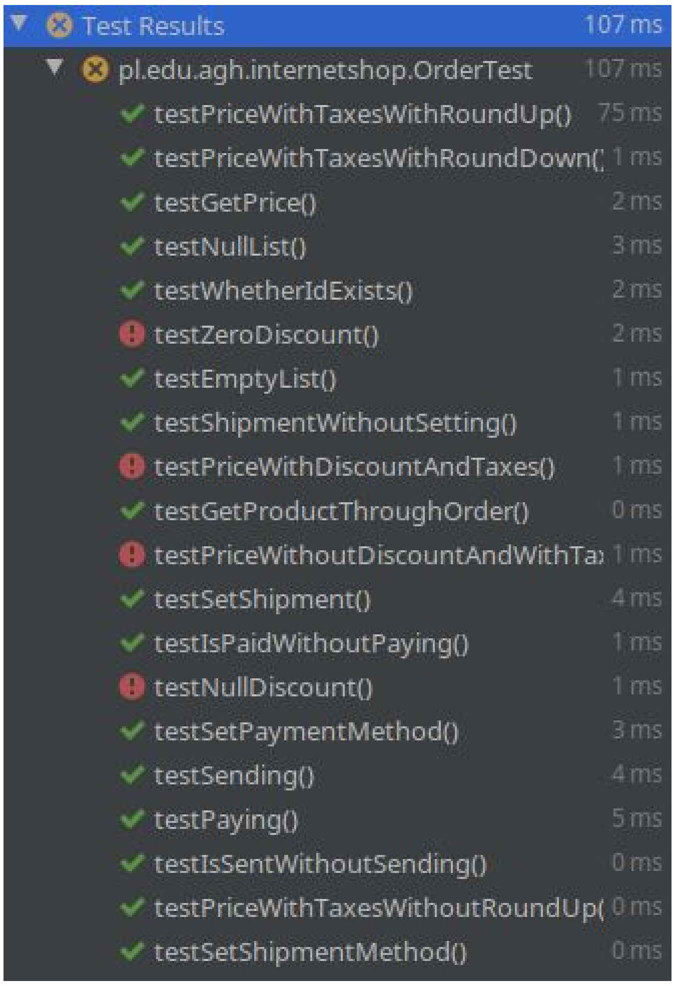
```java
public BigDecimal getPriceWithDiscountsAndTaxes() {
    return null;
}

public BigDecimal getPriceWithoutDiscountAndWithTaxes() {
    return null;
}
```

Oraz napisanie dla niej testów



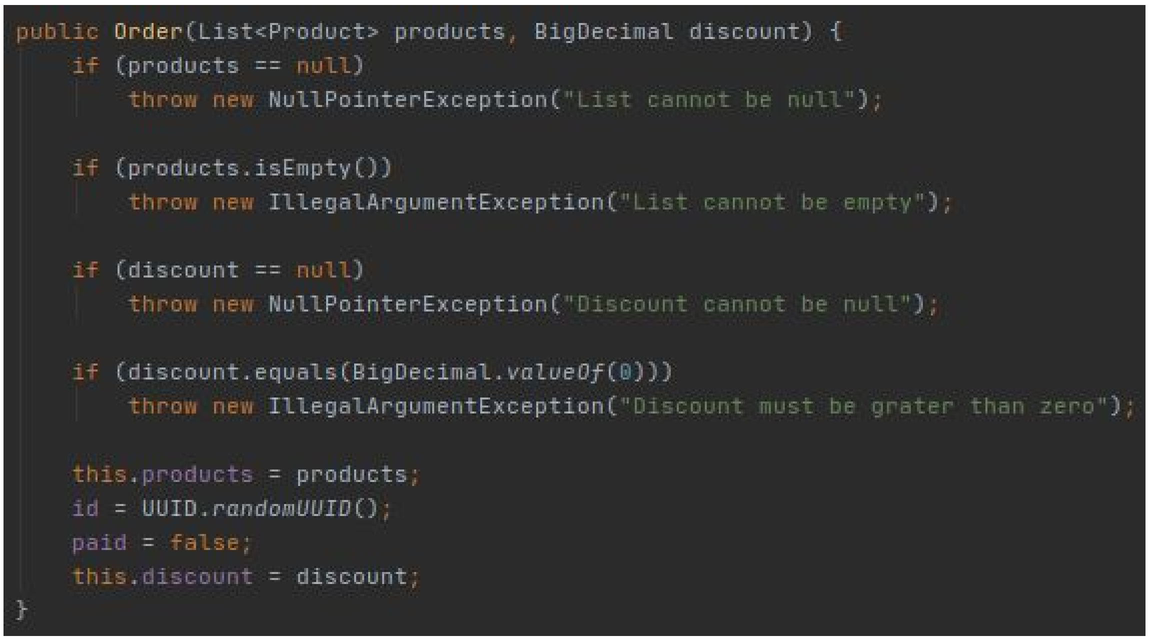
```java
@Test
public void testNullDiscount() {
    // given
    Product product = mock(Product.class);

    // when then
    assertThrows(NullPointerException.class, ()->{
        Order order = new Order(Collections.singletonList(product), discount: null);
    });
}

@Test
public void testZeroDiscount() {
    // given
    Product product = mock(Product.class);

    // when then
    assertThrows(IllegalArgumentException.class, ()->{
        Order order = new Order(Collections.singletonList(product), BigDecimal.valueOf(0));
    });
}

@Test
public void testPriceWithoutDiscountAndWithTaxes() {
    // given
    Product product = new Product( name: "African kid", BigDecimal.valueOf(10), DISCOUNT);

    // when
    Order order = new Order(Collections.singletonList(product), DISCOUNT);

    // then
    assertBigDecimalCompareValue(order.getPriceWithoutDiscountAndWithTaxes(), BigDecimal.valueOf(6.15));

}

@Test
public void testPriceWithDiscountAndTaxes() {
    // given
    Product product = new Product( name: "African kid", BigDecimal.valueOf(10), DISCOUNT);

    // when
    Order order = new Order(Collections.singletonList(product), DISCOUNT);

    // then
    assertBigDecimalCompareValue(order.getPriceWithDiscountAndWithTaxes(), BigDecimal.valueOf(3.075));
}
```

Zgodnie z oczekiwaniami część testów nie przeszła.

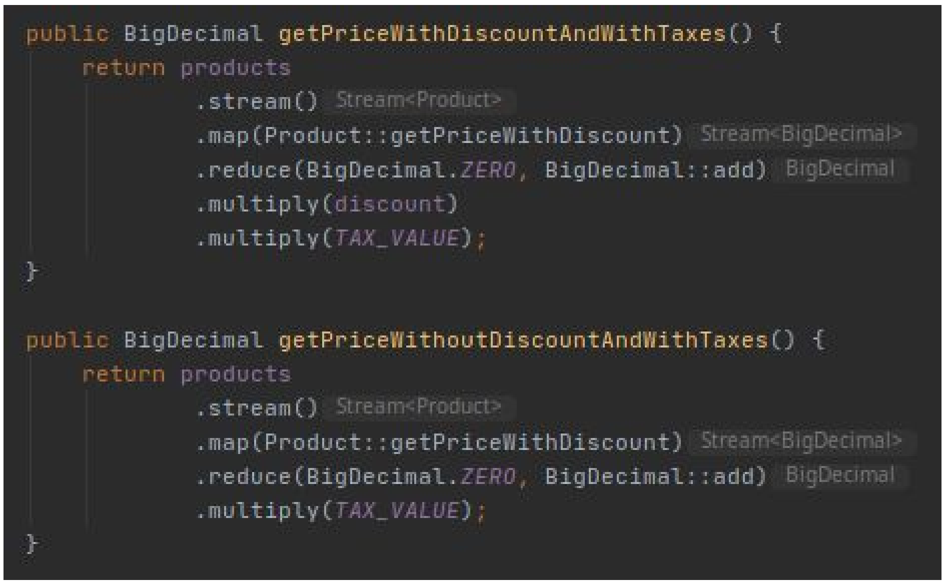


```
Test Results                                   107 ms
  pl.edu.agh.internetshop.OrderTest            107 ms
    ✓ testPriceWithTaxesWithRoundUp()          75 ms
    ✓ testPriceWithTaxesWithRoundDown(         1 ms
    ✓ testGetPrice()                           2 ms
    ✓ testNullList()                           3 ms
    ✓ testWhetherIdExists()                    2 ms
    ! testZeroDiscount()                       2 ms
    ✓ testEmptyList()                          1 ms
    ✓ testShipmentWithoutSetting()             1 ms
    ! testPriceWithDiscountAndTaxes()          1 ms
    ✓ testGetProductThroughOrder()             0 ms
    ! testPriceWithoutDiscountAndWithTa        1 ms
    ✓ testSetShipment()                        4 ms
    ✓ testIsPaidWithoutPaying()                1 ms
    ! testNullDiscount()                       1 ms
    ✓ testSetPaymentMethod()                   3 ms
    ✓ testSending()                            4 ms
    ✓ testPaying()                             5 ms
    ✓ testIsSentWithoutSending()               0 ms
    ✓ testPriceWithTaxesWithoutRoundUp(        0 ms
    ✓ testSetShipmentMethod()                  0 ms
```

Zmodyfikowaliśmy poszczególne metody tak aby spełniały wymagania testów.

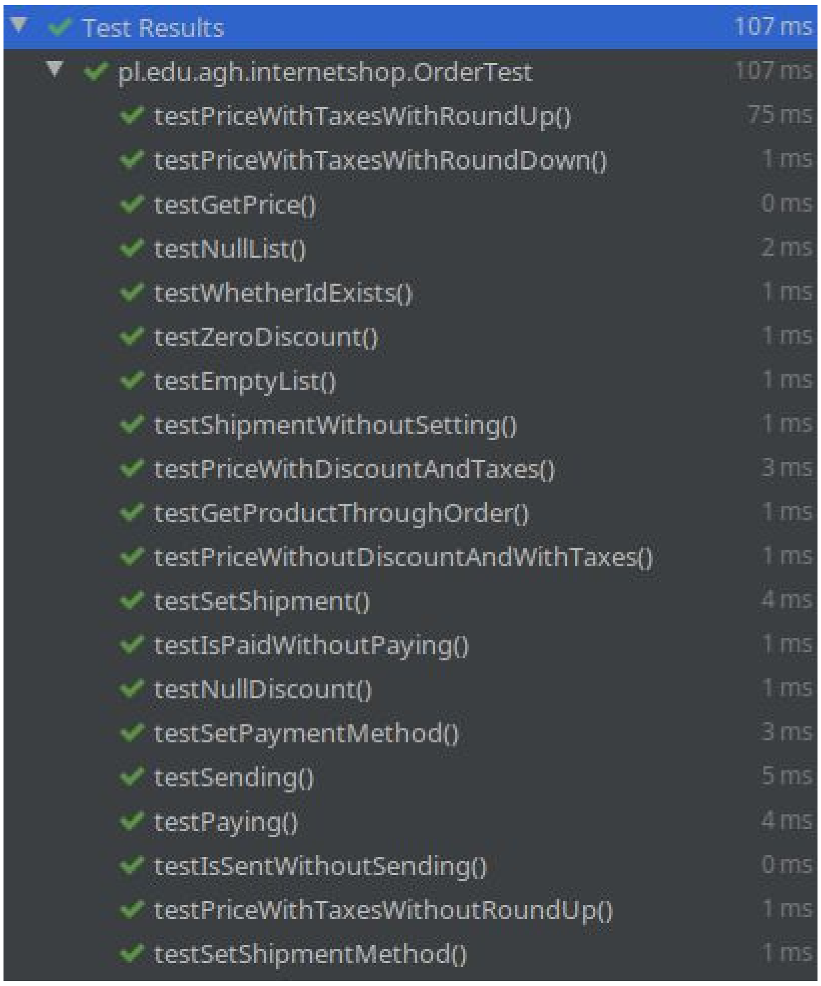


```java
public Order(List<Product> products, BigDecimal discount) {
    if (products == null)
        throw new NullPointerException("List cannot be null");

    if (products.isEmpty())
        throw new IllegalArgumentException("List cannot be empty");

    if (discount == null)
        throw new NullPointerException("Discount cannot be null");

    if (discount.equals(BigDecimal.valueOf(0)))
        throw new IllegalArgumentException("Discount must be grater than zero");

    this.products = products;
    id = UUID.randomUUID();
    paid = false;
    this.discount = discount;
}
```



```java
public BigDecimal getPriceWithDiscountAndWithTaxes() {
    return products
            .stream() Stream<Product>
            .map(Product::getPriceWithDiscount) Stream<BigDecimal>
            .reduce(BigDecimal.ZERO, BigDecimal::add) BigDecimal
            .multiply(discount)
            .multiply(TAX_VALUE);
}

public BigDecimal getPriceWithoutDiscountAndWithTaxes() {
    return products
            .stream() Stream<Product>
            .map(Product::getPriceWithDiscount) Stream<BigDecimal>
            .reduce(BigDecimal.ZERO, BigDecimal::add) BigDecimal
            .multiply(TAX_VALUE);
}
```

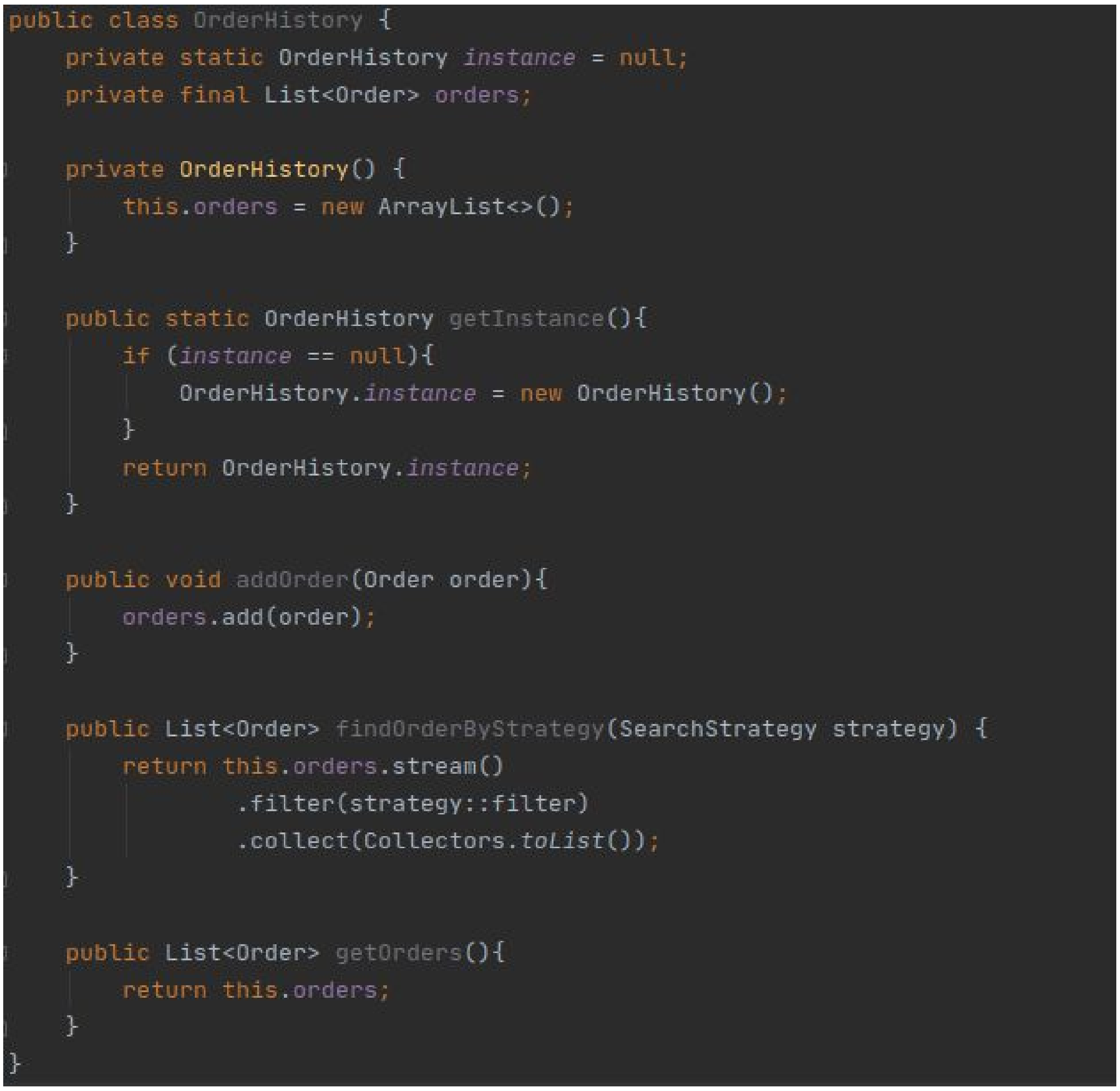
Na końcu ponownie uruchomiliśmy testy w celu weryfikacji wprowadzonych zmian.



```
▼ ✓ Test Results                                   107 ms
  ▼ ✓ pl.edu.agh.internetshop.OrderTest            107 ms
      ✓ testPriceWithTaxesWithRoundUp()             75 ms
      ✓ testPriceWithTaxesWithRoundDown()            1 ms
      ✓ testGetPrice()                               0 ms
      ✓ testNullList()                               2 ms
      ✓ testWhetherIdExists()                        1 ms
      ✓ testZeroDiscount()                           1 ms
      ✓ testEmptyList()                              1 ms
      ✓ testShipmentWithoutSetting()                 1 ms
      ✓ testPriceWithDiscountAndTaxes()              3 ms
      ✓ testGetProductThroughOrder()                 1 ms
      ✓ testPriceWithoutDiscountAndWithTaxes()       1 ms
      ✓ testSetShipment()                            4 ms
      ✓ testIsPaidWithoutPaying()                    1 ms
      ✓ testNullDiscount()                           1 ms
      ✓ testSetPaymentMethod()                       3 ms
      ✓ testSending()                                5 ms
      ✓ testPaying()                                 4 ms
      ✓ testIsSentWithoutSending()                   0 ms
      ✓ testPriceWithTaxesWithoutRoundUp()           1 ms
      ✓ testSetShipmentMethod()                      1 ms
```

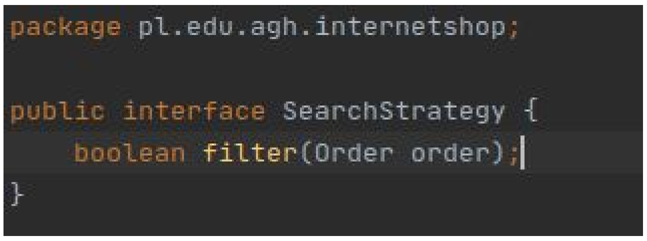
Zadanie 4

Rozpoczęliśmy od definicji klasy OrderHistory, której zadaniem jest trzymanie listy wszystkich zamówień w sklepie. W tym celu skorzystaliśmy z wzorca singleton (co wymusiło delikatnie inne podejście do testów).

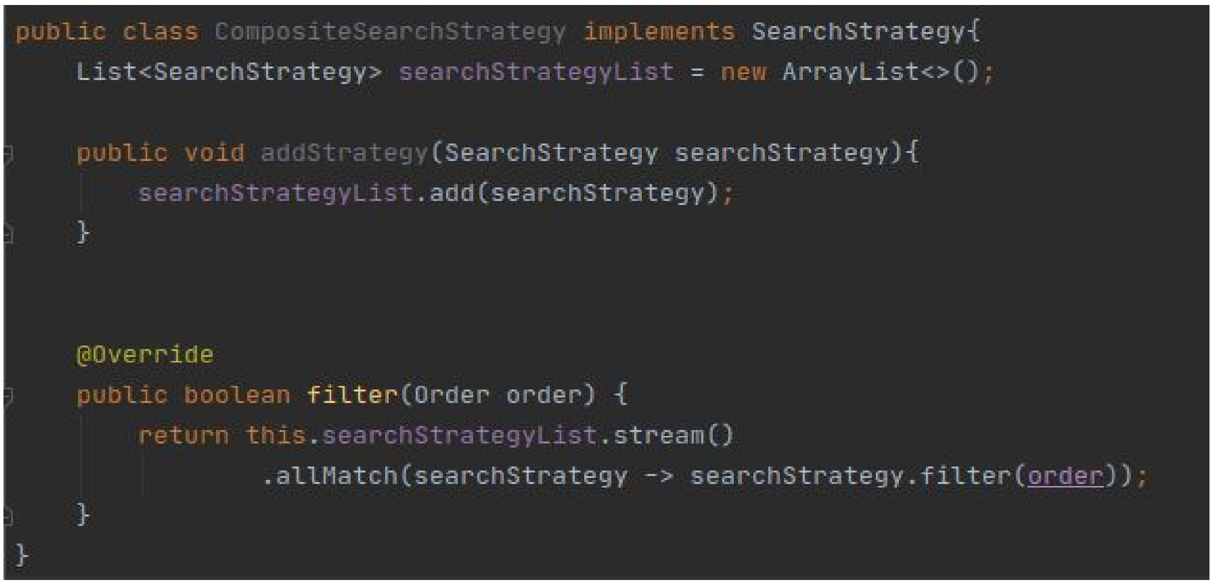


```java
public class OrderHistory {
    private static OrderHistory instance = null;
    private final List<Order> orders;

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

    public static OrderHistory getInstance(){
        if (instance == null){
            OrderHistory.instance = new OrderHistory();
        }
        return OrderHistory.instance;
    }

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

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

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

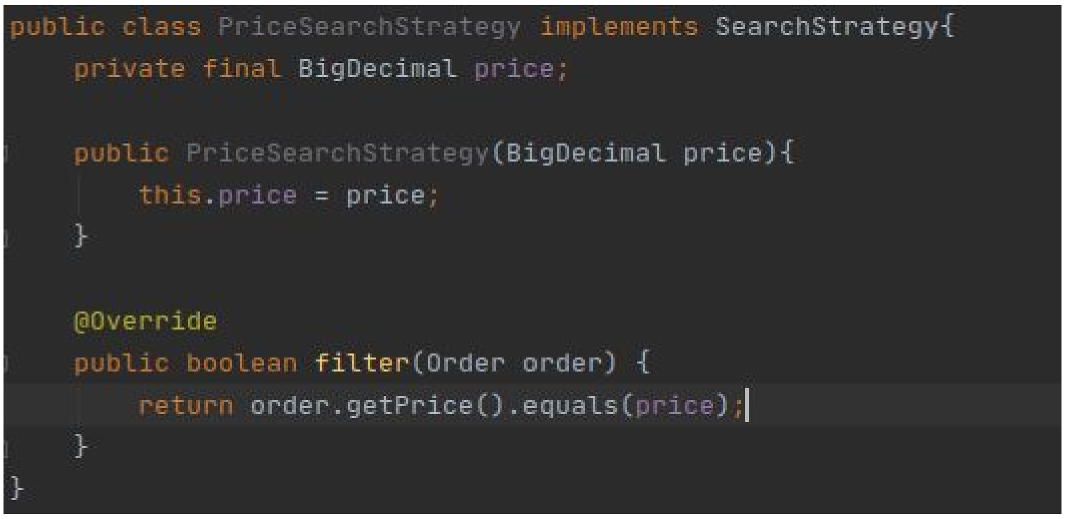
Następnie stworzyliśmy wyszukiwanie zamówień korzystając ze wzorca Composite. Obejmowało ono interfejs dla klas szukających oraz same klasy.



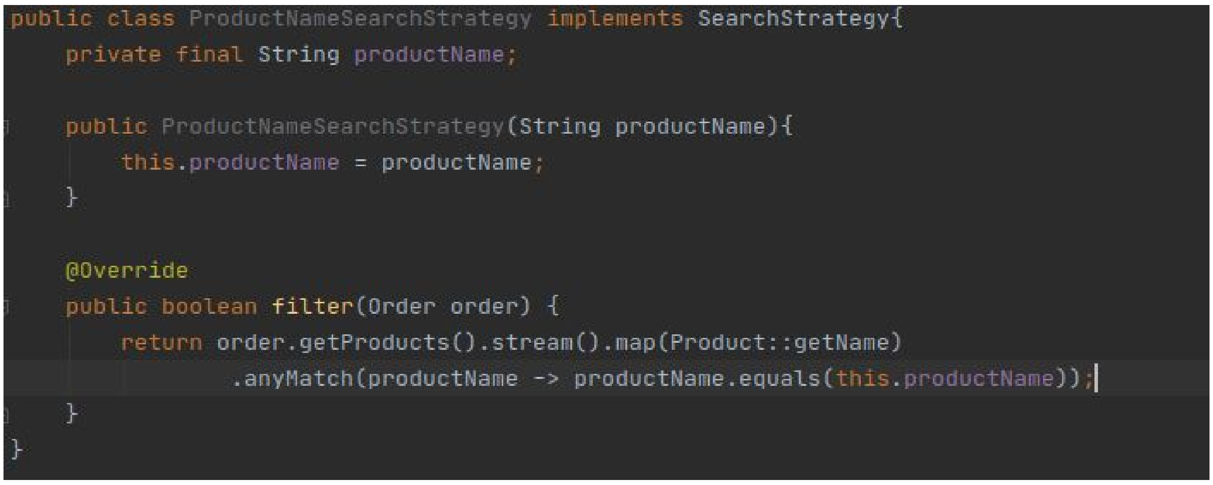
```java
package pl.edu.agh.internetshop;

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



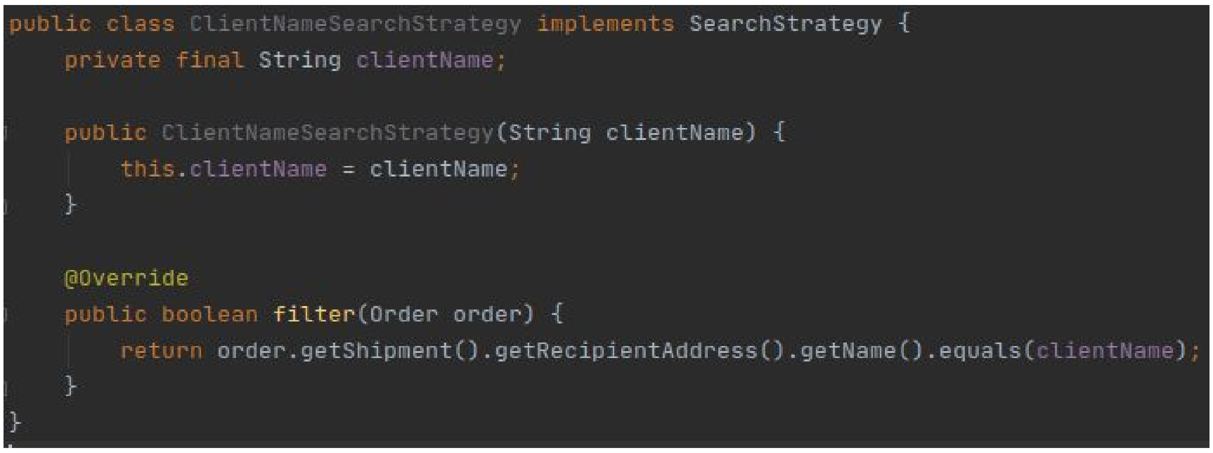
```java
public class CompositeSearchStrategy implements SearchStrategy{
    List<SearchStrategy> searchStrategyList = new ArrayList<>();

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


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



```java
public class PriceSearchStrategy implements SearchStrategy{
    private final BigDecimal price;

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

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

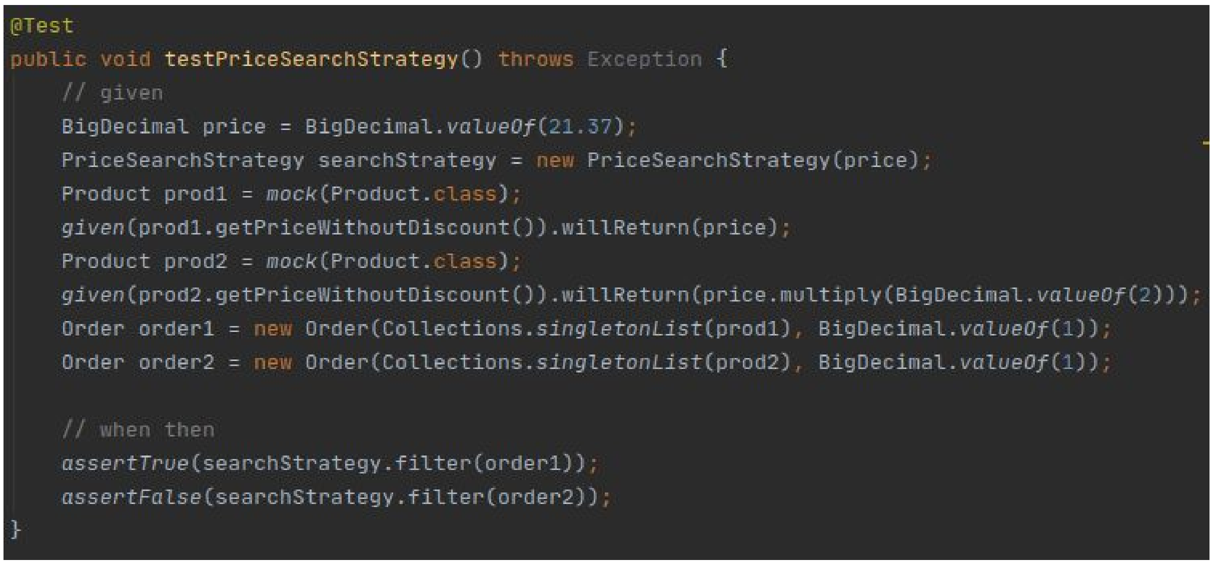


```java
public class ProductNameSearchStrategy implements SearchStrategy{
    private final String productName;

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

    @Override
    public boolean filter(Order order) {
        return order.getProducts().stream().map(Product::getName)
                .anyMatch(productName -> productName.equals(this.productName));
    }
}
```

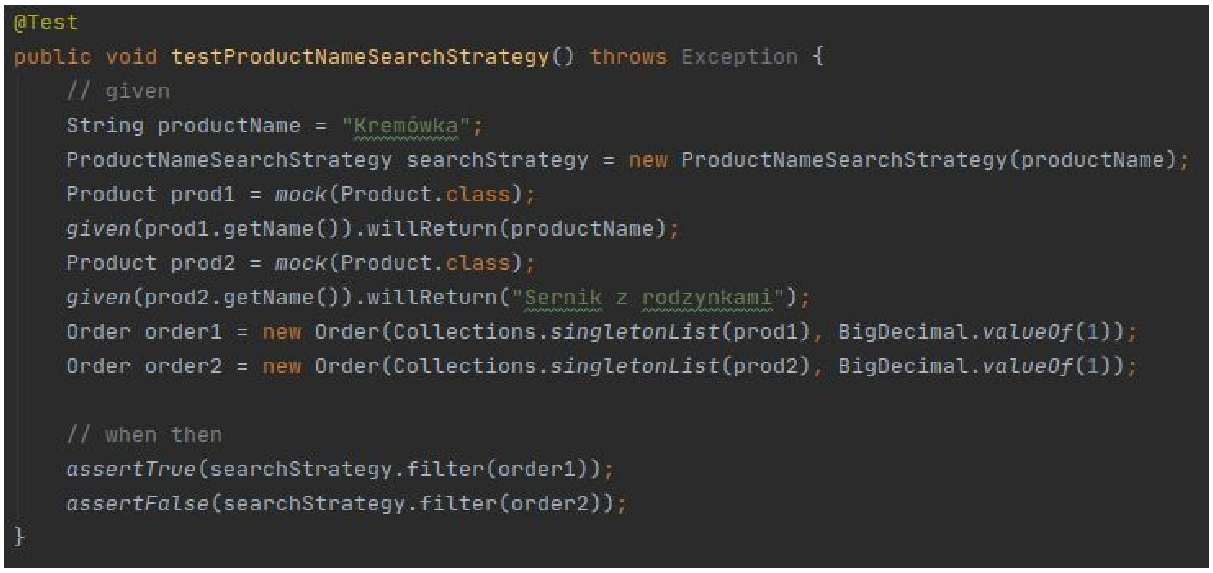


```java
public class ClientNameSearchStrategy implements SearchStrategy {
    private final String clientName;

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

    @Override
    public boolean filter(Order order) {
        return order.getShipment().getRecipientAddress().getName().equals(clientName);
    }
}
```

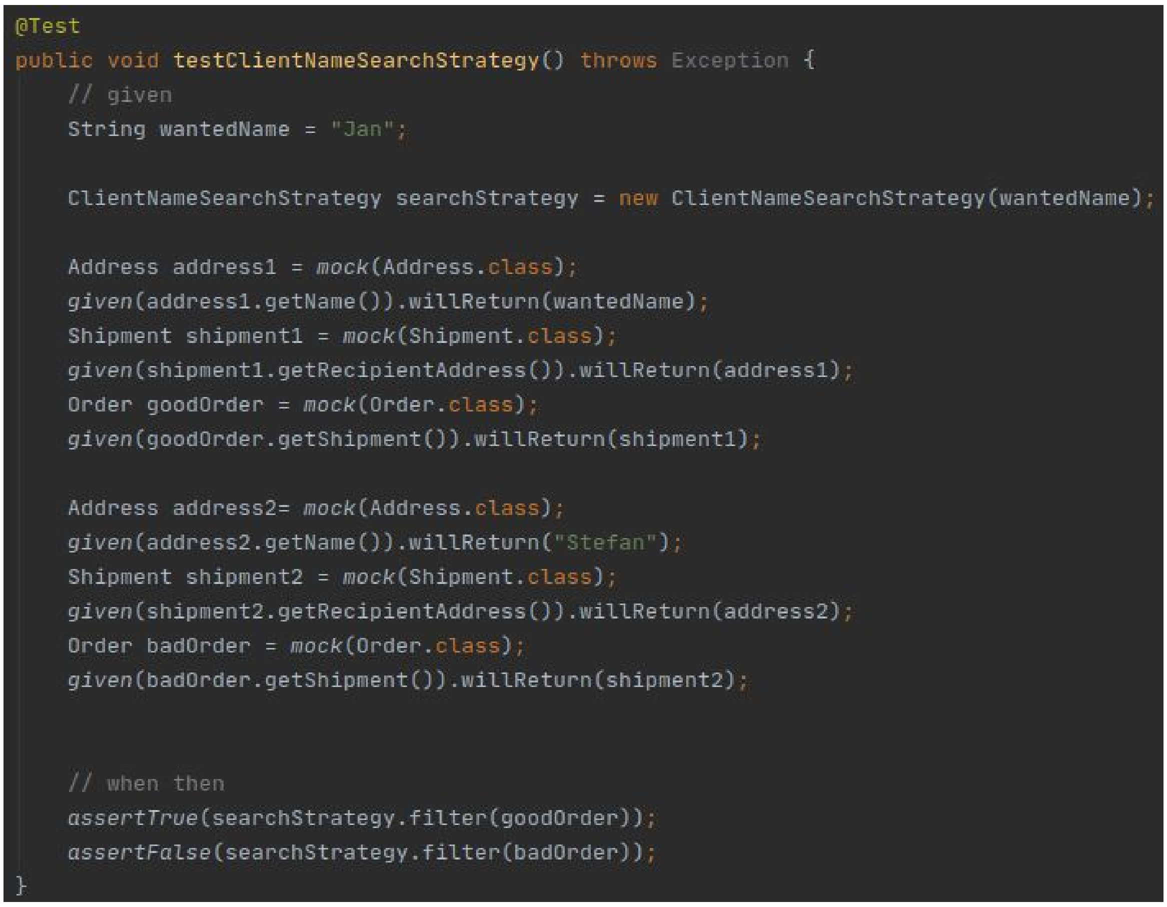
Następnie napisaliśmy testy dla klas implementujących interfejs SearchStrategy. Ze względu na specyficzne przeznaczenie tych klas ograniczyliśmy się do jednego tekstu na klasę.



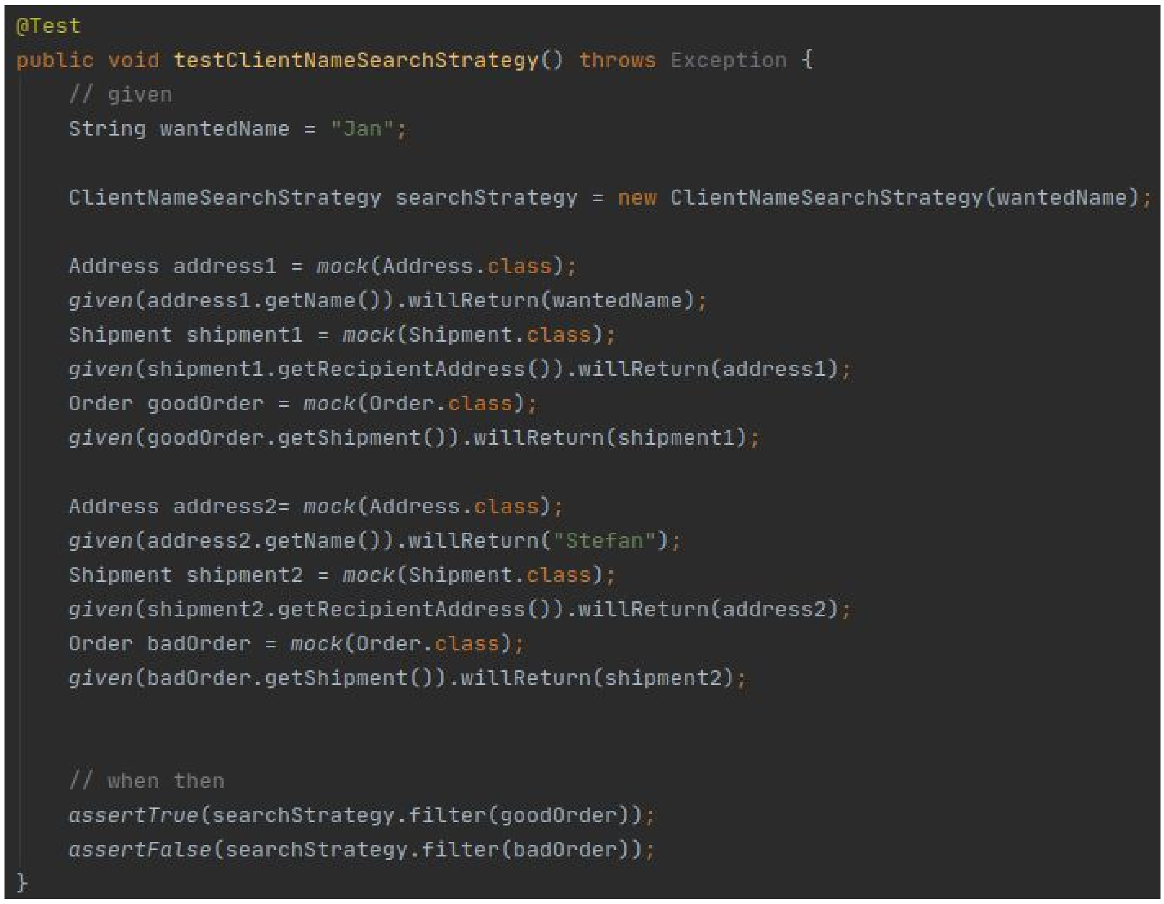
```java
@Test
public void testPriceSearchStrategy() throws Exception {
    // given
    BigDecimal price = BigDecimal.valueOf(21.37);
    PriceSearchStrategy searchStrategy = new PriceSearchStrategy(price);
    Product prod1 = mock(Product.class);
    given(prod1.getPriceWithoutDiscount()).willReturn(price);
    Product prod2 = mock(Product.class);
    given(prod2.getPriceWithoutDiscount()).willReturn(price.multiply(BigDecimal.valueOf(2)));
    Order order1 = new Order(Collections.singletonList(prod1), BigDecimal.valueOf(1));
    Order order2 = new Order(Collections.singletonList(prod2), BigDecimal.valueOf(1));

    // when then
    assertTrue(searchStrategy.filter(order1));
    assertFalse(searchStrategy.filter(order2));
}
```



```java
@Test
public void testProductNameSearchStrategy() throws Exception {
    // given
    String productName = "Kremówka";
    ProductNameSearchStrategy searchStrategy = new ProductNameSearchStrategy(productName);
    Product prod1 = mock(Product.class);
    given(prod1.getName()).willReturn(productName);
    Product prod2 = mock(Product.class);
    given(prod2.getName()).willReturn("Sernik z rodzynkami");
    Order order1 = new Order(Collections.singletonList(prod1), BigDecimal.valueOf(1));
    Order order2 = new Order(Collections.singletonList(prod2), BigDecimal.valueOf(1));

    // when then
    assertTrue(searchStrategy.filter(order1));
    assertFalse(searchStrategy.filter(order2));
}
```

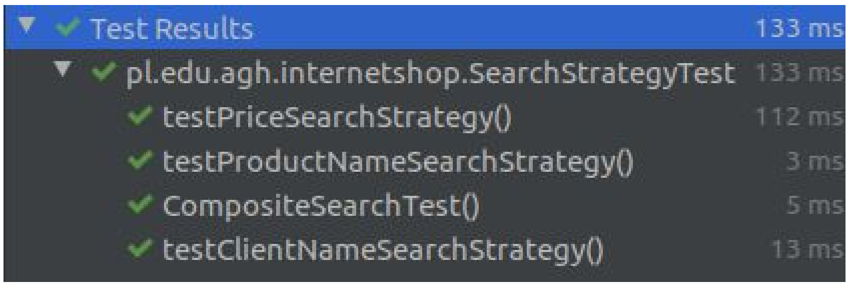


```java
@Test
public void testClientNameSearchStrategy() throws Exception {
    // given
    String wantedName = "Jan";

    ClientNameSearchStrategy searchStrategy = new ClientNameSearchStrategy(wantedName);

    Address address1 = mock(Address.class);
    given(address1.getName()).willReturn(wantedName);
    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("Stefan");
    Shipment shipment2 = mock(Shipment.class);
    given(shipment2.getRecipientAddress()).willReturn(address2);
    Order badOrder = mock(Order.class);
    given(badOrder.getShipment()).willReturn(shipment2);


    // when then
    assertTrue(searchStrategy.filter(goodOrder));
    assertFalse(searchStrategy.filter(badOrder));
}
```



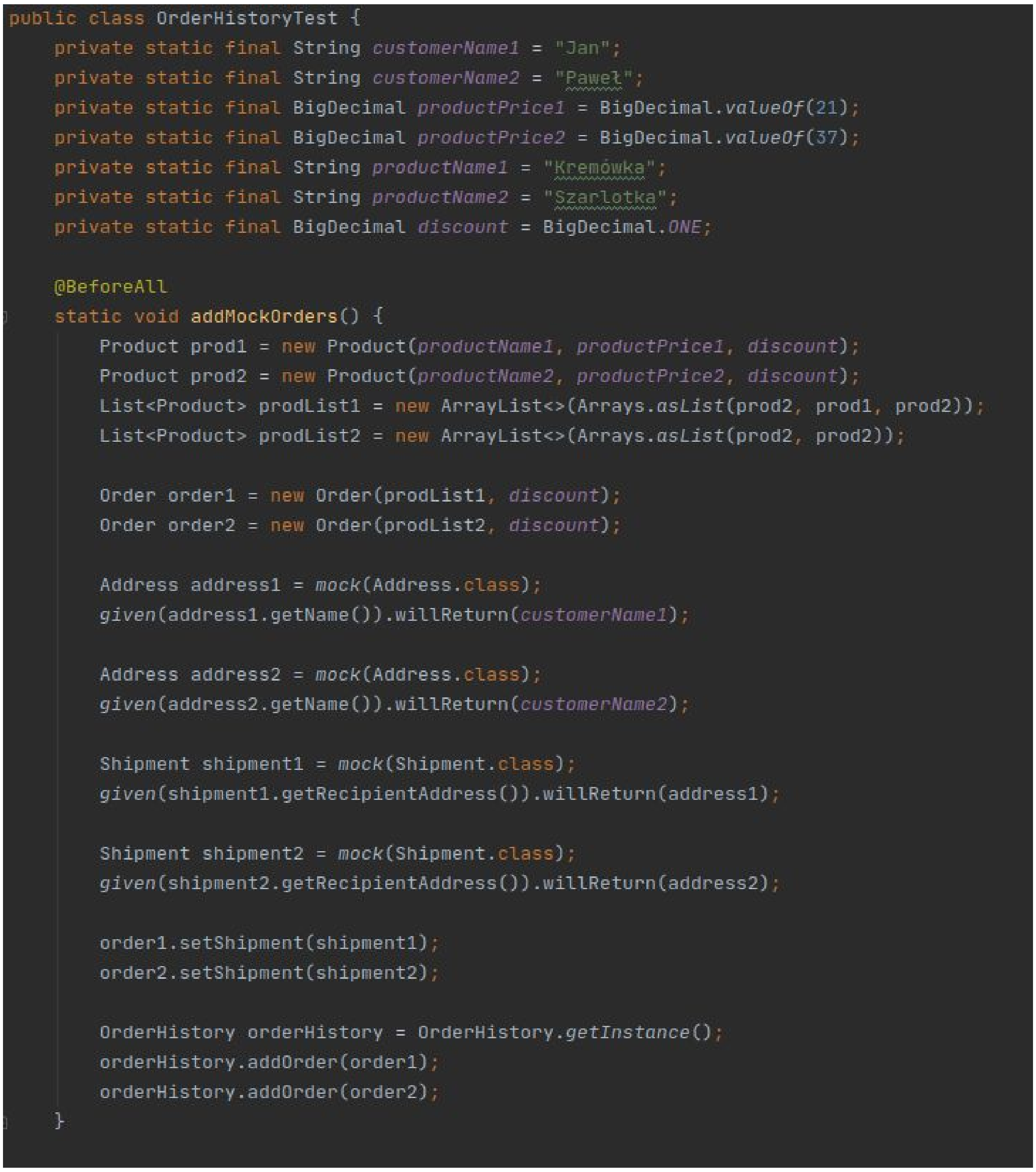
```java
@Test
public void testClientNameSearchStrategy() throws Exception {
    // given
    String wantedName = "Jan";

    ClientNameSearchStrategy searchStrategy = new ClientNameSearchStrategy(wantedName);

    Address address1 = mock(Address.class);
    given(address1.getName()).willReturn(wantedName);
    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("Stefan");
    Shipment shipment2 = mock(Shipment.class);
    given(shipment2.getRecipientAddress()).willReturn(address2);
    Order badOrder = mock(Order.class);
    given(badOrder.getShipment()).willReturn(shipment2);


    // when then
    assertTrue(searchStrategy.filter(goodOrder));
    assertFalse(searchStrategy.filter(badOrder));
}
```

Jak widać wszystkie testy przeszły.

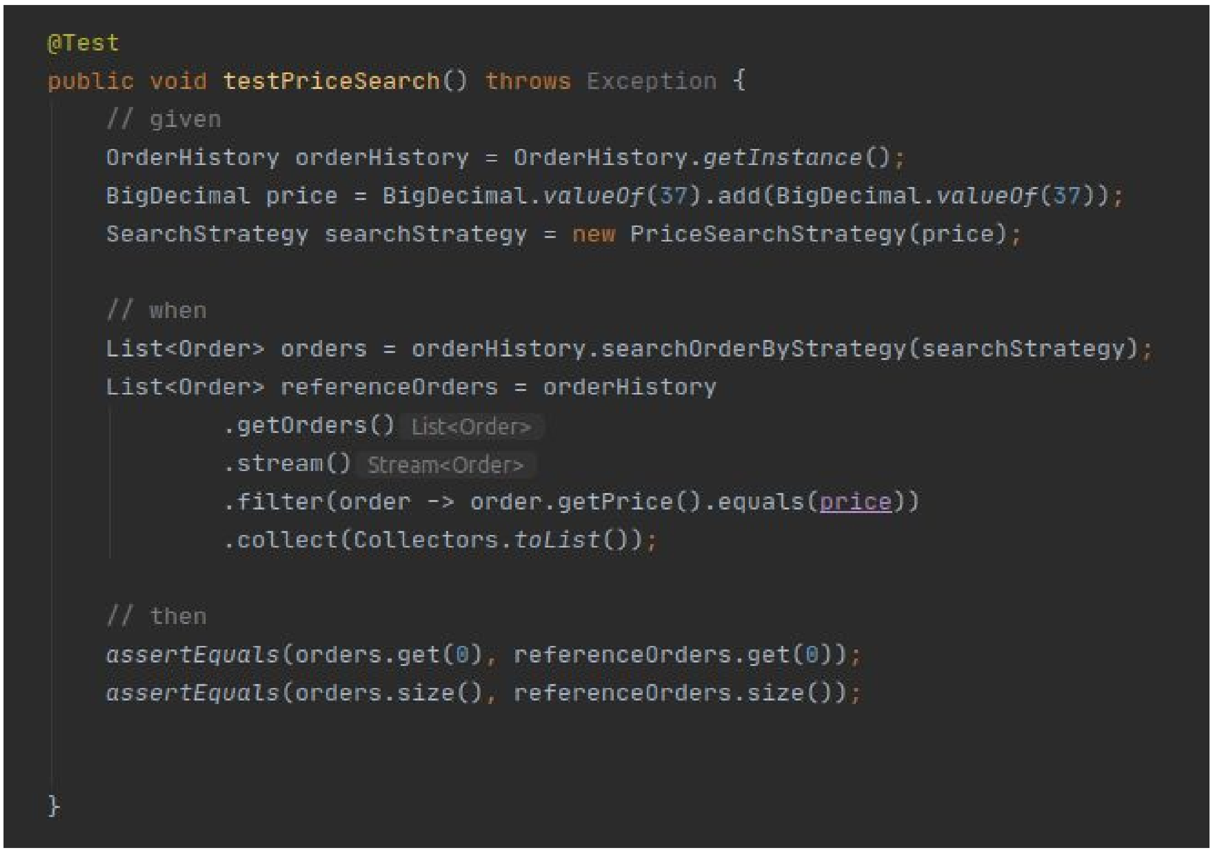


```
▼ ✓ Test Results                                      133 ms
  ▼ ✓ pl.edu.agh.internetshop.SearchStrategyTest      133 ms
      ✓ testPriceSearchStrategy()                     112 ms
      ✓ testProductNameSearchStrategy()                 3 ms
      ✓ CompositeSearchTest()                           5 ms
      ✓ testClientNameSearchStrategy()                 13 ms
```

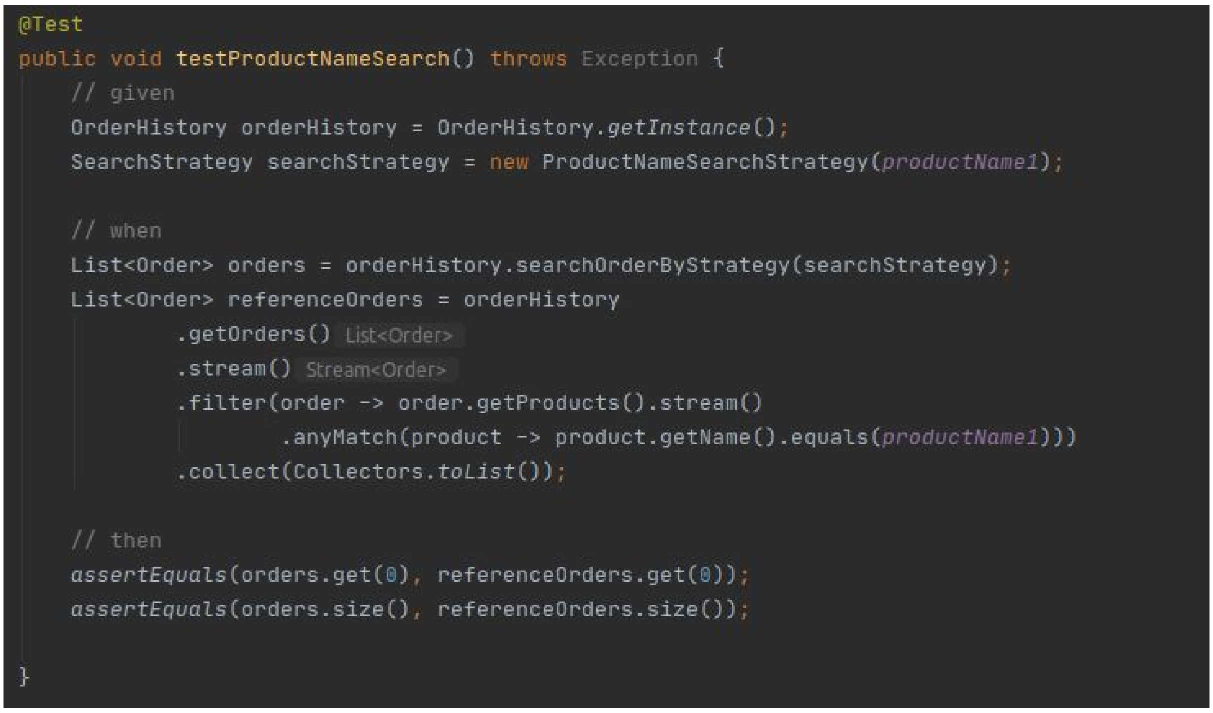
Na samym testy weryfikujące poprawne działanie OrderHistory.



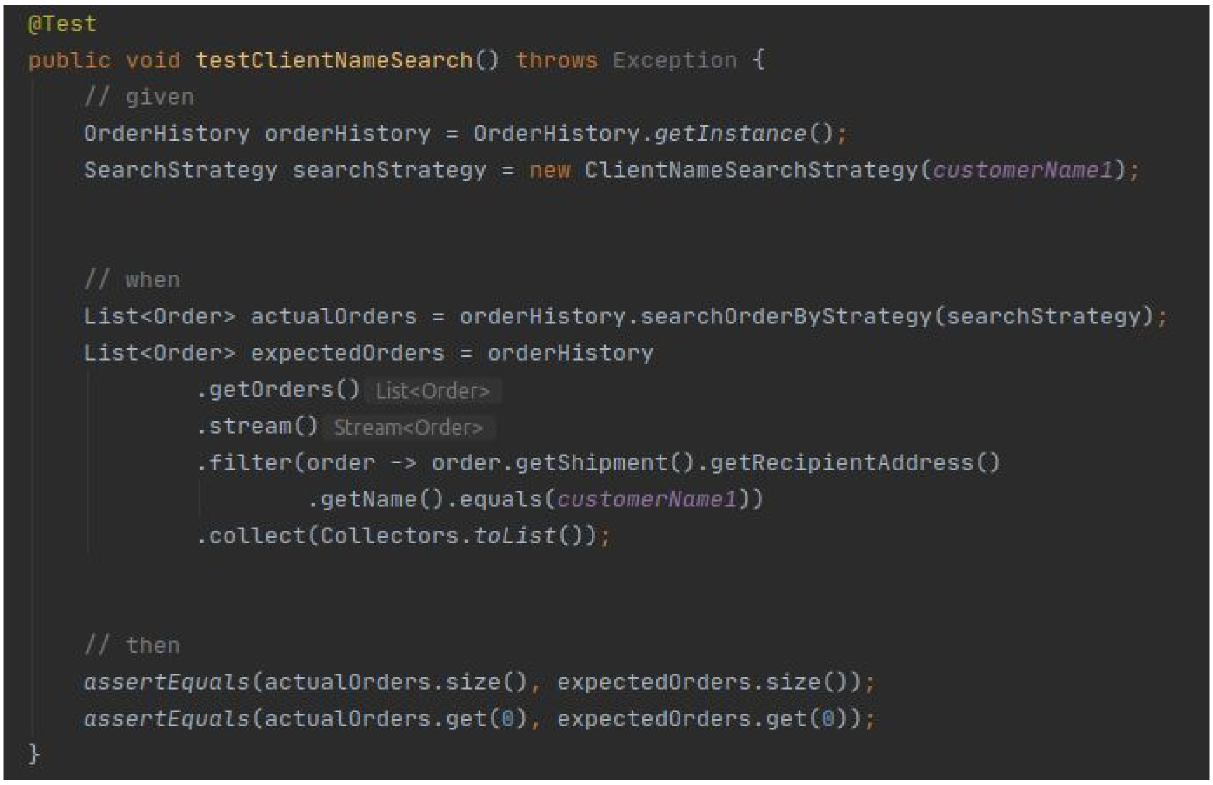
```java
public class OrderHistoryTest {
    private static final String customerName1 = "Jan";
    private static final String customerName2 = "Paweł";
    private static final BigDecimal productPrice1 = BigDecimal.valueOf(21);
    private static final BigDecimal productPrice2 = BigDecimal.valueOf(37);
    private static final String productName1 = "Kremówka";
    private static final String productName2 = "Szarlotka";
    private static final BigDecimal discount = BigDecimal.ONE;

    @BeforeAll
    static void addMockOrders() {
        Product prod1 = new Product(productName1, productPrice1, discount);
        Product prod2 = new Product(productName2, productPrice2, discount);
        List<Product> prodList1 = new ArrayList<>(Arrays.asList(prod2, prod1, prod2));
        List<Product> prodList2 = new ArrayList<>(Arrays.asList(prod2, prod2));

        Order order1 = new Order(prodList1, discount);
        Order order2 = new Order(prodList2, discount);

        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);

        OrderHistory orderHistory = OrderHistory.getInstance();
        orderHistory.addOrder(order1);
        orderHistory.addOrder(order2);
    }
```



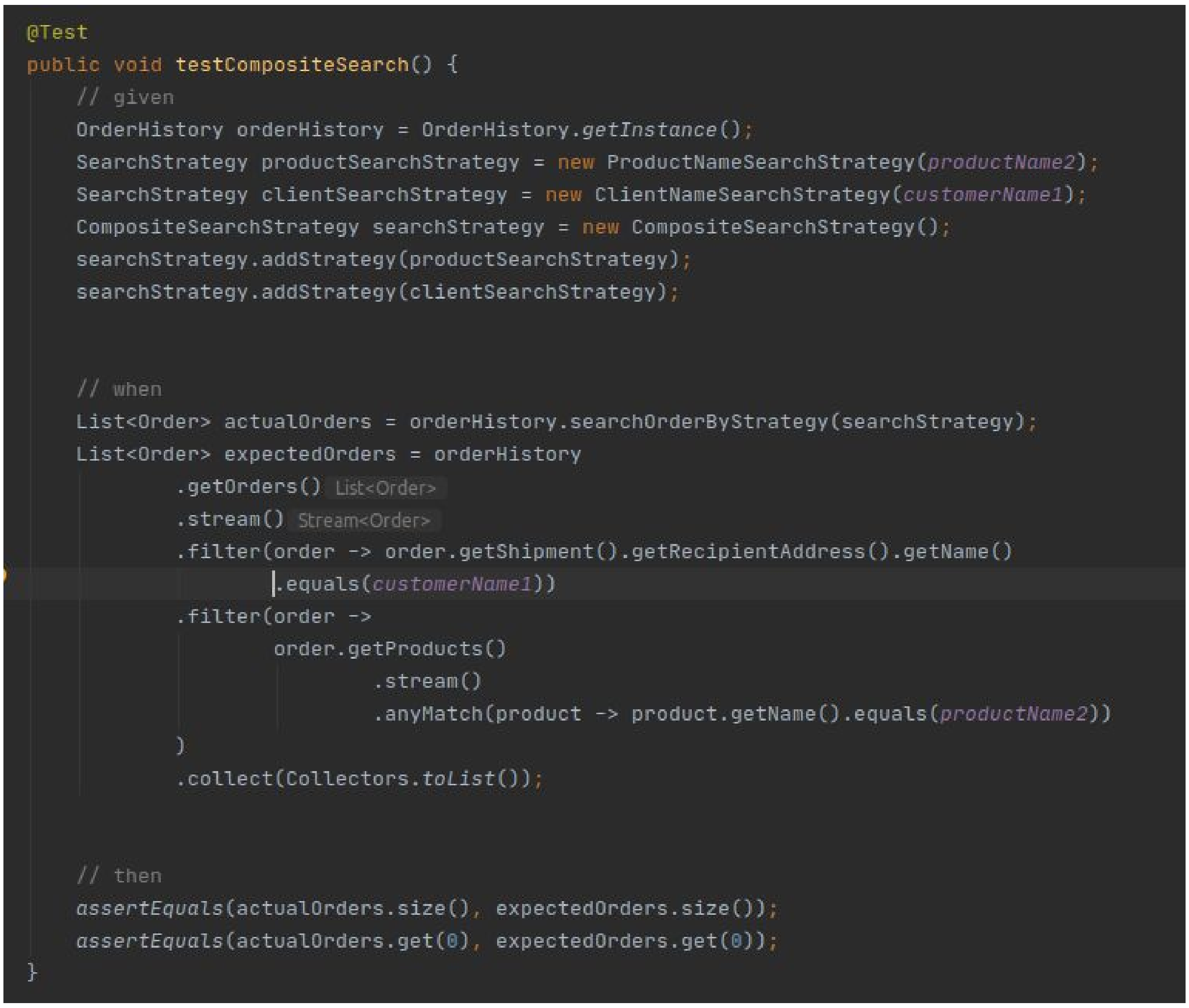
```java
@Test
public void testPriceSearch() throws Exception {
    // given
    OrderHistory orderHistory = OrderHistory.getInstance();
    BigDecimal price = BigDecimal.valueOf(37).add(BigDecimal.valueOf(37));
    SearchStrategy searchStrategy = new PriceSearchStrategy(price);

    // when
    List<Order> orders = orderHistory.searchOrderByStrategy(searchStrategy);
    List<Order> referenceOrders = orderHistory
            .getOrders() List<Order>
            .stream() Stream<Order>
            .filter(order -> order.getPrice().equals(price))
            .collect(Collectors.toList());

    // then
    assertEquals(orders.get(0), referenceOrders.get(0));
    assertEquals(orders.size(), referenceOrders.size());


}
```



```java
@Test
public void testProductNameSearch() throws Exception {
    // given
    OrderHistory orderHistory = OrderHistory.getInstance();
    SearchStrategy searchStrategy = new ProductNameSearchStrategy(productName1);

    // when
    List<Order> orders = orderHistory.searchOrderByStrategy(searchStrategy);
    List<Order> referenceOrders = orderHistory
            .getOrders() List<Order>
            .stream() Stream<Order>
            .filter(order -> order.getProducts().stream()
                    .anyMatch(product -> product.getName().equals(productName1)))
            .collect(Collectors.toList());

    // then
    assertEquals(orders.get(0), referenceOrders.get(0));
    assertEquals(orders.size(), referenceOrders.size());

}
```

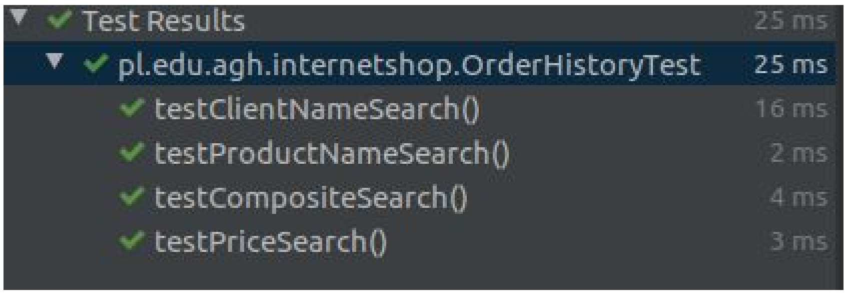


```java
@Test
public void testClientNameSearch() throws Exception {
    // given
    OrderHistory orderHistory = OrderHistory.getInstance();
    SearchStrategy searchStrategy = new ClientNameSearchStrategy(customerName1);


    // when
    List<Order> actualOrders = orderHistory.searchOrderByStrategy(searchStrategy);
    List<Order> expectedOrders = orderHistory
            .getOrders() List<Order>
            .stream() Stream<Order>
            .filter(order -> order.getShipment().getRecipientAddress()
                    .getName().equals(customerName1))
            .collect(Collectors.toList());


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



```java
@Test
public void testCompositeSearch() {
    // given
    OrderHistory orderHistory = OrderHistory.getInstance();
    SearchStrategy productSearchStrategy = new ProductNameSearchStrategy(productName2);
    SearchStrategy clientSearchStrategy = new ClientNameSearchStrategy(customerName1);
    CompositeSearchStrategy searchStrategy = new CompositeSearchStrategy();
    searchStrategy.addStrategy(productSearchStrategy);
    searchStrategy.addStrategy(clientSearchStrategy);


    // when
    List<Order> actualOrders = orderHistory.searchOrderByStrategy(searchStrategy);
    List<Order> expectedOrders = orderHistory
            .getOrders() List<Order>
            .stream() Stream<Order>
            .filter(order -> order.getShipment().getRecipientAddress().getName()
                    .equals(customerName1))
            .filter(order ->
                    order.getProducts()
                            .stream()
                            .anyMatch(product -> product.getName().equals(productName2))
            )
            .collect(Collectors.toList());


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

Testy dla OrderHistory również przeszły.



```
▼ ✓ Test Results                                25 ms
  ▼ ✓ pl.edu.agh.internetshop.OrderHistoryTest  25 ms
      ✓ testClientNameSearch()                  16 ms
      ✓ testProductNameSearch()                  2 ms
      ✓ testCompositeSearch()                    4 ms
      ✓ testPriceSearch()                        3 ms
```

Cały kod znajduje się w załączonym zipie.