Laboratorium 5
Adrian Wójciak, Lena Obertyńska

1. **Zmiana wartości procentowej naliczanego podatku**

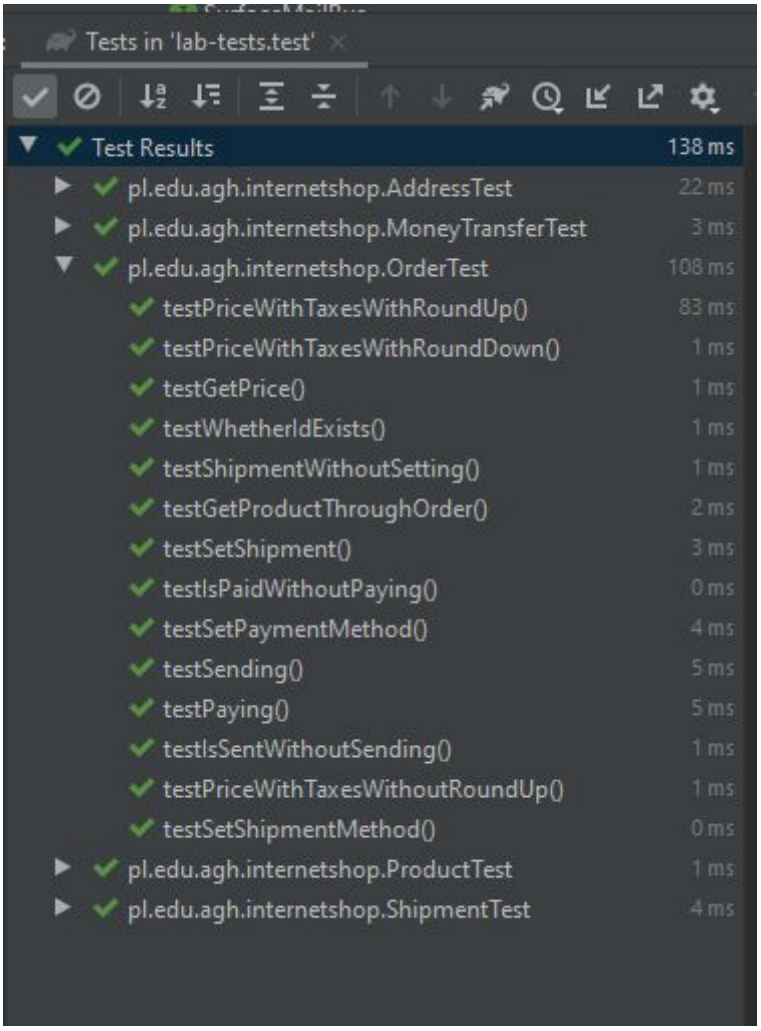
W klasie Order dokonano modyfikacji wartości procentowej podatku:

```
public class Order {
    private static final BigDecimal TAX_VALUE = BigDecimal.valueOf(1.23);
    private final UUID id;
    private final Product product;
    private boolean paid;
```

Dokonano też w klasie OrderTest modyfikacji testu testPriceWithTaxesWithoutRoundUp, by jego wynik uwzględniał powiększenie podatku:

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

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

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

Wyniki testów:



2. **Możliwość zamówienia więcej niż jednego produktu na raz**

W klasie Order dokonano zmiany własności “product” na listę produktów “products” w celu obsługi wielu produktów w jednym zamówieniu. Modyfikacji uległ też konstruktor, który w przypadku podanie pustej listy/nulla wyrzuca błąd, getter dla listy, oraz metoda getPrice, która teraz iteruje po liście.

```
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;

   public Order(List<Product> products) {
       if(products == null || products.isEmpty()) {
```

```
            throw new IllegalArgumentException("Lista musi zawierać
przynajmniej jeden produkt");
        }

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

    public UUID getId() {
        return id;
    }

    public void setPaymentMethod(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }

    public PaymentMethod getPaymentMethod() {
        return paymentMethod;
    }

    public boolean isSent() {
        return shipment != null && shipment.isShipped();
    }

    public boolean isPaid() { return paid; }

    public Shipment getShipment() {
        return shipment;
    }

    public BigDecimal getPrice() {
          BigDecimal price = BigDecimal.valueOf(0.0);

          for(Product product : this.products) {
              price = price.add(product.getPrice());
          }

          return price;
    }

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

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

    public ShipmentMethod getShipmentMethod() {
        return shipmentMethod;
    }

    public void setShipmentMethod(ShipmentMethod shipmentMethod) {
        this.shipmentMethod = shipmentMethod;
    }

    public void send() {
        boolean sentSuccesful = getShipmentMethod().send(shipment,
shipment.getSenderAddress(), shipment.getRecipientAddress());
        shipment.setShipped(sentSuccesful);
    }

    public void pay(MoneyTransfer moneyTransfer) {

moneyTransfer.setCommitted(getPaymentMethod().commit(moneyTransfer));
        paid = moneyTransfer.isCommitted();
    }

    public void setShipment(Shipment shipment) {
        this.shipment = shipment;
    }
}
```

Istniejące testy jednostkowe zmodyfikowano tak, by podawały do konstruktora klasy Order zamiast pojedynczego produktu listę go zawierającą, a także pobierały produkty ze zwracanych list.

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

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

  // when
```

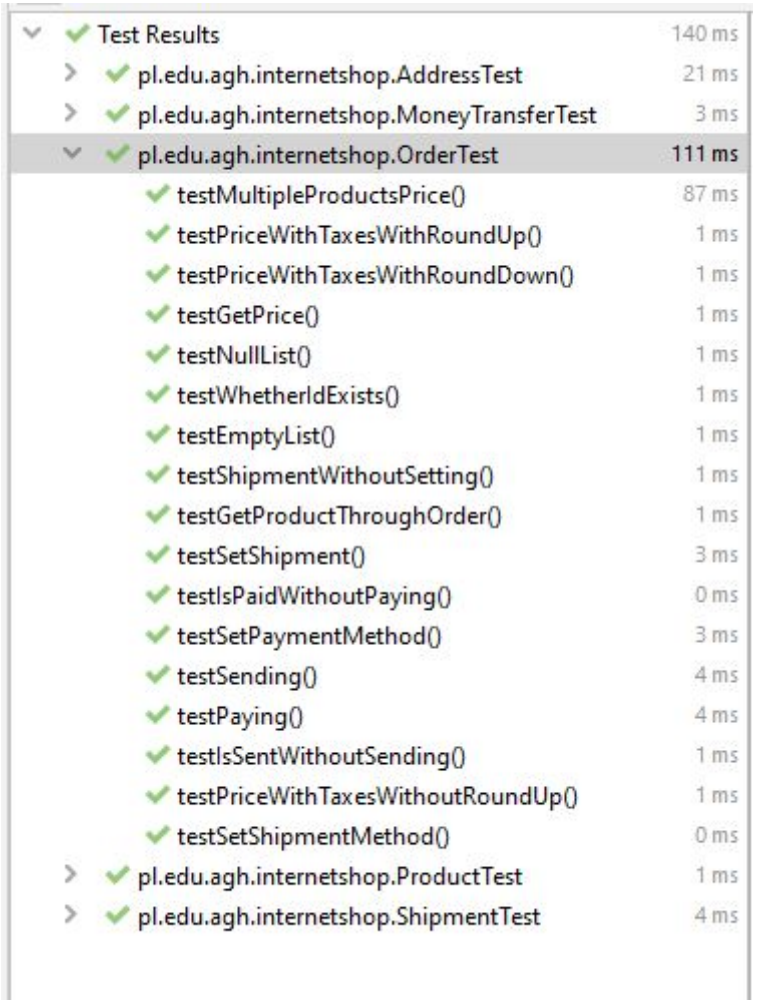
```
  Product actualProduct = (order.getProducts()).get(0);

  // then
  assertSame(expectedProduct, actualProduct);
}

@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);
}

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 jeszcze trzy nowe testy do klasy OrderTest. Pierwsze dwa sprawdzają, czy konstruktor klasy Order wyrzuci wyjące po podaniu listy pustej bądź będącej nullem. Trzeci sprawdza wyznaczanie ceny zamówienia przy ilości produktów większej od 1.

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

  // when

  // then
  assertThrows(IllegalArgumentException.class, () -> new Order(null));
}

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

```
  // when

  // then
  assertThrows(IllegalArgumentException.class, () -> new
Order(Collections.emptyList()));
}

@Test
public void testMultipleProductsPrice() throws Exception {
  // given
  Product expectedProduct1 = mock(Product.class);
  BigDecimal price1 = BigDecimal.valueOf(21.0);
  given(expectedProduct1.getPrice()).willReturn(price1);

  Product expectedProduct2 = mock(Product.class);
  BigDecimal price2 = BigDecimal.valueOf(0.37);
  given(expectedProduct2.getPrice()).willReturn(price2);

  Order order = new Order(Arrays.asList(expectedProduct1,
expectedProduct2));

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

  // then
  assertBigDecimalCompareValue(price1.add(price2), actualProductPrice);
}
```

Wyniki testów:



3. Możliwość naliczania rabatu

Do klasy Order dodano własność “discount” wraz z getterem i setterem (ustawioną domyślnie na 0 - brak rabatu), oraz zmodyfikowano metodę getPrice tak, by przy obliczaniu ceny odejmowała wartość rabatu.

```
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;
   private BigDecimal discount = BigDecimal.valueOf(0.0);

   public Order(List<Product> products) {
       if(products == null || products.isEmpty()) {
           throw new IllegalArgumentException("Lista musi zawierać
przynajmniej jeden produkt");
```

```
        }

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

    public UUID getId() {
        return id;
    }

    public void setPaymentMethod(PaymentMethod paymentMethod) {
        this.paymentMethod = paymentMethod;
    }

    public PaymentMethod getPaymentMethod() {
        return paymentMethod;
    }

    public boolean isSent() {
        return shipment != null && shipment.isShipped();
    }

    public boolean isPaid() { return paid; }

    public Shipment getShipment() {
        return shipment;
    }

    public BigDecimal getPrice() {
        BigDecimal price = BigDecimal.valueOf(0.0);

        for(Product product : this.products) {
            price = price.add(product.getPrice());
        }

        return price.subtract(price.multiply(discount));
    }

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

    public List<Product> getProducts() {
```

```
        return products;
    }

    public ShipmentMethod getShipmentMethod() {
        return shipmentMethod;
    }

    public void setShipmentMethod(ShipmentMethod shipmentMethod) {
        this.shipmentMethod = shipmentMethod;
    }

    public void send() {
        boolean sentSuccesful = getShipmentMethod().send(shipment,
shipment.getSenderAddress(), shipment.getRecipientAddress());
        shipment.setShipped(sentSuccesful);
    }

    public void pay(MoneyTransfer moneyTransfer) {

moneyTransfer.setCommitted(getPaymentMethod().commit(moneyTransfer));
        paid = moneyTransfer.isCommitted();
    }

    public void setShipment(Shipment shipment) {
        this.shipment = shipment;
    }

    public BigDecimal getDiscount() {
        return this.discount;
    }

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

Analogicznie postąpiono dla klasy Product.

```
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 BigDecimal discount = BigDecimal.valueOf(0.0);
```

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

    public String getName() {
        return name;
    }

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

    public BigDecimal getDiscount() {
        return this.discount;
    }

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

Do klasy OrderTest dodano testy sprawdzające ustawianie wartości zniżki oraz działanie medtody getPrice przy ustawionej zniżce. Ponadto dodano helper getOrderWithMultipleItems, by nie powtarzać tego samego kodu, i zastosowano go w jednym z wcześniej stworzonych testów:

```
public Order getOrderWithMultipleItems() {
  Product expectedProduct1 = mock(Product.class);
  BigDecimal price1 = BigDecimal.valueOf(21.0);
  given(expectedProduct1.getPrice()).willReturn(price1);

  Product expectedProduct2 = mock(Product.class);
  BigDecimal price2 = BigDecimal.valueOf(0.37);
  given(expectedProduct2.getPrice()).willReturn(price2);

  return new Order(Arrays.asList(expectedProduct1, expectedProduct2));
}

@Test
public void testMultipleProductsPrice() throws Exception {
  // given
  Order order = getOrderWithMultipleItems();
```

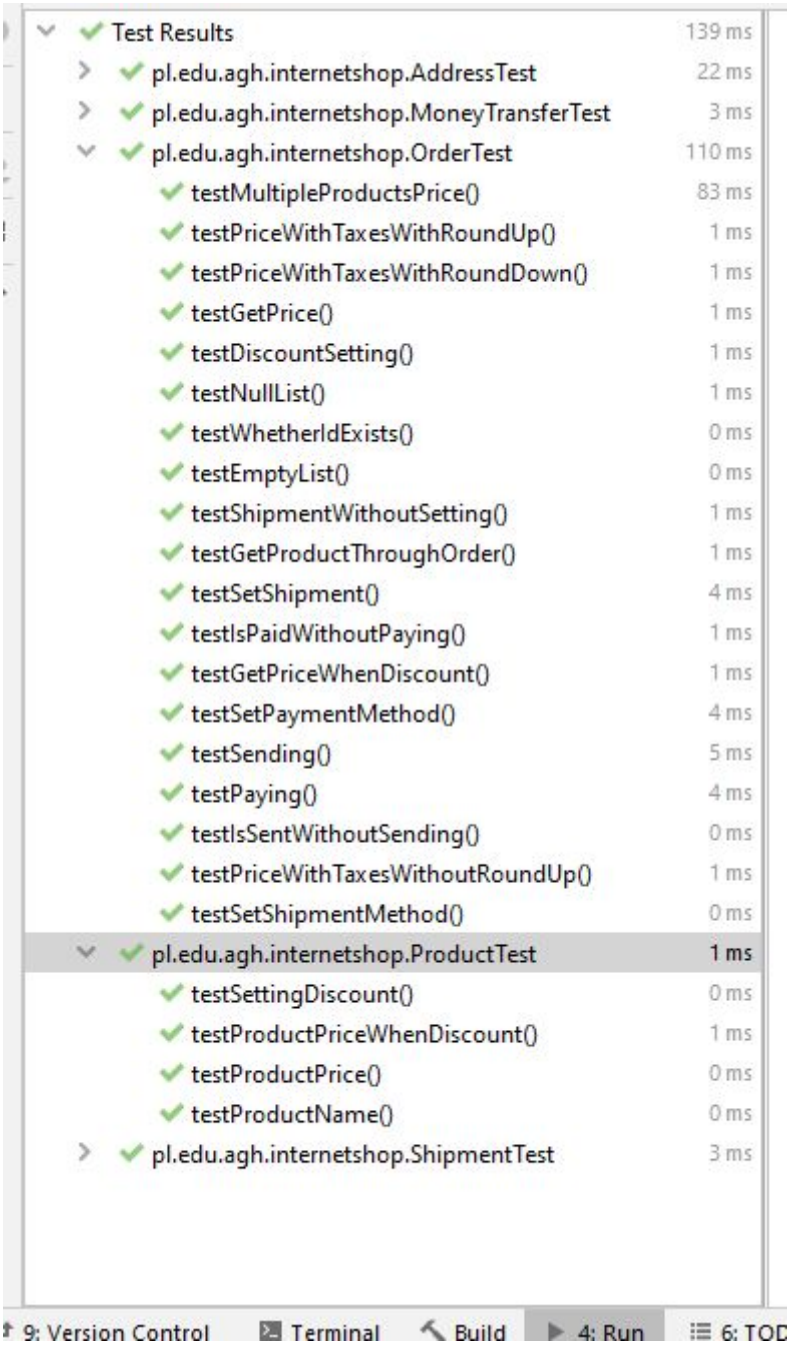
```
  // when
  BigDecimal actualProductPrice = order.getPrice();

  // then
  assertBigDecimalCompareValue(BigDecimal.valueOf(21.37),
actualProductPrice);
}

@Test
public void testDiscountSetting() throws Exception {
  // given
  Order order = getOrderWithMultipleItems();

  // when
  BigDecimal discount = BigDecimal.valueOf(0.911);
  order.setDiscount(discount);
  BigDecimal actualDiscount = order.getDiscount();

  // then
  assertBigDecimalCompareValue(discount, actualDiscount);
}

@Test
public void testGetPriceWhenDiscount() throws Exception {
  // given
  Order order = getOrderWithMultipleItems();

  // when
  BigDecimal discount = BigDecimal.valueOf(0.911);
  order.setDiscount(discount);
  BigDecimal actualPrice = order.getPrice();
  BigDecimal expectedPrice =
(BigDecimal.valueOf(21.37)).subtract((BigDecimal.valueOf(21.37)).multiply(
discount));

  // then
  assertBigDecimalCompareValue(expectedPrice, actualPrice);
}
```

Do klasy ProductTest także dodaję testy sprawdzające działanie ustawiania wartości zniżki oraz funkcji getPrice gdy zniżka jest niezerowa.

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

```
    // when
    Product product = new Product(NAME, PRICE);
    BigDecimal discount = BigDecimal.valueOf(0.911);
    product.setDiscount(discount);

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

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

    // when
    Product product = new Product(NAME, PRICE);
    BigDecimal discount = BigDecimal.valueOf(0.911);
    product.setDiscount(discount);
    BigDecimal expectedPrice = PRICE.subtract(PRICE.multiply(discount));

    // then
    assertBigDecimalCompareValue(product.getPrice(), expectedPrice);
}
```

Wyniki testów:



4. **Historia zamówień**

Stworzono interfejs Aggregate dla klas realizujących filtrowanie po kryteriach:

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

Stworzono klasy odpowiedzialne za filtrowanie produktów po nazwisku nabywcy, nazwie produktu i cenie:

```java
public class ClientNameAggregate implements Aggregate {
    private String name;

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

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

public class ProductNameAggregate implements Aggregate {
    private String name;

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

    @Override
    public boolean filter(Order order) {
        for(Product product : order.getProducts()) {
            if((product.getName()).equals(this.name)) {
                return true;
            }
        }

        return false;
    }
}

public class PriceAggregate implements Aggregate {
    private BigDecimal price;

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

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

Stworzono też klasę FullAggregate umożliwiającą filtrowanie zamówień przy użyciu wielu kryteriów:

```
public class FullAggregate implements Aggregate {
   List<Aggregate> aggregates;

   public FullAggregate(List<Aggregate> aggregates) {
       this.aggregates = aggregates;
   }

   @Override
   public boolean filter(Order order) {
       for(Aggregate aggregate : this.aggregates) {
           if (aggregate.filter(order) == false) {
               return false;
           }
       }

       return true;
   }
}
```

Następnie stworzona została klasa OrdersHistory umożliwiająca przechowywanie historii zamówień i wyszukiwanie przy użyciu agregatu:

```
public class OrdersHistory {
   private List<Order> history = new ArrayList<>();

   public List<Order> getHistory() {
       return this.history;
   }

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

   public List<Order> useAggregate(Aggregate aggregate) {
       List<Order> filtered = new ArrayList<>();

       for(Order order : this.history) {
           if(aggregate.filter(order)) {
               filtered.add(order);
           }
       }

       return filtered;
```

```
    }
}
```

Dla klas filtrujących stworzone zostały testy mające na celu weryfikację poprawności działania metod filtrowania:

```
public class ProductNameAggregateTest {
    public Order getTestOrder() {
        Product product = mock(Product.class);
        given(product.getName()).willReturn("Pomidor");
        return new Order(Collections.singletonList(product));
    }

    @Test
    public void ProductInOrderTest() throws Exception {
        // given
        Aggregate aggregate = new ProductNameAggregate("Pomidor");

        Order order = getTestOrder();
        // when

        // then
        assertTrue(aggregate.filter(order));
    }

    @Test
    public void ProductNotInOrderTest() throws Exception {
        // given
        Aggregate aggregate = new ProductNameAggregate("Ogórek");

        Order order = getTestOrder();
        // when

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

public class PriceAggregateTest {
    public Order getTestOrder() {
        Order order = mock(Order.class);
        given(order.getPrice()).willReturn(BigDecimal.valueOf(21.37));
        return order;
    }

    @Test
```

```
    public void PriceInOrderTest() throws Exception {
        // given
        Aggregate aggregate = new
PriceAggregate(BigDecimal.valueOf(21.37));

        Order order = getTestOrder();
        // when

        // then
        assertTrue(aggregate.filter(order));
    }

    @Test
    public void PriceNotInOrderTest() throws Exception {
        // given
        Aggregate aggregate = new PriceAggregate(BigDecimal.valueOf(9.11));

        Order order = getTestOrder();
        // when

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

public class ClientNameAggregateTest {
    public Order getTestOrder() {
        Address address = mock(Address.class);
        given(address.getName()).willReturn("Jan Kowalski");

        Shipment shipment = mock(Shipment.class);
        given(shipment.getRecipientAddress()).willReturn(address);

        Order order = mock(Order.class);
        given(order.getShipment()).willReturn(shipment);

        return order;
    }

    @Test
    public void ClientNameInOrderTest() throws Exception {
        // given
        Aggregate aggregate = new ClientNameAggregate("Jan Kowalski");

        Order order = getTestOrder();
        // when
```

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

    @Test
    public void ClientNameNotInOrderTest() throws Exception {
        // given
        Aggregate aggregate = new ClientNameAggregate("Janusz Nowak");

        Order order = getTestOrder();
        // when

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

public class FullAggregateTest {
    public Order getTestOrder() {
        Product product = mock(Product.class);
        given(product.getName()).willReturn("Pomidor");

        Address address = mock(Address.class);
        given(address.getName()).willReturn("Jan Kowalski");

        Shipment shipment = mock(Shipment.class);
        given(shipment.getRecipientAddress()).willReturn(address);

        Order order = mock(Order.class);
        given(order.getShipment()).willReturn(shipment);

given(order.getProducts()).willReturn(Collections.singletonList(product));
        return order;
    }

    @Test
    public void OrderPassesAllAggregatesTest() throws Exception {
        // given
        List<Aggregate> partialAggregates = Arrays.asList(new
ProductNameAggregate("Pomidor"), new ClientNameAggregate("Jan Kowalski"));
        Aggregate aggregate = new FullAggregate(partialAggregates);

        Order order = getTestOrder();
        // when
```

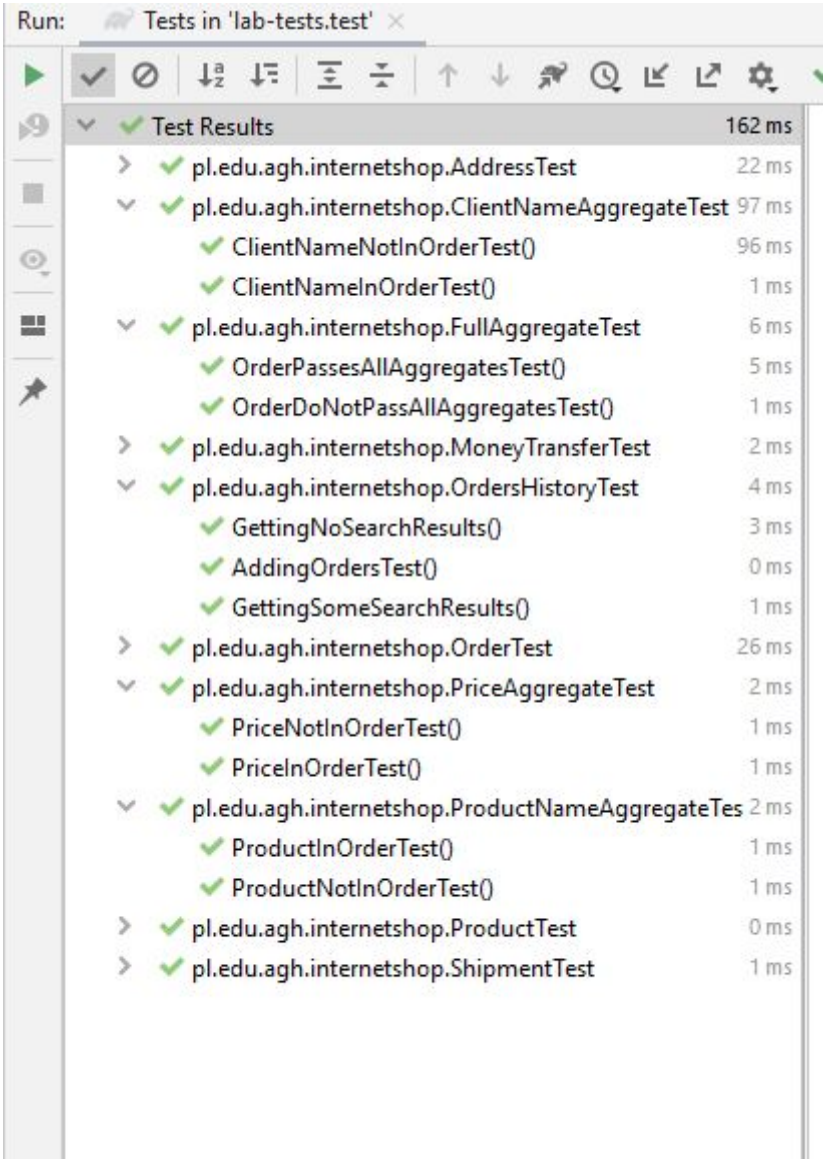
```
        // then
        assertTrue(aggregate.filter(order));
    }

    @Test
    public void OrderDoNotPassAllAggregatesTest() throws Exception {
        // given
        List<Aggregate> partialAggregates = Arrays.asList(new
ProductNameAggregate("Pomidor"), new ClientNameAggregate("Janusz Nowak"));
        Aggregate aggregate = new FullAggregate(partialAggregates);

        Order order = getTestOrder();
        // when

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

Na koniec zaś stworzone zostały testy mające na celu weryfikację działania klasy OrdersHistory:

```
public class OrdersHistoryTest {
    public Order getTestOrder(String productName) {
        Product product = mock(Product.class);
        given(product.getName()).willReturn(productName);

        return new Order(Collections.singletonList(product));
    }

    @Test
    public void AddingOrdersTest() throws Exception {
        // given
        Order order1 = mock(Order.class);
        Order order2 = mock(Order.class);

        OrdersHistory history = new OrdersHistory();
        history.addOrder(order1);
        history.addOrder(order2);
        // when

        // then
        assertEquals((history.getHistory()).size(), 2);
    }

    @Test
```

```
    public void GettingSomeSearchResults() throws Exception {
        // given
        Order order1 = getTestOrder("Pomidor");
        Order order2 = getTestOrder("Ogórek");

        OrdersHistory history = new OrdersHistory();
        history.addOrder(order1);
        history.addOrder(order2);

        Aggregate aggregate = new ProductNameAggregate("Ogórek");

        // when
        List<Order> results = history.useAggregate(aggregate);

        // then
        assertFalse(results.isEmpty());
    }

    @Test
    public void GettingNoSearchResults() throws Exception {
        // given
        Order order1 = getTestOrder("Pomidor");
        Order order2 = getTestOrder("Ogórek");

        OrdersHistory history = new OrdersHistory();
        history.addOrder(order1);
        history.addOrder(order2);

        Aggregate aggregate = new ProductNameAggregate("Dynia");

        // when
        List<Order> results = history.useAggregate(aggregate);

        // then
        assertTrue(results.isEmpty());
    }
}
```

Wyniki testów:


Run:
Tests in 'lab-tests.test'
Test Results 162 ms
pl.edu.agh.internetshop.AddressTest 22 ms
pl.edu.agh.internetshop.ClientNameAggregateTest 97 ms
ClientNameNotInOrderTest() 96 ms
ClientNameInOrderTest() 1 ms
pl.edu.agh.internetshop.FullAggregateTest 6 ms
OrderPassesAllAggregatesTest() 5 ms
OrderDoNotPassAllAggregatesTest() 1 ms
pl.edu.agh.internetshop.MoneyTransferTest 2 ms
pl.edu.agh.internetshop.OrdersHistoryTest 4 ms
GettingNoSearchResults() 3 ms
AddingOrdersTest() 0 ms
GettingSomeSearchResults() 1 ms
pl.edu.agh.internetshop.OrderTest 26 ms
pl.edu.agh.internetshop.PriceAggregateTest 2 ms
PriceNotInOrderTest() 1 ms
PriceInOrderTest() 1 ms
pl.edu.agh.internetshop.ProductNameAggregateTes 2 ms
ProductInOrderTest() 1 ms
ProductNotInOrderTest() 1 ms
pl.edu.agh.internetshop.ProductTest 0 ms
pl.edu.agh.internetshop.ShipmentTest 1 ms